Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Уральский федеральный университет имени первого Президента России Б. Н. Ельцина»

УТВЕРЖДАЮ

Директор по образовательной деятельности

_____ С.Т. Князев

«_7_» _сентября_ 2023 г.

Виртуализация и облачные технологии

Учебно-методические материалы по направлению подготовки
09.03.03 Прикладная информатика
Образовательная программа «Прикладной искусственный интеллект»

Екатеринбург

РАЗРАБОТЧИКИ УЧЕБНО-МЕТОДИЧЕСКИХ МАТЕРИАЛОВ

Доцент кафедры «Аналитика больших данных и методы видеоанализа»		Новиков Максим Юрьевич

СОДЕРЖАНИЕ

Раздел 1. Пояснительная записка к учебно-методическим материалам 4
Раздел 2. Список методических материалов 5

Раздел 1. Пояснительная записка к учебно-методическим материалам

Реализация дисциплины проводится с помощью учебно-методических материалов, предоставленных ПАО «Ростелеком». Материалы разделены на отдельные разделы (модули), которые имеют внутреннюю нумерацию, отличную от тематических разделов дисциплины «Виртуализация и облачные технологии».

Для сопоставления материалов ПАО «Ростелеком» и разделов дисциплины «Виртуализация и облачные технологии» следует использовать таблицу:

Код раздела, темы	**Раздел, тема дисциплины**	**Соответствующие модули учебно-методических материалов**
1	Облака и туман. Возможности и преимущества облачных технологий	Модуль 1. Введение в профессию Devops-инженер. Модуль 7. Масштабируемость и отказоустойчивость.
2	Облачные приложения и сервисы	Модуль 10. Облачные решения
3	Виртуализация и виртуальные машины	Модуль 9. Виртуализация в DevOps. Модуль 15. Практические навыки работы с Docker. Модуль 17. Оркестровка контейнеров, кластеры Kubernetes.
4	Хранение и управление данными в облаке	Модуль 4. Системы хранения данных. Модуль 5. Базы данных. Модуль 6. Администрирование баз данных.

Другие модули учебно-методических материалов следует использовать в случае, если базовой подготовки студентов по дополнительным темам недостаточно.

Раздел 2. Список методических материалов

Модуль 1. Введение в профессию Devops-инженер.

Модуль 2. Основы администрирования OS.

Модуль 3. Системы и сети передачи данных.

Модуль 4. Системы хранения данных.

Модуль 5. Базы данных.

Модуль 6. Администрирование баз данных.

Модуль 7. Масштабируемость и отказоустойчивость.

Модуль 8. Мониторинг, логирование и оповещение событий.

Модуль 9. Виртуализация в DevOps.

Модуль 10. Облачные решения.

Модуль 11. Работа с Terraform

Модуль 12. Конфигурационное управление.

Модуль 13. Системы контроля версий.

Модуль 14. Жизненный цикл ПО.

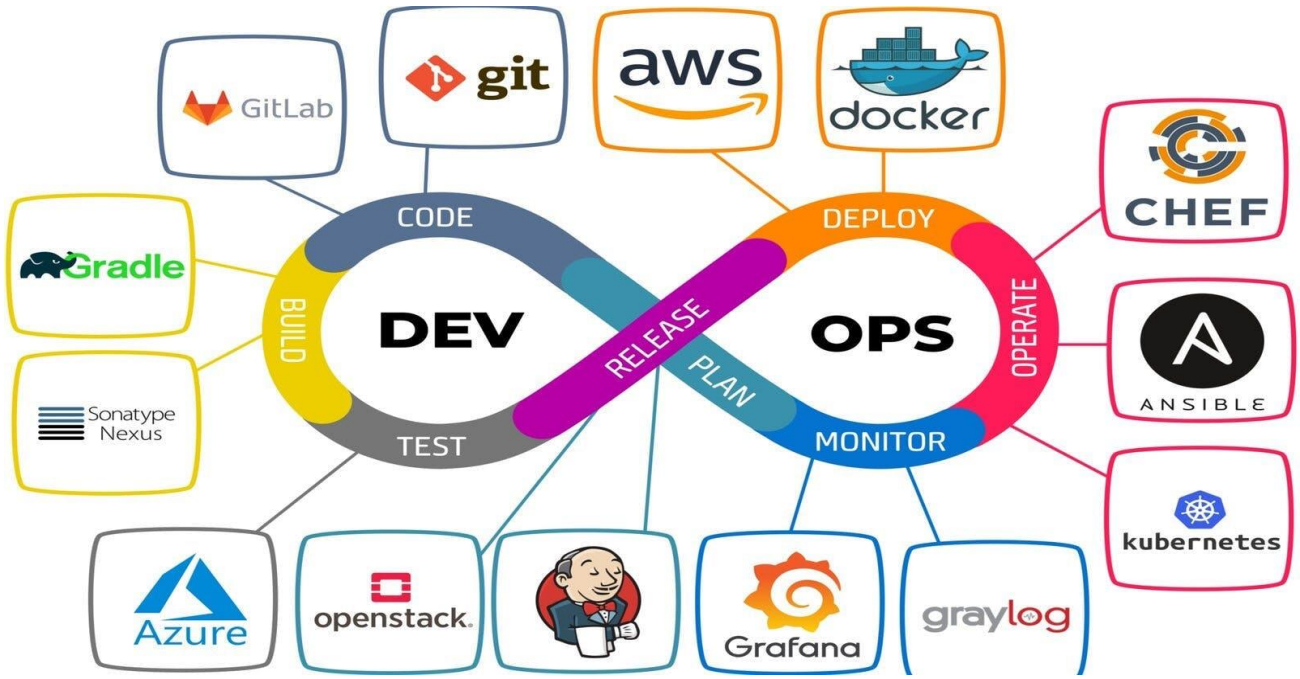
Модуль 15. Практические навыки работы с Docker.

Модуль 16. Микросервисы и микросервисная архитектура.

Модуль 17. Оркестровка контейнеров, кластеры Kubernetes.

Модуль 18. Kubernetes конфигурация развертывания.

Модуль 19. Планирование безопасности для кластера Kubernetes.



МОДУЛЬ 1. ВВЕДЕНИЕ В ПРОФЕССИЮ “DEVOPS-ИНЖЕНЕР”

1. Введение в профессию "DevOps-инженер"
2. Знакомство с основной терминологией
3. Цели методологии DevOps
4. Общее представление об инфраструктуре современной разработки
5. Российские и иностранные разработки в области DevOps

Некоторое время назад популярностью стал пользоваться DevOps. Инженеры, занимающиеся в соответствующей области, высоко ценятся. Но не совсем понятно, чем они занимаются. DevOps – весьма обширная сфера. Она способно объединить нескольких специалистов в единое целое ради выпуска и внедрения максимально выгодного и полезного потребителям продукта.

1. Введение в профессию "DevOps-инженер"

Что такое DevOps

Термин DevOps состоит из двух английских слов:

- development – развитие;
- operations – эксплуатация, использование.

Подобное понятие впервые возникло в 2009 году. Оно является относительно новым, как и профессия DevOps-инженер. И не все сразу понимают, о чем конкретно идет речь.

DevOps – суперсовременная технология. Это особая методология, которая предусматривает активное взаимодействие разработчиков, а также системных администраторов и IT-специалистов. Все это делается ради того, чтобы в итоге выпустить качественный продукт (приложение).

Из чего состоит

Важно понимать, что данное направление состоит из нескольких ключевых блоков. Для создания качественного приложения или иного продукта требуется их одновременное взаимодействие и реализация.

Всего у DevOps четыре составные части:

1. Культура. Это объединение развития и использования (эксплуатации) в единое целое. Чтобы успешно воплотить задумку в жизнь инженеру-DevOps предстоит проявить навыки управления кадрами. Ему нужно будет уладить конфликты между системными администраторами, разработчиками и программистами одновременно.
2. Автоматизация.
3. Измерения. В них потребность возникает не всегда, а лишь в ситуациях, при которых IT-системы дают сбой. Тогда компании зовут на помощь системного инженера. Но в DevOps, если приложение работает с ошибками, ситуация обстоит иначе. Приборы измерения типа всевозможных сеток устанавливаются сразу. Метрики же стараются внедрить непосредственно в код приложения (как будущего, так и текущего).
4. Обмен. Предполагается, что все решения относительно программирования, тестирования и так далее будут приниматься открыто.

Специфические знания для выбранного проекта находится в доступе для всей команды. Основная проблема заключается в том, что DevOps-инженеру придется уговорить каждого коллегу поделиться личными наработками и представить их остальным.

Если рассматриваемую методологию принять неграмотно, бизнесу не стоит ждать быстрого успеха. Развитие компании будет осуществляться очень медленно. Также создание приложения принесет определенные трудности – если кто-то из участников проекта займет лидирующие позиции, он сможет эгоистично настаивать и внедрять исключительно собственные наработки.

DevOps-инженер – это

В реальном мире полно перспективных и относительно новых профессий. Сделать карьеру DevOps-инженера может каждый, но сначала предстоит разобраться, кто это вообще такой. Уловить связь выбранного направления с аналитикой и программированием весьма проблематично.

DevOps- инженер — это многопрофильный специалист. Они:

- занимаются автоматизацией процессов;
- разбираются в принципах работы разработчиков и менеджеров;
- проявляют аналитические навыки.

Такой инженер объединяет всех членов команды при разработке приложений и иных продуктов в единое целое. Иногда соответствующую роль распределяют на нескольких людей.

Внимание: не стоит путать данную профессию с системным администрированием, а также со специалистами по BigData.

Главная задача DevOps -сотрудников – это максимальное ускорение, упрощение и автоматизация создания приложений и иного программного обеспечения на каждом этапе реализации. Начиная с написания кода, заканчивая непосредственным релизом утилиты.

Отличие от схожих направлений

DevOps освоить не так-то просто «с нуля», особенно если не понимать специфику профессии. В IT есть несколько «смежных» должностей. Чтобы не запутаться, кто есть кто, рекомендуется в первую очередь рассмотреть особенности каждого направления: системный администратор, BigData-администратор, DevOps-администратор.

По общим параметрам

Исторически перечисленные профили появились в совершенно разное время:

- системный администратор – в 80-х годах 20-го века;
- BigData-специалисты – 2010–2011 годы (вторая половина 21 века);

- DevOps-инженер — начало 21 века (примерно в 2000 году).

Также необходимо учитывать, что каждый из перечисленных работников берут на себя ответственность в совершенно разных областях деятельности. Может показаться, что они практически одинаковы, но это не так:

3. Сисадмины работают с программным и аппаратным обеспечением компьютеров. Они также занимаются периферийными устройствами, девайсами для настройки сетей.

3. BigData администраторы разбираются с кластерами типа Hadoop, облачными сервисами и виртуальными серверами.

3. DevOps-администраторы при создании приложения отвечают за конфигурацию рабочих, а также тестовых и Продакшн-серверов.

Области взаимодействия у перечисленных кадров тоже кардинально отличаются друг от друга:

- сисадмины решают свои вопросы с пользователями и руководителями;
- администраторы по большим данным сотрудничают с аналитиками, разработчиками программного обеспечения и инженерами BigData при создании приложений и поиске оптимальных корпоративных решений;
- DevOps-инженер поддерживает общение при разработке приложений с программистами, разработчиками и тестировщиками ПО.

Если компания нанимает DevOps-инженеров, они могут иногда замещать перечисленных сотрудников. Пример – проводить тестирование или разработку приложения «с нуля».

По задачам и целям

На сегодняшний день все перечисленные направления имеют собственные цели и задачи. Каждый сотрудник будет заниматься «чем-то своим». Это – самый простой способ разобраться в DevOps сфере.

Системные администраторы занимаются:

- работой с программным и аппаратным обеспечением, а также операционными системами;
- созданием и поддержкой пользовательских профилей/учетных записей;
- резервированием информации, а также ее проверкой и удалением;
- обеспечением информационной безопасности предприятия;
- работой с локальными сетями;

- администрированием серверов корпоративного типа;
- ремонтом периферийных устройств и аппаратных частей компьютеров;
- пользовательской поддержкой (помощью при решении вопросов, связанных с ноутбуками и ПК).

Big Data administrator отвечает за:

- IT-инфраструктуру для BigData-систем;
- конфигурации и управление облачных и локальных кластеров;
- настройку решений инфраструктурного характера, а также облачных сервисов;
- администрирование баз данных;
- развертывание так называемых ETL-систем;
- работу с корпоративными хранилищами данных;
- разработку и внедрение политики управления «большими объемами информации»;
- кластерную безопасность;
- производительность и балансировку нагрузки на серверы;
- резервное копирование информации с последующим восстановлением оной при необходимости.

DevOps инженеры отвечают за:

- развертку проектного релиза;
- конфигурирование, установку и обеспечение исправной работы веб-серверов;
- стандартизацию окружения разработки;
- подготовку среды, обеспечивающей легкое и простое внедрение корректировок в софт;
- создание непрерывной интеграции проекта;
- бесперебойную работу приложений и их доступность;
- масштабирование контента;
- резервное копирование/мониторинг серверов и разнообразных сервисов.

Также DevOps предусматривает одну очень полезную для предприятия цель – автоматизировать разработку ПО и иных продуктов по максимуму с минимальными потерями. Такой сотрудник будет проводить отслеживание жизненного цикла программы, а также при необходимости вносить корректировки.

Построение DevOps карьеры

Обычно для того, чтобы добиться определенных успехов в тех или иных направлениях деятельности, требуется заранее узнать об обучении, необходимых спектрах навыков, а также используемых специалистами инструментах. Если использовать эти познания грамотно, можно довольно быстро добиться успеха при продвижении по любой карьерной лестнице.

Необходимые знания

DevOps инженер должен:

- знать основы программирования (лучше изучить на базовом уровне несколько языков);
- освоиться в принципах работы операционных систем;
- понимать облачные и гибридные решения;
- разбираться в системах оркестрации;
- освоить принципы работы микросервисов;
- вникнуть в принципы работы с системами конфигурации.

Такой специалист отвечает за обновления контента, а также настраивает команду на слаженную и совместную плодотворную работу. Хороший сотрудник DevOps ищет подход к каждому «члену экипажа» и доводит имеющиеся процессы до предельной автоматизации без потерь.

Личностные качества

Хотя может показаться, что в популярном нынче направлении IT личностные качества не важны, это совсем не так. Любая инфраструктура предусматривает наличие не только определенного багажа знаний, но и человеческих «особенностей».

Так, DevOps специалист должен быть:

- усидчивым;
- стрессоустойчивым;
- талантливым стратегом и аналитиком;
- умельцем в общении с компанией и ее членами;
- восприимчивым к критике;
- с развитой логикой.

Каждой компании нужен сотрудник, который легко обучается и схватывает все налету. DevOps-специалист быстро проводит мониторинг ситуации и принимает те или иные решения. Он способен усваивать большой объем информации для дальнейшего использования на практике.

Инструменты для работы

Для того, чтобы добиться успехов в выбранном направлении, а также получать хорошую зарплату, пользователю предстоит постоянно изучать что-

то новое. В стандартных требованиях к DevOps специалистам имеются запросы относительно используемых инструментов. Это – приложения, без которых справляться с поставленными компанией задачами невозможно.

Начинающий и хороший специалист использует:

- Jenkins;
- системы контейнеризации Docker;
- оркестраторы контейнеров;
- платформы организации воркфлоу под названием GitLab;
- приложения для управления инфраструктурой;
- платформенные облачные сервисы;
- утилиты мониторинга и оповещений.

Теперь каждый более-менее понял, о каком работнике идет речь. Инженеру предстоит много практиковаться и учиться, набираться опыта в выбранном направлении. Иначе получать хорошую зарплату не представится возможным.

Обязанности

Когда DevOps специалист начинает свою карьеру, он должен хорошо понимать, что делает. В обязанности соответствующего лица входят следующие занятия:

- выбор среды оркестрации и архитектуры;
- автоматизация тестирования и иных этапов разработки приложения;
- настройка серверов;
- заливка кодификаций;
- проведение проверки среды после и до создания контента;
- настройка взаимосвязей и окружения между инструментарием для разработки софта;
- контроль за обеспечением постоянно функционирующей автоматизированной линии сборки ПО – конвейера.

Это – основные обязанности DevOps-специалиста. Но каждый случай в компании индивидуален. Поэтому спектр требований к работнику и его должностные «занятия» могут варьироваться.

Финансовый вопрос

Чтобы существовал спрос на ту или иную вакансию, ее необходимо достойно оплачивать. Выбранное направление не только перспективное, но и доходное.

Компании предлагают множество мест для трудоустройства в качестве DevOps-подчиненного. Новичкам удается получать в месяц порядка 90–100 тысяч рублей. По мере развития и накапливания опыта ставка увеличивается.

В среднем такой работник за свой труд получает 350–400 тысяч. Некоторые компании готовы платить от 500 000 рублей, но тем, кто уже имеет некоторый опыт в DevOpse.

Внимание: зарубежные компании предлагают более выгодные «версии» в плане оплаты труда DevOps-админам. Они в среднем платят 700–800 тысяч рублей в месяц. И это не предел.

Как стать специалистом

Если DevOps сфера интересна, стоит задуматься над тем, как начать развитие в данном направлении. Существуют различные варианты развития событий:

- Отправиться в ВУЗ на обучение. Лучше выбирать зарубежные высшие образовательные учреждения. Они обучают специалистов именно в сфере DevOps. В России остается надеяться на IT-специализации, а также информатику и программирование.
- Отдать предпочтение самообразованию. Можно заказать специализированную литературу в интернете (способы доставки обычно предусматривают курьерскую, что позволяет лишний раз не беспокоиться о получении книг), на основании которой происходит обучение. Но со временем подобного приема окажется мало.
- Посетить специализированные курсы. В Москве и других регионах образовательные центры подготавливают DevOps-специалистов за несколько месяцев. Иногда срок обучения составляет порядка года. Это – лучшее решение. Можно выбрать курсы для новичков или продвинутых работников. А еще – определиться с направлением DevOpsa.

Главное – не бояться, учиться и практиковаться. По мере накапливания опыта человек сможет развиваться и получать достойную зарплату.

2. Знакомство с основной терминологией

В этой главе мы собрали небольшой словарь часто употребляемых терминов и сокращений DevOps-инженерами.

Agile (Гибкая разработка программного обеспечения)- методология доставки программного обеспечения, основанная на коротких итерационных этапах разработки, где каждый спринт должен приводить к эксплуатационному продукту. Это позволяет легко корректировать требования проекта в случае необходимости и дает возможность творчества и гибкости в командах разработчиков.

Amazon AWS — Amazon Web Services — самый популярный поставщик облачных услуг (CSP) согласно отчету о состоянии DevOps за 2017 год, предлагающий широкий спектр услуг облачных вычислений для предприятий любого масштаба.

Apache — один из самых популярных веб-серверов с открытым исходным кодом (уступает только NGINX), кроссплатформенный инструмент для запуска веб-сайтов и приложений.

ALB — Балансировка нагрузки приложений — сервис AWS, используемый для разделения входящего трафика между несколькими экземплярами приложений в нескольких зонах доступности для обеспечения бесперебойной работы вашего приложения.

ASG — Auto Scaling Group — сервис AWS, используемый для объединения нескольких экземпляров EC2 в логические группы в целях проектирования инфраструктуры и простоты управления; группа состоит из идентичных экземпляров, которые добавляются или удаляются в соответствии с требованиями рабочей нагрузки.

AWS CLI — интерфейс командной строки AWS — инструмент AWS для управления различными сервисами и продуктами AWS из терминала командной строки.

Bastion host — специальный сервер, используемый для доступа к частным сетям и противостояния хакерским атакам. Обычно размещает одно приложение (например, прокси-сервер) и ключи SSH для доступа и управления базовой облачной инфраструктурой.

Ветвление — ветви представляют собой отдельные копии кода проекта на GitHub или другой системе контроля версий кода, что позволяет многим разработчикам работать над проектом одновременно.

Bucket — логическая единица в Amazon S3 (Simple Storage Service), используемая для хранения нескольких типов объектов (в основном, различных данных и метаданных, которые их описывают).

Back-end — программный движок, к которому у пользователя нет прямого доступа. Этот механизм получает запросы от пользовательского интерфейса и выполняет определенные операции, включая загрузку и сохранение данных в базе данных и т. д.

Сборка — это конкретная версия программного кода, в основном называемая этапом разработки новых функций. Наиболее важными являются сборки Canary, где новый код тестируется на соответствие существующим функциональным возможностям приложения в рабочей среде.

Bare-metal — случай, когда программное обеспечение установлено на физических устройствах (жестких дисках), пропуская уровень виртуализации.

Canary release — промежуточный сервер, который является точной копией производственной среды. Там запускаются новые сборки программного обеспечения, чтобы обеспечить соответствие существующим функциям и коду перед развертыванием их для всей пользовательской базы.

CI / CD — Непрерывная интеграция / Непрерывная доставка — основа современной культуры DevOps. CI гарантирует, что новый код передается в централизованное хранилище кода несколько раз в день, чтобы пройти автоматические модульные тесты и ускорить сборку нового программного обеспечения. Если тесты пройдены успешно, CD гарантирует, что новая версия приложения будет автоматически отправлена в промежуточную и производственную среды без простоев службы. Рабочий процесс CI / CD гарантирует, что все ошибки будут найдены и исправлены на ранней стадии, а продукт будет доступен в любое время.

Кластер — это набор взаимосвязанных экземпляров (серверы без поддержки, виртуальные машины, модули Kubernetes и т. Д.), Которые рассматриваются как единое целое для обеспечения балансировки нагрузки, автоматического масштабирования и высокой доступности.

Commit (Комит) — процесс отправки кода в репозиторий Git и полученный фрагмент кода.

Управление конфигурацией — процесс установки и поддержания требуемых параметров программной экосистемы с помощью инструментов автоматического управления конфигурацией, таких как Kubernetes, Ansible, Puppet, Chef, Saltstack и т. Д.

Dark launch (Темный запуск) — практика выпуска кода в производственную среду без уведомления пользователей о новых доступных

функциях. Код запускается в производство для заключительного этапа тестирования, затем объявляется о выпуске новой функции, а сама функция уже доступна.

Docker — платформа с открытым исходным кодом для создания, доставки и запуска контейнеров приложений. Docker является основой современных облачных вычислений, поскольку он позволяет максимально эффективно использовать облачные ресурсы, обеспечивая повсеместный уровень для построения облачной инфраструктуры.

Фреймворк Django — это высокоуровневый фреймворк Python, ориентированный на чистый дизайн, быструю разработку и высокую производительность приложений. Нашел широкое применение в веб-разработке и обработке больших данных.

Datadog — эффективный сервис облачного мониторинга, позволяющий анализировать процессы в любой инфраструктуре, базе данных или приложении в любом масштабе, используя платформу на основе SaaS.

Среда (Environment) — все ресурсы сервера (ОС, библиотеки, API, инструменты и платформы и т. Д.), Необходимые для запуска программного обеспечения на различных этапах его жизненного цикла (разработка, тестирование, подготовка, производство).

ElasticSearch — RESTful, распределенный движок для поиска и анализа данных, построенный на Apache Lucene. Как ядро стека Elastic, Elasticsearch позволяет хранить и обрабатывать данные из нескольких облачных инструментов мониторинга и ведения журналов.

Envoy — мощный прокси C ++ для обработки трафика между микросервисами.

FluentD — инструмент для сбора и обработки данных с открытым исходным кодом, написанный на Ruby. Он позволяет вводить данные из огромного количества инструментов, таких как ElasticSearch, и выводить данные на широкий выбор панелей мониторинга, настроенных с использованием нескольких плагинов.

Fargate — Amazon Fargate — это сервис Amazon для запуска контейнеров Docker в управляемой инфраструктуре, такой как EKS, без необходимости что-либо настраивать. Он работает по схеме выставления счетов без сервера — вы указываете, что необходимо сделать, и оплачиваете потребляемые ресурсы без какой-либо ручной настройки кластера.

Helm — менеджер приложений, работающий поверх Kubernetes. Этот инструмент позволяет управлять микросервисами в масштабе с помощью

удобных диаграмм Хелма и обеспечивает бесперебойную работу сложной инфраструктуры Kubernetes.

Инфраструктура — весь комплекс оборудования, программного обеспечения и процессов, необходимых для запуска приложений, а также для сбора, управления и хранения данных.

IaC — Инфраструктура как код — один из базовых принципов DevOps. Это означает, что конфигурация инфраструктуры выполняется с помощью машиночитаемых декларативных файлов, а не вручную или с использованием интерактивных инструментов. Эти файлы (например, манифесты Kubernetes или Terraform) могут храниться в репозиториях GitHub, настраиваться и корректироваться так же, как и код, что обеспечивает эффективную автоматизацию обеспечения инфраструктуры.

IaaS — инфраструктура как услуга, модель управления ИТ, в которой вычислительные ресурсы и службы, необходимые для их работы, предоставляются в качестве службы, обеспечивающей функционирование различных платформ и приложений.

Пропускная способность ввода / вывода — количество операций ввода / вывода в секунду, характеристика пропускной способности сети или накопителя.

Ingress controller — программный модуль, используемый для обеспечения балансировки нагрузки в модулях Kubernetes.

Jenkins — Java-сервер с открытым исходным кодом, позволяющий автоматизировать доставку программного обеспечения.

Работа в Jenkins — процесс в Jenkins, необходимый для сборки кода, запуска модульных тестов, генерации метрик качества кода, развертывания новых версий приложений в рабочей среде и т. д.

Kibana — часть стека Elastic, отвечающая за визуализацию данных и навигацию по кластеру ELK.

Logstash — часть стека Elastic, отвечающая за сбор, обработку и передачу данных на сервер. Logstash необходим для создания решений для облачного мониторинга.

Lead time (Время выполнения) — время, необходимое для перемещения нового пакета кода из коммита в релиз.

MTTR — среднее время до восстановления — среднее ожидаемое время, когда отказавший системный компонент снова заработает; основной параметр сценариев восстановления после сбоев, системного стресс-тестирования и проверки производительности.

Node (узел, нода) — физическая или виртуальная машина в кластере Kubernetes, используемая для размещения модулей, которые запускают контейнеры Docker.

Node pool (Пул узлов) — пул узлов Kubernetes — это группа из нескольких точек кластера, объединяющих машины с одинаковой конфигурацией, которые по сути можно рассматривать и управлять ими как одним объектом.

Nexus3 — платформа управления выпуском от Sonatype, созданная для объединения входных сигналов от нескольких модулей с открытым исходным кодом, чтобы обеспечить быстрый, безопасный и эффективный жизненный цикл доставки программного обеспечения.

OpenStack — платформа с открытым исходным кодом для создания локальных облачных инфраструктур.

OpenShift — платформа управления контейнерами корпоративного уровня для Kubernetes, работающая в локальных облачных инфраструктурах, разработанная Red Hat.

PaaS — платформа как услуга, модель доставки программного обеспечения, когда разработчики получают все необходимые библиотеки, инструменты и сервисы для разработки программного обеспечения со всей базовой инфраструктурой, которая обрабатывается платформой, предоставляющей сервис.

Prometheus — решение для облачного мониторинга с открытым исходным кодом с мощным языком запросов, базой данных временных рядов, моделью данных измерений и возможностями интеллектуального оповещения.

Pod — базовое структурное подразделение Kubernetes, группа контейнеров Docker, развернутых на одном хосте.

Playbook — Ansible playbook — это инструкции по развертыванию инфраструктуры с подробными руководствами по выполнению серии команд для выполнения конкретных задач.

ProxMox — основанная на Debian платформа с открытым исходным кодом для развертывания и управления виртуальными машинами.

Production environment (PROD, Производственная среда)- среда, в которой программный продукт или услуга используется целевой аудиторией.

Rollback (Откат) — ручное или автоматическое восстановление ранее сохраненного состояния программы или базы данных.

Regression testing (Регрессионное тестирование) — комплексное тестирование обновленной версии продукта, чтобы убедиться, что последняя

сборка не оказала негативного влияния на уже имеющуюся функциональность.

RabbitMQ — брокер сообщений, программное обеспечение, используемое для сбора сообщений из ваших приложений и их хранения до тех пор, пока они не потребуются для операций.

S3 — Amazon Simple Storage Service — сервис облачных вычислений для хранения любых объектов данных, необходимых для стабильной работы ваших приложений.

Snapshot (Amazon EBS snapshot EBS) — команда для создания статической копии содержимого вашего экземпляра EC2 в целях резервного копирования и восстановления.

Staging environment (Stage, промежуточная среда) — контролируемая копия вашей производственной среды, максимально напоминающая ее. Это позволяет тестировать новые версии программного обеспечения, чтобы находить ошибки перед выпуском в эксплуатацию.

Technical debt (Технический долг) — концепция нежелательного количества работы разработчиков, необходимого для исправления простого кода, используемого для быстрого получения результатов вместо того, чтобы тратить время на разработку и внедрение наилучшего решения.

Модульное тестирование (Unit testing) — основа CI / CD, модульное тестирование — это практика тестирования кода приложения небольшими блоками на основе кода автоматизированного тестирования перед сборкой приложения, чтобы минимизировать время, необходимое для обнаружения и исправления ошибок, сокращая время выхода на рынок.

VPC peering — AWS VPC — это сервис, который логически изолирует определенное количество общедоступного облака AWS для создания виртуальных частных облаков. AWS VPC peering позволяет объединить ресурсы нескольких таких облаков в случае необходимости.

Vault — продукт Hashicorp для безопасного хранения таких секретов, как ключи SSH, токены, пароли, ключи API и другие важные элементы инфраструктуры Kubernetes.

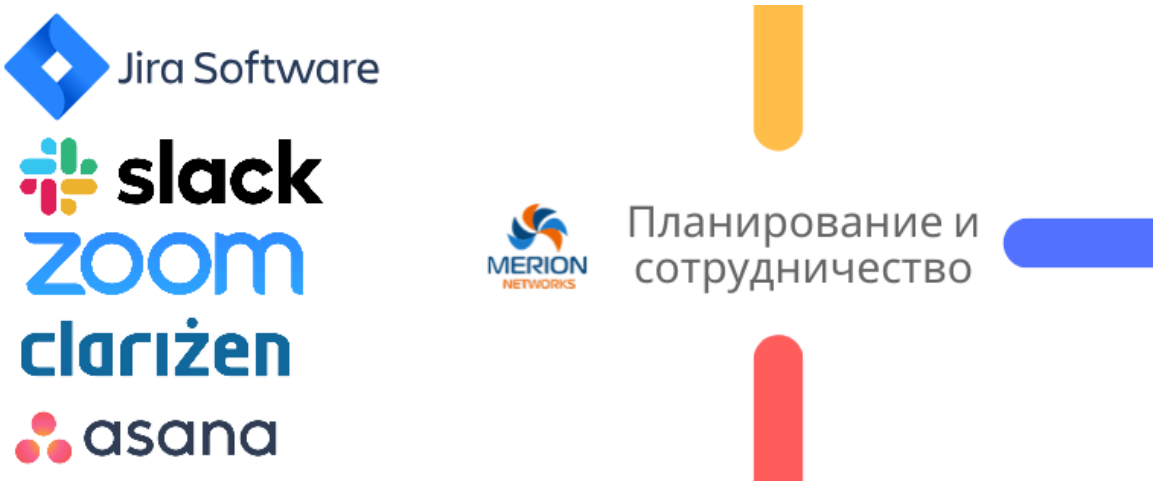
Zabbix — сервис мониторинга облачной инфраструктуры с открытым исходным кодом для отслеживания состояния различных сетевых ресурсов и сервисов. Состоит из сервера и агентов, которые обеспечивают интеллектуальное оповещение для распределенных систем.

Инструменты devops

Основная часть работы DevOps инженера приходится на этап выпуска релиза — поставки продукта заказчику. В центре внимания находится производительность всех потоков процесса доставки. Такой специалист следит за тем, чтобы известные баги никогда не передавались на следующий этап работ, никогда не развивалась локальная оптимизация, приводящая к созданию глобальной деградации.

Какой инструментарий используется для выполнения данной задачи и работы - зависит от разрабатываемого программного продукта, его архитектуры, языка и принципов разработки и собственно квалификации команды разработки.

Планирование и сотрудничество



Jira

JIRA — это один из популярных инструментов управления проектами, разработанный Atlassian, который используется для отслеживания проблем, ошибок и проектов. Это позволяет пользователю отслеживать проект и выпускать статус. Его можно легко интегрировать с другими продуктами Atlassian, такими как Bitbucket, в дополнение к другим инструментам DevOps, таким как Jenkins.

Slack

Slack — это бесплатный инструмент для совместной работы на основе облака, для групповой коммуникации и совместной работе в одном месте. Этот инструмент также можно использовать для обмена документами и другой информацией среди членов команды. Это также может быть легко интегрировано с другими инструментами, такими как GIT, Jenkins, JIRA и так далее. А еще Slack можно интегрировать с Asterisk.

Zoom

Zoom — это платформа для веб-конференций и мгновенного обмена экранами. Вы можете общаться со своей командой через аудио или видео. Неважно, насколько велика ваша команда, Zoom способен принять до 1000 получателей на онлайн-встречу.

Clarizen

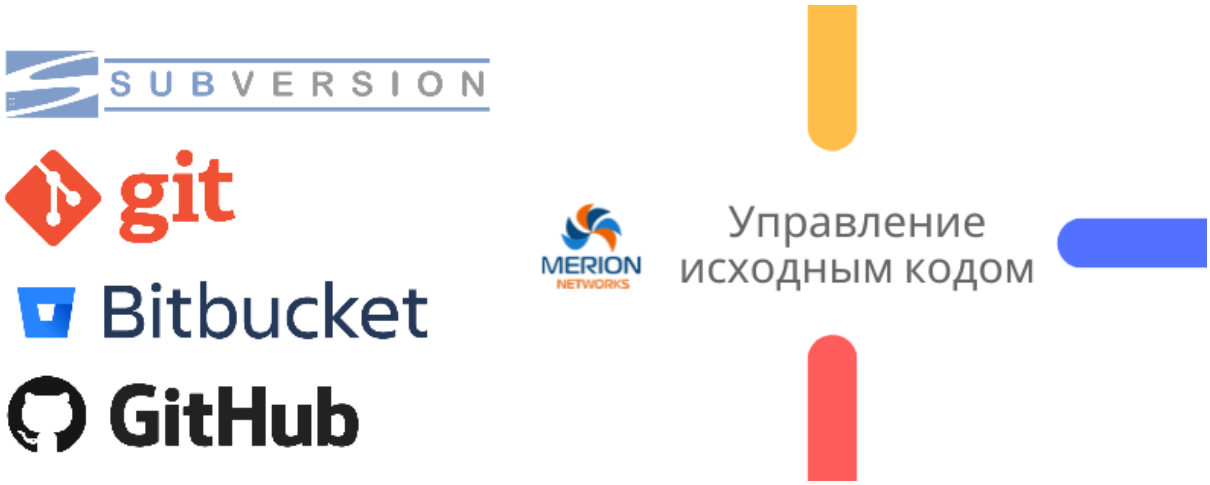
Clarizen — это программное обеспечение для совместной работы и управления проектами, которое помогает отслеживать проблемы, управлять задачами и управлять портфелем проектов. Clarizen легко настроить, и он имеет удобный интерактивный интерфейс пользователя.

Asana

Asana — это мобильное и веб-приложение, которое помогает командам эффективно и результативно организовывать, отслеживать и управлять своей работой. Он используется для отслеживания ежедневных командных задач и поддержки обмена сообщениями и общения в организации.

Управление исходным кодом

Система управления исходным кодом (SCCS) — это система контроля версий, предназначенная для отслеживания изменений в исходном коде и других текстовых файлах во время разработки программного обеспечения



SVN

SVN — это централизованный инструмент контроля версий и исходного кода, разработанный Apache. SVN помогает разработчикам поддерживать разные версии исходного кода и вести полную историю всех изменений.

Git

Git — это распределенная система контроля версий, которая нацелена на скорость, целостность данных, поддержку распределенных, нелинейных рабочих процессов. Помимо управления исходным кодом, его также можно использовать для отслеживания изменений в любом наборе файлов.

Bitbucket

Bitbucket — это веб-хостинговая платформа, разработанная Atlassian. Bitbucket также предлагает эффективную систему проверки кода и отслеживает все изменения в коде. Его можно легко интегрировать с другими инструментами DevOps, такими как Jenkins, Bamboo.

Github

GitHub — это платформа для размещения кода, предназначенная для контроля версий и совместной работы. Он предлагает все функции распределенного контроля версий и управления исходным кодом (SCM) в Git в дополнение к своим функциям.

GitHub предлагает функции контроля доступа и совместной работы, такие как отслеживание ошибок, создание функций и запросов, управление задачами и так далее.

Инструменты для сборки

Инструменты сборки — это инструменты для управления и организации ваших сборок, которые очень важны в средах, где есть много проектов, особенно если они взаимосвязаны. Они служат для того, чтобы убедиться, что там, где разные люди работают над различными проектами, они ничего не ломают. И чтобы убедиться, что когда вы вносите свои изменения, они тоже ничего не ломают.



ANT

Apache Ant - это инструмент для сборки и развертывания на основе Java с открытым исходным кодом. Он поддерживает формат файла XML. Он имеет несколько встроенных задач, позволяющих нам компилировать, собирать, тестировать и запускать приложения Java.

Maven

Maven - это инструмент для автоматизации сборки, который в основном используется для Java-проектов. Он содержит файл XML, в котором описывается создаваемый программный проект, его зависимости от других внешних компонентов и модулей, последовательность сборки, каталоги и другие необходимые подключаемые модули.

Grunt

Grunt - это инструмент командной строки Javascript, который помогает создавать приложения и помогает разработчикам автоматизировать повторяющиеся задачи, такие как компиляция, модульное тестирование, кодирование кода, проверка и так далее. Это хорошая альтернатива для таких инструментов, как Make или Ant.

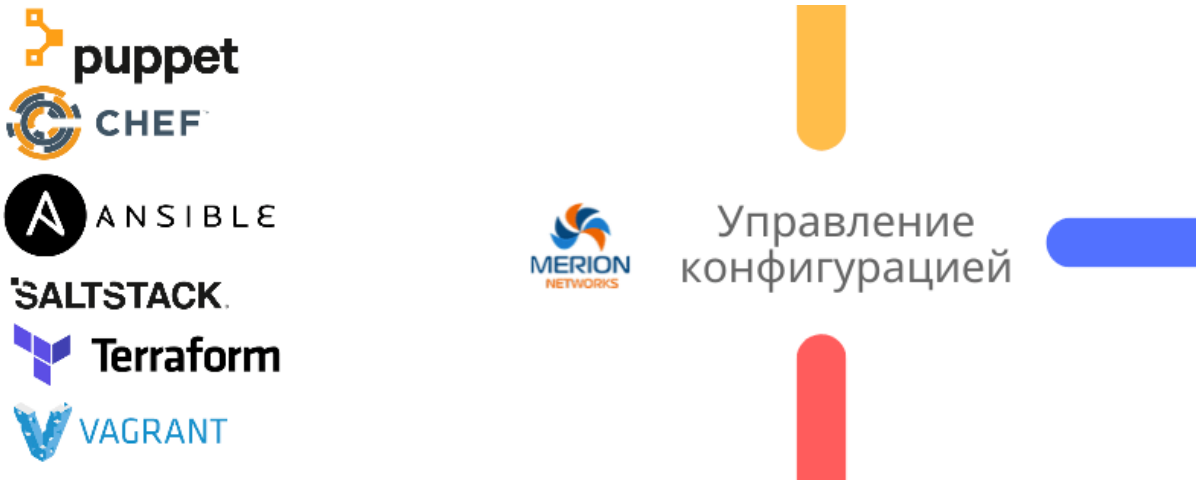
Gradle

Gradle - это система автоматизации сборки с открытым исходным кодом, основанная на концепциях Apache Maven и Apache Ant. Он поддерживает Groovy правильный язык программирования вместо XML-файла конфигурации. Он предлагает поддержку добавочных сборок, автоматически определяя, какие части сборки обновлены.

Управление конфигурацией

Управление конфигурациями – это метод систематического учёта и обработки изменений в продукте для поддержки целостности системы. Этот термин пришёл в программирование из других областей, теперь он широко используется для обозначения управления конфигурацией сервера.

В конфигурационном управлении очень важную роль играет автоматизация и связанные с ней инструменты. Этот механизм позволяет серверу достичь необходимого заранее определённого состояния (использовать конкретный язык, инструмент или функции). Автоматизация, пожалуй, является важнейшим аспектом управления конфигурациями сервера.



Puppet

Puppet — это инструмент управления конфигурацией с открытым исходным кодом, используемый для настройки, развертывания и управления многочисленными серверами. Этот инструмент поддерживает концепцию инфраструктуры как кода и написан на Ruby DSL. Он также поддерживает динамическое масштабирование машин по мере необходимости.

Chef

Che — это инструмент управления конфигурацией с открытым исходным кодом, разработанный Opscode с использованием Ruby для управления инфраструктурой на виртуальных или физических машинах. Он также помогает в управлении сложной инфраструктурой на виртуальных, физических и облачных машинах.

Ansible

Ansible — это инструмент для управления ИТ-конфигурацией с открытым исходным кодом, обеспечения программного обеспечения, оркестровки и развертывания приложений. Это простой, но мощный инструмент для автоматизации простых и сложных многоуровневых ИТ-приложений.

Saltstack

SaltStack — это программное обеспечение с открытым исходным кодом, написанное на python и использующее push-модель для выполнения команд по протоколу SSH. Он предлагает поддержку как горизонтального, так и вертикального масштабирования. Он поддерживает шаблоны YAML для записи любых скриптов.

Terraform

Terraform — это инструмент с открытым исходным кодом для безопасного, эффективного построения, изменения, развертывания и управления версиями инфраструктуры. Он используется для управления существующими и популярными поставщиками услуг, а также для создания собственных решений. Это помогает в определении инфраструктуры в конфиге или коде и позволит пользователю легко перестраивать или изменять и отслеживать изменения в инфраструктуре.

Vagrant

Vagrant является одним из популярных инструментов для создания и управления виртуальными машинами (VM). Он имеет простой в использовании и настраиваемый рабочий процесс, ориентированный на автоматизацию. Это помогает сократить время настройки среды разработки, увеличивает производственный паритет.

Непрерывная интеграция



Непрерывная интеграция — это методика загрузки кода в общий репозиторий несколько раз в день с последующей проверкой. Таким образом вы автоматически выявляете проблемы на ранних этапах, когда их проще всего устранить, и развертываете новые возможности для пользователей настолько быстро, насколько это возможно.

Jenkins

Jenkins является одним из самых популярных инструментов DevOps с открытым исходным кодом для поддержки непрерывной интеграции и доставки через DevOps. Это позволяет осуществлять непрерывную интеграцию и непрерывную доставку проектов независимо от того, над чем работают пользователи платформы, с помощью различных конвейеров сборки и развертывания. Jenkins можно интегрировать с несколькими инструментами тестирования и развертывания.

Travis ci

Travis CI — это облачная распределенная платформа непрерывной интеграции, используемая для создания и тестирования проектов, размещенных на GitHub и Bitbucket. Это настраивается путем добавления файла YAML. Его можно протестировать бесплатно для проектов с открытым исходным кодом и на платной основе для частного проекта.

Bamboo

Bamboo является одним из популярных продуктов, разработанных Atlassian для поддержки непрерывной непрерывной интеграции. Его большая часть функциональности предварительно встроена, что означает, что нам не нужно загружать различные плагины, такие как Jenkins. Он также поддерживает плавную интеграцию с другими продуктами Atlassian, такими как JIRA и Bitbucket.

Hudson

Hudson — это свободное программное обеспечение, написанное на JAVA и работающее в контейнере сервлетов, например, GlassFish и Apache Tomcat. Он обеспечивает возможность запуска вашего пакета автоматизации с любыми изменениями в соответствующей системе управления исходными кодами, такими как GIT, SVN и так далее. Он также обеспечивает поддержку всех базовых проектов maven и Java.

Teamcity

TeamCity — это непрерывная интеграция на основе сервера, которая создает инструмент управления, разработанный JetBrains. Он имеет простой и удобный пользовательский интерфейс (UI) и обеспечивает прогресс сборки, детализирует информацию о сборке и хронологическую информацию для всех конфигураций и проектов.

Circleci

CircleCI доступен как в виде облачных, так и локальных решений для непрерывной интеграции. Запускать и поддерживать легковесные и легкочитаемые конфигурации YAML просто и быстро.

Непрерывная безопасность

Непрерывное тестирование безопасности — это практика проверки, измерения и оптимизации эффективности мер безопасности на постоянной основе, с использованием инструментов автоматического тестирования для непрерывного выявления возникающих слабых мест и их оперативного исправления.



Snyk

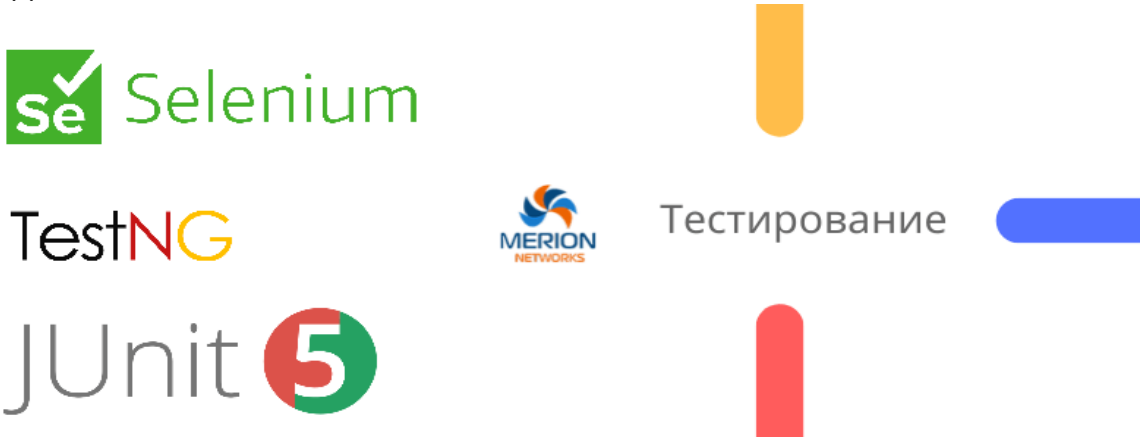
Интегрируйте Snyk в жизненный цикл разработки, чтобы автоматически находить и исправлять уязвимости безопасности с открытым исходным кодом. Он поддерживает JS, .Net, PHP, NPM, jQuery, Python, Java и так далее, и может быть интегрирован в кодирование, управление кодом, CI/CI, контейнер и развертывание. Snyk имеет самую большую базу уязвимостей с открытым исходным кодом

Netsparker

Netsparker автоматически сканирует ваше приложение на наличие уязвимостей и предоставляет действенные секретные отчеты, чтобы вы могли действовать в соответствии с приоритетом. Сценарий безопасности DevOps заключается в проверке нового коммита и сообщении об ошибке непосредственно в систему отслеживания, например, Jira или GitHub, и повторном сканировании после исправления разработчиком. Вы видите, что это интегрируется на каждом этапе SDLC (Software Development Life Cycle - жизненный цикл разработки).

Тестирование

Инструменты тестирования охватывают множество потребностей и возможностей, в том числе глубокое тестирование, управление тестированием и оркестрацию. Однако важнейшая функция пакета инструментов DevOps — автоматизация. В долгосрочной перспективе автоматическое тестирование окупается за счет ускорения циклов разработки и тестирования. В среде DevOps оно важно еще и потому, что повышает осведомленность.



Selenium

Selenium - самый популярный инструмент для тестирования с открытым исходным кодом. Он поддерживает автоматизацию тестирования в различных браузерах и операционных системах. Его легко интегрировать с инструментами управления тестированием, такими как ALM, JIRA, а также с другими инструментами DevOps, такими как Jenkins, Teamcity, Bamboo и так далее.

TestNG

TestNG — это среда тестирования с открытым исходным кодом, разработанная и вдохновленная Junit и Nunit. Ее можно легко интегрировать с веб-драйвером selenium для настройки и запуска сценариев автоматизации тестирования. Она также генерирует различные отчеты о тестировании, такие как HTML или XSLT.

Junit

JUnit — это инфраструктура модульного тестирования с открытым исходным кодом, используемая разработчиками для написания и запуска повторяющихся тестов. JUnit поддерживает различные аннотации тестов, с помощью которых любой разработчик может написать цельный тестовый блок. Его легко интегрировать с другими инструментами DevOps, такими как Jenkins, GIT и так далее.

Мониторинг

Мониторинг — это постоянный сбор и анализ различных параметров (метрик) поведения системы. С его помощью можно описать и измерить в числовом выражении каждый важный аспект проекта. Данные из разных точек среды собираются системой мониторинга, которая отвечает за хранение, агрегацию, визуализацию данных и автоматические реагирует на изменения, когда значения соответствует заданным условиям.



Nagios

Nagios — это открытый исходный код и один из самых популярных инструментов для непрерывного мониторинга. Nagios помогает отслеживать системы, приложения, услуги и бизнес-процессы в культуре DevOps. Он предупреждает пользователей, когда что-то не так с инфраструктурой, и предупреждает их, когда проблема решена.

Подробно про Nagios

Grafana

Grafana — это аналитическая платформа с открытым исходным кодом, позволяющая отслеживать все показатели инфраструктуры, приложений и аппаратных устройств. Вы можете визуализировать данные, создавать и совместно использовать панель инструментов, настраивать оповещения. Вы можете получать данные из более чем 30 источников, включая Prometheus, InfluxDB, Elasticsearch, AWS CloudWatch и так далее.

Sensu

Sensu — это инструмент для мониторинга с открытым исходным кодом, написанный на Ruby, который помогает просто и эффективно контролировать серверы, сервисы, приложения, облачную инфраструктуру. Его легко масштабировать, чтобы вы могли легко отслеживать тысячи серверов.

New relic

New Relic — это программный аналитический продукт для мониторинга производительности приложений (APM - Application Performance Monitoring), который предоставляет в реальном времени трендовые данные о производительности веб-приложений и уровне удовлетворенности, которое испытывают конечные пользователи. Он поддерживает сквозную трассировку транзакций и отображает их с помощью различных цветовых диаграмм, графиков и отчетов.

Datadog

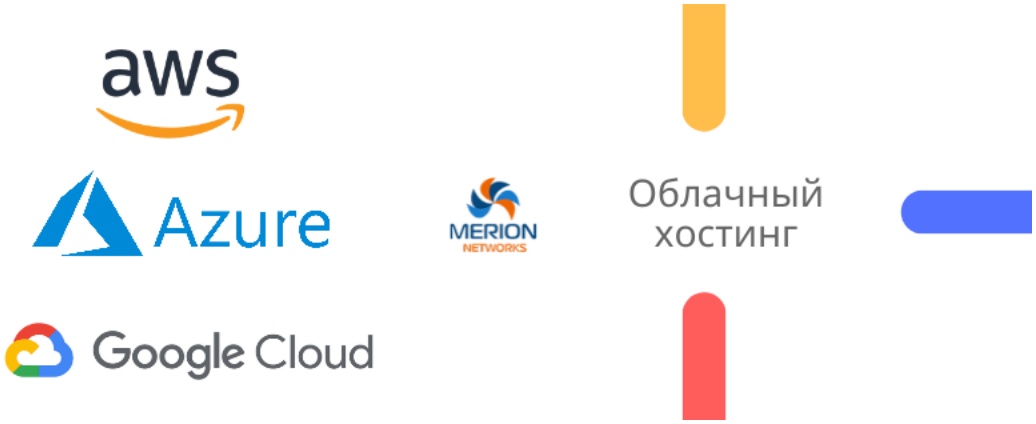
Datadog — это инструментальный инструмент метрики сервера. Он поддерживает интеграцию с различными веб-серверами, приложениями и облачными серверами. Его сервисная панель предоставляет различные графики мониторинга в реальном времени по всей инфраструктуре.

ELK

ELK — это коллекция из трех продуктов с открытым исходным кодом - Elasticsearch, Logstash и Kibana, которые разрабатываются, управляются и поддерживаются компанией Elastic. Он позволяет пользователям получать данные из любого источника в любом формате, а затем искать, анализировать и визуализировать эти данные в режиме реального времени.

Облачный хостинг

Облачный хостинг обеспечивает доступ к приложениям и веб-сайтам через облачные ресурсы. В отличие от обычного хостинга решения не развертываются на одном сервере. Вместо этого приложение или сайт размещается в сети связанных виртуальных и физических облачных серверов, что гарантирует гибкость и масштабируемость



AWS

AWS — это веб-хостинговая платформа, созданная Amazon, которая предлагает гибкие, надежные, масштабируемые, простые в использовании, масштабируемые и экономически эффективные решения. Используя эту облачную платформу, нам не нужно беспокоиться о настройке ИТ-инфраструктуры, которая обычно занимает достаточно много времени.

Azure

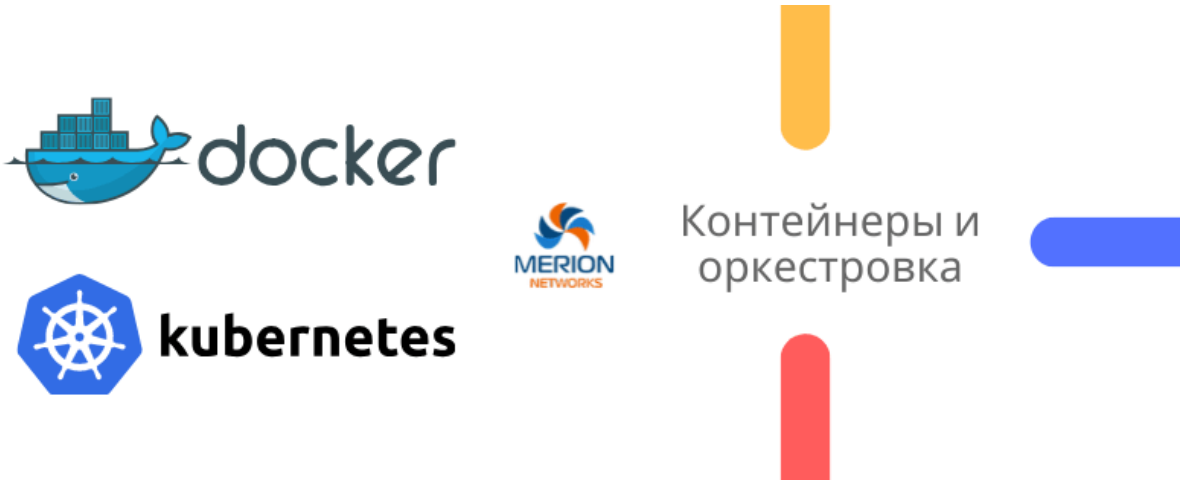
Azure — это платформа облачных вычислений, разработанная Microsoft для создания, развертывания, тестирования и управления приложениями и службами через глобальную сеть своих центров обработки данных. Службы, предоставляемые Microsoft Azure, представлены в форме PaaS (платформа как услуга) и IaaS (инфраструктура как услуга).

GCP

Google Cloud — это полный набор услуг облачного хостинга и вычислений, предлагаемых Google. Он поддерживает широкий спектр услуг для вычислений, хранения и разработки приложений, которые используют Google Hardware.

Контейнеры и оркестровка

Контейнерная оркестровка относится к инструментам и платформам, которые используют для автоматизации, управления и планирования приложений, определённых отдельными контейнерами. Инструменты оркестрации контейнеров, как Kubernetes, помогают запускать и управлять всеми вашими контейнерами в производственной среде и решать проблемы, которых могло не быть на стадии разработки на одной машине”.



Docker

Docker — это инструмент для создания, развертывания и запуска приложений с использованием контейнеров. Этот контейнер позволяет разработчику упаковывать приложение со всеми необходимыми ему

компонентами и подкомпонентами, такими как библиотеки и другие зависимости, и отправлять все это в виде одного пакета.

Kubernetes

Kubernetes — это система контейнерной оркестрации с открытым исходным кодом, изначально разработанная Google, и в настоящее время она поддерживается Cloud Native Computing Foundation. Он используется для автоматизации развертывания, масштабирования и управления приложениями. Он работает с другими инструментами-контейнерами, включая Docker.

3. Цели методологии DevOps

Раскладываем методологию DevOps по полочкам. Анализируем преимущества и недостатки, вспоминаем основополагающие принципы, идеи, культуру и историю развития DevOps.

Что было до DevOps

DevOps (англ. development and operations) – набор практик, который направлен на сокращение жизненного цикла разработки систем и обеспечение непрерывной поставки программного обеспечения высокого качества. Взаимодействие разработчиков и специалистов по информационно-технологическому обслуживанию, а также взаимная интеграция их рабочих процессов, позволяет создавать качественные продукты в короткие сроки.

Разработка (development) и эксплуатация (operations) продолжительное время были изолированными модулями. Код писали программисты, а системные администраторы отвечали за его развертывание и интеграцию. В рамках одного проекта специалисты работали отдельно, поскольку связь между двумя разрозненными хранилищами была ограничена.

Этот метод работал с 1970 года, пока доминировала каскадная модель процесса разработки программного обеспечения, известная как Waterfall. Методика предполагала последовательный переход между этапами без пропусков и возвращений на предыдущие стадии.

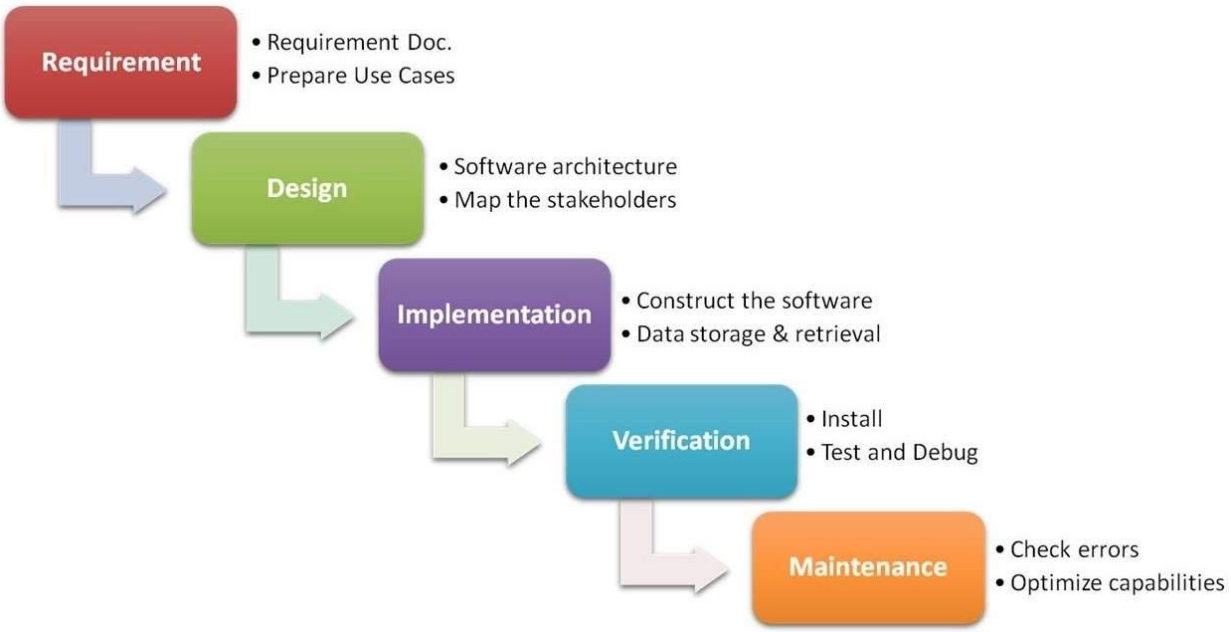


Схема методологии Waterfall

Со временем Waterfall была раскритикована за недостаточную гибкость, а ее основная цель – формальное управление проектом, приносила ущерб срокам, стоимости и качеству.

Команды нуждались в гибкости во время разработки программного обеспечения. Необходимость реализации проекта в форме коротких "спринтов" и выпуска более частых (от двух недель, до двух дней) релизов потребовали нового подхода.

В 2001 году на смену Waterfall пришла гибкая методология разработки или Agile. Она включает ряд подходов и практик, основанных на четырех ценностях и 12 принципах «Манифеста гибкой разработки программного обеспечения». Сюда также относят SCRUM, Kanban, Lean, Feature-driven development (FDD) и другие сходные подходы.

Agile применяется к организации работы небольших групп, которые создают продукт короткими итерациями (от двух до четырех недель). Каждая итерация выглядит как программный проект, который включает все типовые задачи: планирование, анализ требований, проектирование, программирование, тестирование, документирование. В конце итерации заказчик получает рабочий продукт.

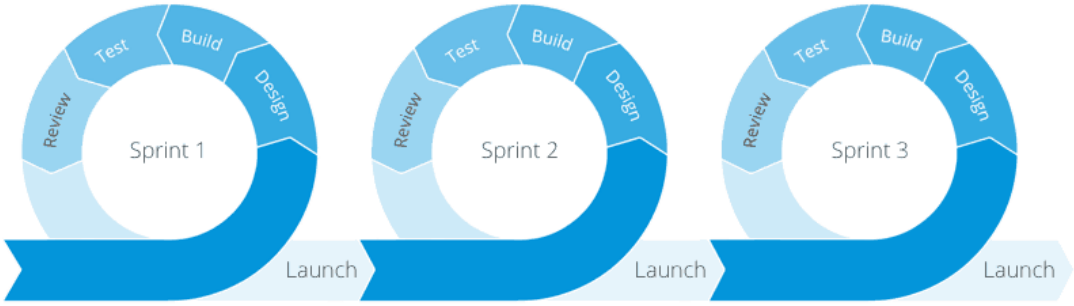


Схема подхода Agile

Методология Agile критиковали за отсутствие управления требованиями. Заказчик может выставить новые требования в конце каждой итерации, что противоречит архитектуре уже созданного продукта. Частые изменения и усовершенствования продукта могут привести к массовому рефакторингу и плавающей стоимости проекта в итоге.

Идея и культура DevOps

Методология DevOps получила широкое распространение после организованной бельгийским разработчиком Патриком Дебуа в 2009 году конференции DevOpsDays.

Главная идея DevOps заключается в том, чтобы устранить перекладывание ответственности на других членов команды в больших коллективах. Взаимозависимость между созданием и эксплуатацией программного обеспечения преследовала цель привить команде новую культуру разработки продукта.

Культура DevOps предполагает, что каждый из членов команды ответственен за конечный результат. Базируется она на нескольких основных положениях:

- **Регулярное сотрудничество и общение.** Команда должна работать слаженно, понимать потребности и ожидания всех ее членов.
- **Постепенное развертывание.** Внедрение постепенного развертывания позволяет группам доставки выпускать продукт, имея возможность вносить обновления и делать откат, если что-то пойдет не так.
- **Общая ответственность.** Все члены команды должны двигаться к единой цели и отвечать за проект в равной степени.

- **Решение проблем на ранних этапах.** Методология DevOps требует, чтобы в жизненном цикле проекта задачи выполнялись как можно быстрее. Это помогает оперативно решать возникшие проблемы.

Принципы DevOps

В 2010 году Дэймоном Эдвардсом и Джоном Уиллисом была разработана модель CAMS, ключевые идеи которой стали принципами DevOps. Согласно ей, развитие DevOps идет в трех направлениях: люди, процессы и инструменты. При этом важна поддержка каждого пункта на всех этапах развития.

Аббревиатура CAMS расшифровывается следующим образом:

- культура (culture);
- автоматизация (automation);
- измерение (measurement);
- обмен (sharing).

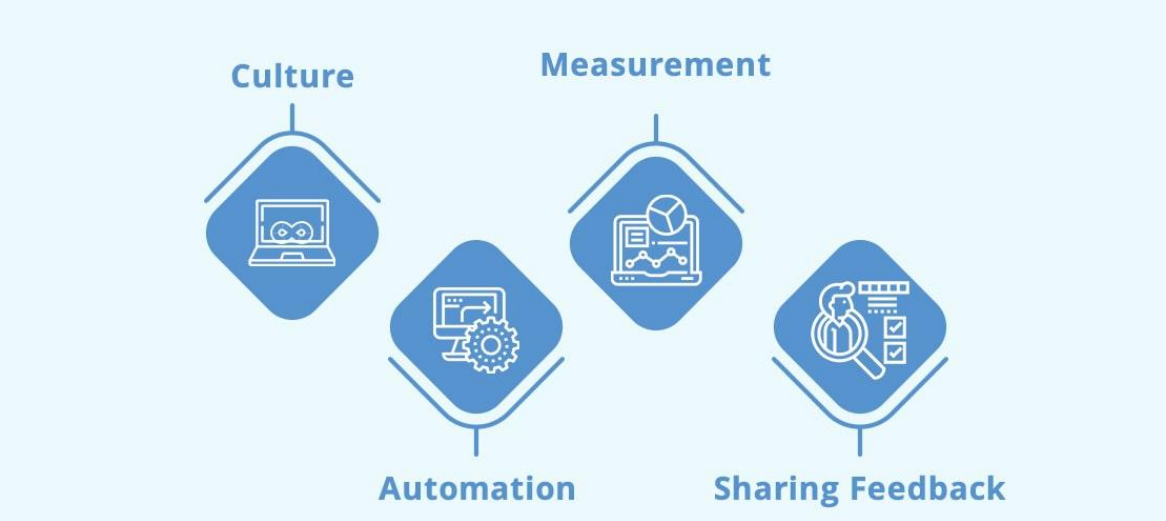


Схема модели CAMS

Культура

Классические бизнес-модели в IT разделяют специалистов по разработке и эксплуатации на две отдельные группы. До появления DevOps они общались на разных языках, ведь перед разработчиками стояла задача быстро внедрять инновации, а операционный персонал отвечал за поддержание стабильной среды и инфраструктуры.

Конкурирующие рабочие цели создавали между специалистами по разработке и эксплуатации недопонимание, поэтому основная задача DevOps – изменить бизнес-культуру, разделить ответственность двух групп и объединить их профессиональные навыки.

Автоматизация

Следуя пути DevOps, код требуется переводить из стадии разработки в производство непрерывно в автоматическом режиме, поэтому автоматизацию можно считать синонимом DevOps.

В идеале автоматизировать нужно почти все:

- инфраструктуру;
- выпуски программного обеспечения (software releases);
- тестирование;
- развертывание;
- основные задачи по безопасности;
- политику соглашений;
- задачи управления конфигурацией.

Автоматизация упрощает рабочие процессы, сокращает количество сбоев и откатов, уменьшает количество ошибок, которые возникают при ручной настройке. Повышение эффективности, улучшение производительности и польза для конечного потребителя – главные преимущества автоматизации.

Измерение

Измерение необходимо для постоянного предложения ценности и улучшений. В DevOps важно отслеживать ключевые показатели, которые зависят от целей проекта.

Измерять нужно показатели следующих процессов:

- мониторинга и отслеживания производительности на протяжении всего жизненного цикла разработки программного обеспечения;
- сбора, анализа и предоставления способов реагирования на обратную связь;
- анализа ошибок и способов их предотвращения;
- оказания помощи командам в работе над общими целями.

DevOps способствует развитию только в том случае, если конкретные показатели собираются и анализируются непрерывно.

Обмен

DevOps подразумевает тесное сотрудничество специалистов по разработке и эксплуатации. Большое внимание уделяется прозрачности и открытости в коллективе. Чем больше знаний распространяется между сотрудниками, тем больше обратной связи они получают – это помогает улучшить их работу в целом.

DevOps и Agile

DevOps является естественным продолжением гибких подходов и подходов к непрерывной доставке. DevOps и Agile могут дополнять друг друга и применяться в тандеме, но сравнивать эти методологии не стоит.

По сути, DevOps объединяет две разрозненные команды (разработку и эксплуатацию), чтобы обеспечить быстрые выпуски программного обеспечения. Agile ориентирован на сотрудничество небольших команд друг с другом для быстрого реагирования на изменчивые потребности пользователей.

Основные различия между DevOps и Agile:

- Разработка, тестирование и развертывание программного обеспечения происходят как в DevOps, так и в Agile. Подход Agile характерен тем, что разработка завершается сразу после развертывания. DevOps же включает операции, которые происходят постоянно, например, мониторинг и модификации программного обеспечения;
- В Agile разные специалисты несут ответственность за разработку, тестирование и развертывание программного обеспечения. В DevOps за все эти процессы отвечают специально обученные инженеры;
- Agile выступает за поэтапное развертывание после каждого спринта. Для DevOps характерна непрерывная доставка (до нескольких раз в день).

Жизненный цикл DevOps

Выделим основные этапы проекта, следующего принципам DevOps:

1. Непрерывное развитие;
2. Непрерывная интеграция;
3. Непрерывное тестирование;
4. Непрерывное развертывание;
5. Непрерывный мониторинг;
6. Постоянная обратная связь.

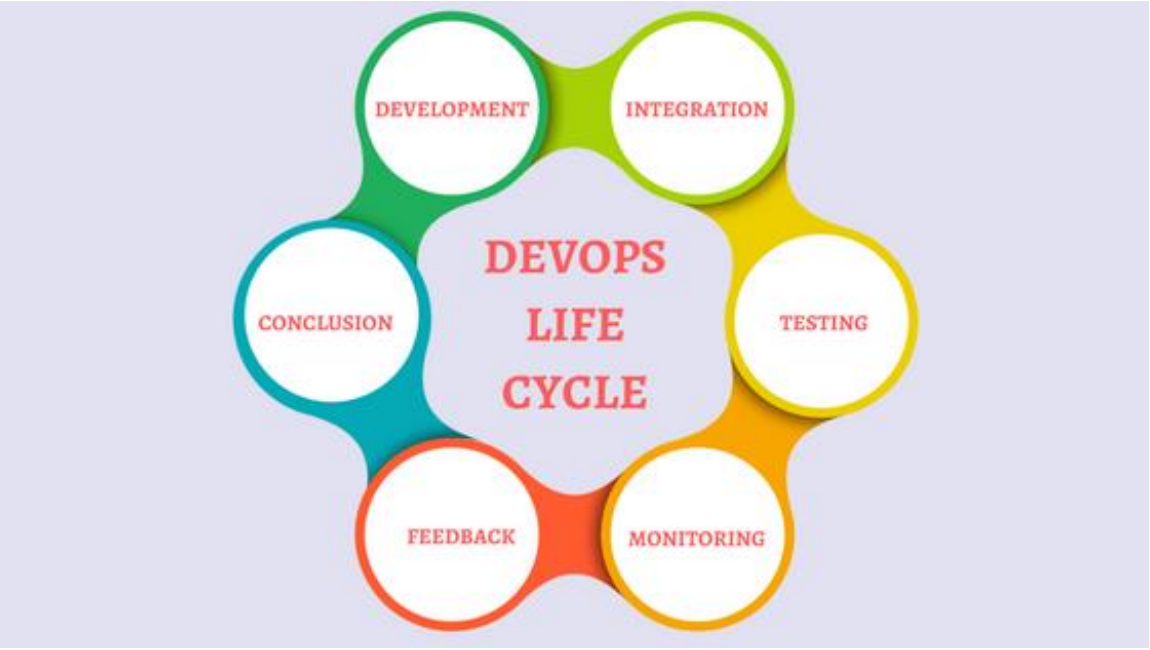


Схема жизненного цикла DevOps

Эти этапы гарантируют оптимизацию всех процессов разработки, от предложения до производства и поставки.

Непрерывное развитие

На первом этапе происходит планирование и кодирование программного обеспечения, а также формируется видение проекта.

Код может быть написан на любом языке, но поддерживается с помощью средств контроля версий. Процесс поддержки называется управлением исходным кодом (SCM). В нем используются следующие инструменты: Git, SVN, Mercurial, CVS и JIRA.

Непрерывная интеграция

На этом этапе чаще всего применяется Jenkins. Когда в репозитории Git происходят изменения, Jenkins извлекает обновленный код и готовит сборку. Упакованный код переходит к следующему этапу и пересылается либо на рабочий, либо на тестовый сервер.

Этап включает в себя не только компиляцию кода, но и проверку, модульное тестирование, интеграционное тестирование и упаковку. Непрерывная интеграция нового кода в существующий исходный код помогает отразить изменения для конечных пользователей.

Непрерывное тестирование

На этапе непрерывного тестирования разрабатываемое программное обеспечение проверяется на наличие ошибок. Автоматическое тестирование позволяет разработчикам экономить силы и время.

Для непрерывного тестирования используются следующие инструменты: Selenium, TestNG, JUnit и т.д. Они позволяют инженерам QA тестировать несколько баз кода параллельно, чтобы гарантировать отсутствие недостатков в функциональности. При этом тестируемое приложение часто запускается в виртуальной среде или контейнерах, например, с помощью Docker.

Автоматическое тестирование выполняет Selenium, а отчеты генерирует TestNG. Весь этап можно автоматизировать с помощью инструмента непрерывной интеграции Jenkins. В конце, протестированный код повторно отправляется на этап непрерывной интеграции для обновления исходного кода.

Непрерывное развертывание

Когда код развертывается на производственных серверах, важно получить корректный результат на всех серверах.

Управление конфигурацией – ключевой процесс на этом этапе. Он обеспечивает точное развертывание приложения и поддерживает согласованность конфигураций на всех серверах.

Ansible, Puppet и Chef – инструменты, которые чаще всего используются в DevOps для быстрого и непрерывного развертывания нового кода.

Не менее важную роль на данном этапе играют инструменты контейнеризации. Vagrant и Docker обеспечивают согласованность в различных средах – от разработки и тестирования, до подготовки и производства.

Непрерывный мониторинг

Важный этап жизненного цикла DevOps, на котором в реальном времени отслеживается производительность приложения. Для этого автоматически собираются определенные показатели телеметрии и метаданных, а также настраиваются оповещения об отклонениях в работе.

Непрерывный мониторинг помогает поддерживать доступность сервисов. Он также определяет угрозы и основные причины повторяющихся системных ошибок, а проблемы безопасности автоматически решаются в момент появления.

Активное участие в непрерывном мониторинге принимают операционные группы. Они наблюдают за действиями пользователей, проверяют системы на предмет необычного поведения и отслеживают наличие ошибок.

Для этого используются следующие популярные инструменты: Prometheus, Splunk, ELK Stack, Nagios и другие. Они

обеспечивают полный контроль над производительностью системы, рабочего сервера и приложения.

Постоянная обратная связь

Непрерывная обратная связь – особый этап, на котором анализируются улучшения, сделанные на этапах непрерывного тестирования и непрерывной интеграции.

Чтобы оценить результаты изменений в конечном продукте, необходимо получить обратную связь от клиентов. Процесс разработки приложения обновляется с учетом их отзывов – после этого разработчики начинают вносить изменения в продукт. Когда отзывы становятся положительными, открывается путь для выпуска новых версий, либо для поддержки приложения.

Преимущества и недостатки методологии

Преимущества DevOps:

- Повышение конкурентоспособности бизнеса за счет быстрой доставки приложения конечному потребителю.
- Стандартизация и автоматизация процессов в DevOps помогает команде сосредоточиться на основных задачах и не тратить время на тестирование и исправление ошибок.
- Быстрое тестирование и развертывание позволяет увеличить частоту выпуска приложений.
- Оперативное повышение качества продукции и уменьшение времени реагирования на возникшие ошибки.
- Невозможность перекладывания ответственности, поскольку каждый из членов команды отвечает за общий результат.

Недостатки DevOps:

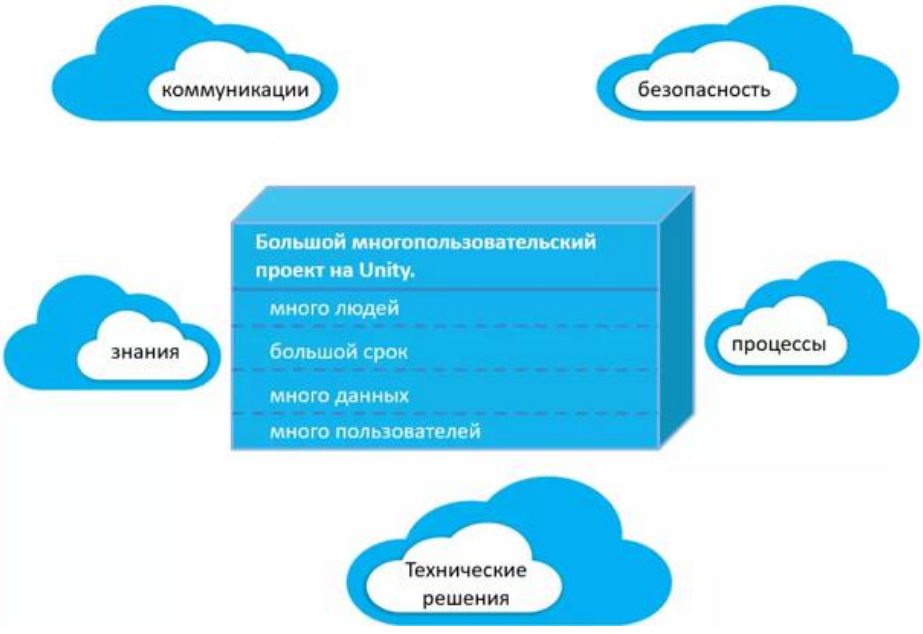
- Специалисты DevOps должны разбираться в разработке, тестировании, развертывании и поддержке, но эти знания могут быть поверхностными.
- DevOps – рабочая культура, которую нельзя реализовать, просто следуя жизненному циклу разработки программного обеспечения. Чтобы компания могла влиться в среду DevOps, ей необходимо постоянно соблюдать основные принципы методологии и стремиться к изменениям.
- Нанять высококвалифицированного инженера DevOps непросто. Эта специализация требует большого набора навыков, опыта и знания широкого стека технологий, поэтому требования достаточно высоки. Кроме того, DevOps – одни из самых высокооплачиваемых специалистов в IT.

4. Общее представление об инфраструктуре современной разработки

Как нужно вести Инфраструктуру разработки проекта в сфере информационных технологий

Термин инфраструктура означает ***«комплекс взаимосвязанных обслуживающих структур или объектов, составляющих и обеспечивающих основу функционирования системы»***. Не так давно это слово вошло в обиход, но проще говоря, инфраструктура – это фундамент, который позволяет создавать что-либо, в том числе игровые проекты.

Инфраструктура проекта существует даже тогда, когда ею никто специально не занимается. В качестве примера можно привести постройку какого-нибудь сарая без фундамента, со временем где-то появятся подпорки, пол утрамбуется, то есть образуется инфраструктура. Но это не значит, что инфраструктура проекта возникнет сама по себе и заниматься ею не нужно. **Необходимо придерживаться принципа: если известно, что появятся какие-то большие и сложные задачи, то лучше о них подумать заранее, что бы избежать проблем в дальнейшем**.



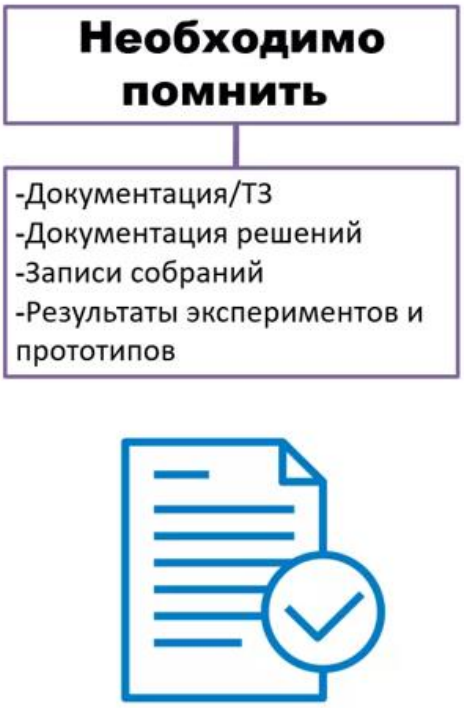
Возьмем большой многопользовательский проект, разрабатываемый в Unity. Можно предположить, что проект будет охватывать несколько областей, либо больших, либо сложных. В ходе реализации проекта планируется задействовать множество людей, сам проект будет долго разрабатываться. Будет использоваться много данных и в процессе производства проекта, и в дальнейшем процессе функционирования проекта.

Будет много пользователей, и нужно придумать какие-то решения, какие-то процессы для коммуникаций во время разработки. В процессе работы будут появляться множества знаний, нужно будет заниматься безопасностью, и придумывать массу технических решений.

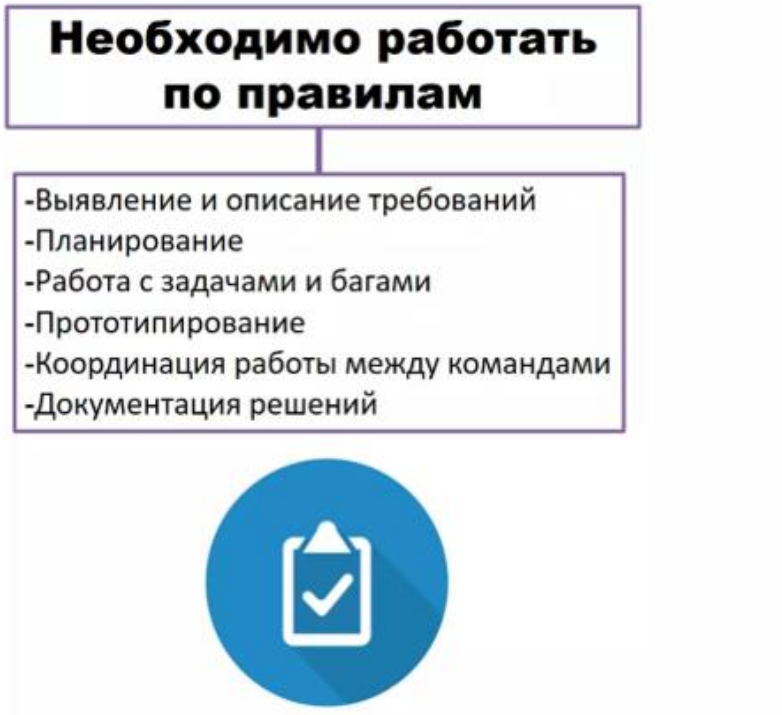


В первую очередь будет **необходимо общаться**. Кажется, что это легко, но на самом деле, кода коллектив превышает 10 человек, то это совсем не просто, а когда возникают разнопрофильные команды, а их будет не 2, не 4, а от 6 до 10, это еще сложнее. И это только команда проекта. Также существуют другие сотрудники компании, это менеджмент компании, с ними тоже нужно общаться, потому что они финансируют проект. Также есть люди из внешнего окружения, ими могут выступать партнеры, которые являются держателями платформ. Во время разработки необходимо будет общаться с ними. Как это все реализуется? Есть средства, для мгновенной коммуникации, которые на первый взгляд кажутся весьма удобными. Это, например, чаты Telegram: чат по командам, чат проекта, чат компании. Это удобно, но проблема в том, что мгновенные сообщения работают, если их прочитали сейчас, здесь, в течение часа или дня. Если человека не было, он болел, был в отпуске или пришел в проект через два дня после сообщения, вся информация в чатах будет для него потеряна. Поэтому достаточно надежным каналом коммуникаций выступает e-mail, но он тоже не решает всех проблем. Это – канал, который точно доставит информацию, которую могут прочитать все участники проекта. Но не все умеют общаться почтой, эту «культуру» нужно прививать. Канал коммуникации должен работать как на внутреннюю, так и на внешнюю среду проекта. Когда запускается проект, всем интересно, что происходит в нем, в итоге все спрашивают, что происходит внутри разработки. Для этого можно создать блог, в котором можно рассказывать о тех или иных интересных

решениях. С одной стороны, это снимает каждодневные вопросы: «Что у Вас?», «Как у Вас?». С другой стороны, это порождает новые, какие-нибудь конкретные советы или вопросы по уже опубликованному материалу. Также можно устраивать ежедневные demo, которые могут опробовать другие коллеги, это позволяет держать всех участников в курсе происходящего. Это полезно тем, что можно узнать «новое» мнение. Это – повод, чтобы задуматься, пересмотреть какие-нибудь подходы.



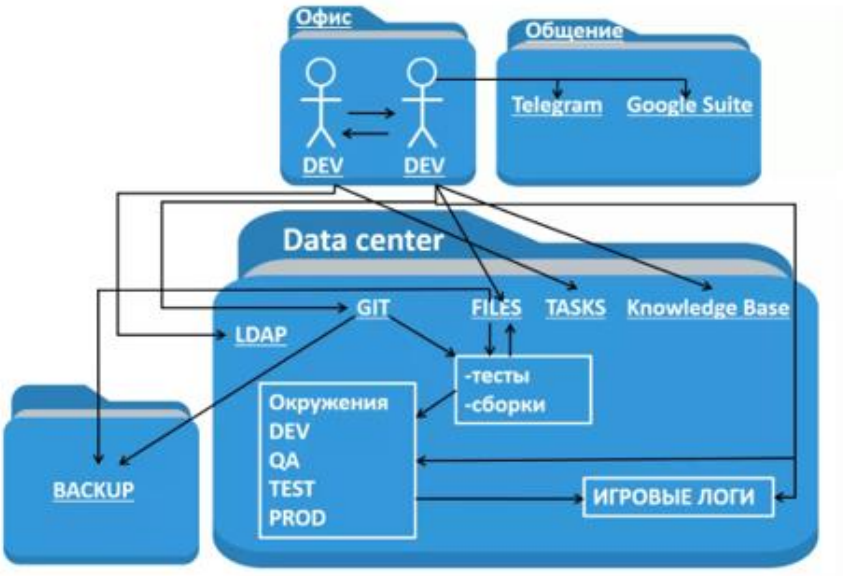
Когда разрабатывается большой проект, **нужно помнить массу вещей**. Производится большое количество документов, артефактов, решений, если это где-то не запомнить, то потом будет невозможно понять, для чего делается то или иное в проекте. В процессе разработки и реализации проекта появляется документация, но также важно помнить: «что мы решили?», «почему решили когда-то давно?». Если проект длится не один месяц и не один год, то это необходимо, потому что команда ротируется, команда растет, а если есть несколько человек, которые помнят все о проекте, не факт, что они будут работать над ним через полгода, через год. Важную информацию нужно фиксировать, и ход мыслей «почему что-либо было совершено» тоже лучше фиксировать. Это также касается записи собраний, экспериментов, чтобы что-то не проводить дважды, трижды. Для этого можно использовать GoogleSuite, таблицы, документы. Они удобны, в них могут одновременно работать несколько человек. Так же можно использовать сервис Confluence, он очень удобен для командной разработки.



Нужно работать по правилам. Одной из важных вещей инфраструктуры являются процессы, по которым работают. Нужно создать у команды общее понимание в тех или иных ситуациях, к кому обращаться, что делать, и в этом помогают процессы. Например, если нужно что-то нарисовать, нужно идти к художникам, или к какому-нибудь конкретному художнику, то есть к человеку, который исполняет определенную роль. Роли тоже нужно распределить. Необходимо определять задачи и работать над ними. Все это нужно делать, и это занимает время. Очень важно скоординировать работу между командами. Например, в начале разработки проекта вся команда находилась в одной комнате, потом команда выросла и стала находиться в двух, в трех комнатах, это сильно снижает уровень коммуникации. Если раньше можно было что сказать сидящему рядом коллеге, то сейчас нужно написать в чат, а его могут не прочитать. Получается, что некоторые несложные вопросы зависают и решения по ним не принимаются. Можно начать с простой инфраструктуры – поставить доску и вешать на нее заметки: что нужно сделать, когда, что важно, что неважно и т. д. Это хорошо работает, кода в команде от 4 до 5 человек. Когда людей в команде много, нужно их поделить на новые команды, в которых они будут заниматься определенным процессом. Эти процессы могут меняться.



Нужно управлять работой. Необходимо планировать работу на год, на полгода, потом это разбить на кварталы, на месяцы, на дни. Для мониторинга задач изначально можно использовать доску с заметками, но потом лучше использовать систему Trello или Jira.



Большой частью в разработке проекта являются **технические решения**. Например, если разработчиков много, то часть вопросов можно решить через внешние сервисы (Чаты, Google Suite), но не стоит полагаться только на них. Поэтому нужно предполагать несколько решений. Условно за центр можно взять серверы, которые предлагают много разных решений по обеспечению сохранности кода, по работе с задачами, по работе со знаниями. Также есть файловое хранилище, окружение для работы проекта, игровые «логи» (все данные, которые будут производиться проектом, их нужно хранить, с точки

зрения аудита, для разбора каких-то ситуаций у пользователя, а также для аналитики). Необходимо иметь «бэкап», чтобы в случае проблемы все данные можно было восстановить.



Нужно хранить большой объём данных. Требуется хранить проект, потому что там проявляется достаточно много бинарных данных, это могут быть гигабайты, десятки гигабайтов данных. Нужно хранить оригинальные «ассеты», они могут достигать десятки терабайтов, это то, что создают в основном художники, также и аудиоинженеры. Это сырые исходные данные, то, из чего потом собирается проект, импортируются модели, текстуры, все что угодно. В качестве инфраструктуры выступает код. Артефактами являются «логи». В качестве хранилища можно использовать Google Drive.



Так как ожидается большое количество пользователей, то важным инфраструктурным вопросом является наличие **большого количества «железных машин»**, которые должны обеспечить сессии пользователям. Хочется, чтобы весь этот «парк машин» работал предсказуемо, чтобы была возможность дополнять новыми машинами, или изымать другие машины. Будут сотни хостов, все это нужно обновлять, ставить «софт», как свой, так и чужой. Так как задача выглядит сложной, то нужно её полностью автоматизировать, то есть всё запрограммировать. Для этого можно выбрать Ansible («Используется для автоматизации настройки и развертывания программного обеспечения.»).



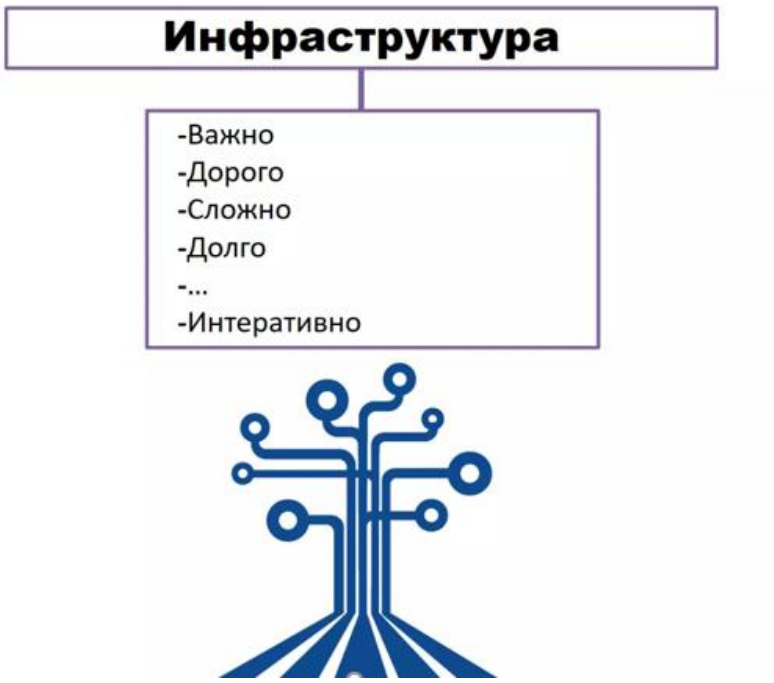
У всех участников проекта есть свои требования, которые могут противоречить друг другу. Часть людей, то есть разработчики хотят работать с нестабильным окружением. Им нужно очень быстро делать «фичи», запускать, смотреть, проверять их функционирование. С другой стороны, тестировщикам нужно стабильное окружение, чтобы провести весь цикл тестов. И, естественно, нужно стабильное окружение на «Продакшн». Поэтому нужно развернуть подобные инфраструктуры для всего окружения, и уметь разворачивать разные версии приложений для него.



Важным вопросом является **анализ рисков**. Его нужно проводить раз в год, полгода, квартал. Могут случаться разные непредвиденные, в том числе форс-мажорные ситуации. Нужно подумать, что требуется сделать, чтобы этого не произошло, либо как можно больше снизить последствия всех этих рисков. Учёт возможных рисков помогает в планировании работы.



Также следует помнить о **безопасности**. Фактически, это – следствие рисков. Всё это не просто, так как могут быть физические угрозы безопасности (например, воровство компьютера), так и угрозы цифровой безопасности (например, вирусы). Дело в том, что проекты, как правило, стоят дорого, инвестор тратит деньги, и он не хочет, чтобы внезапно код сервера клиента оказался еще у кого-нибудь, и еще кто-то вместо того, кто платит деньги, стал зарабатывать. Необходимо защищать свои инвестиции.



Инфраструктура – это важно, но дорого, если ею серьезно заниматься, то это еще и сложно, и требует затрат времени. Из положительных черт можно

выделить то, что в отличие от строителей домов, у разработчиков проекта есть возможность делать инфраструктуру интерактивно. Можно попробовать одно решение, или заменить его другим, может быть, более эффективным. Есть возможность делать фундамент даже во время «постройки» проекта.

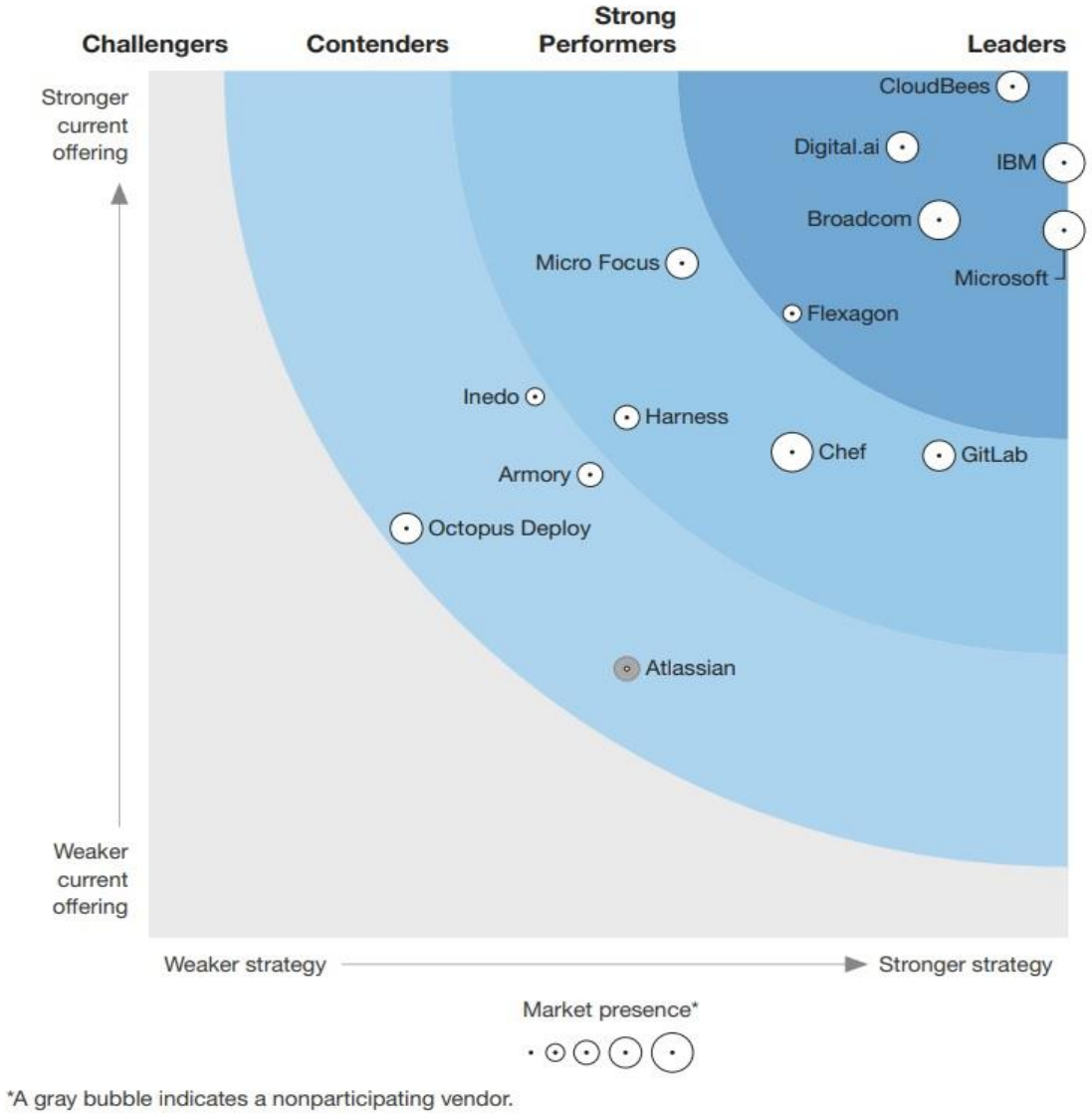
Инфраструктура проекта зависит от инфраструктуры всей компании. Эффективные решения, уже используемые в компании, можно применять для управления проектом. Интересные решения, которые появляются в процессе работы над проектом, можно внедрить в деятельность компании для повышения её результативности.

Непрерывная разработка и интеграция (CI/CD) дает крупным компаниям возможность значительно ускорить процесс разработки функционала и установки обновлений. Однако для оптимизации этих процессов нужно выбрать правильные инструменты среди большого многообразия, представленного на рынке. Для выбора подходящего инструмента разработчики могут ориентироваться на публикации на специализированных форумах, собственный опыт, опыт коллег, материалы вендоров. А можно воспользоваться экспертными знаниями независимых агентств и сделать предварительный выбор, основываясь на аналитических отчетах.

5. Российские и иностранные разработки в области DevOps

Названы иностранные лидеры рынка DevOps-решений

В июне 2020 независимое аналитическое агентство Forrester выпустило отчет «The Forrester Wave: Continuous Delivery and Release Automation», в котором представило обзор 14 наиболее крупных игроков рынка решений для обеспечения непрерывной интеграции и автоматизации релизов: Armory, Atlassian, Broadcom, Chef, CloudBees, Digital.ai (бывшая XebiaLabs), Flexagon, Gitlab, Harness, IBM, Inedo, Micro Focus, Microsoft и Octopus Deploy.



*A gray bubble indicates a nonparticipating vendor.

Аналитики Forrester оценили эти компании и их решения по ряду критериев, сгруппированных по трем категориям: функционал решения, стратегия компании, куда вошла инновационная политика, маркетинг, поддержка и обучение, ценообразование, партнерская экосистема, и присутствие на рынке.

	Forrester's weighting	Armory	Atlassian*	Broadcom	Chef	CloudBees	Digital.ai	Flexagon
Current offering	50%	2.67	1.55	4.20	2.80	4.91	4.56	3.60
Modeling resources and releases	35%	2.60	1.30	4.10	2.70	5.00	5.00	3.60
Delivering the release	35%	2.80	1.50	4.30	2.20	5.00	4.00	3.50
CDRA platform and deployment	30%	2.60	1.90	4.20	3.60	4.70	4.70	3.70
Strategy	50%	2.40	2.60	4.30	3.50	4.70	4.10	3.50
CDRA vision and value proposition	25%	3.00	3.00	5.00	3.00	5.00	3.00	3.00
Product innovation	20%	3.00	1.00	5.00	3.00	5.00	5.00	3.00
Market approach and viability	20%	1.00	3.00	3.00	3.00	5.00	5.00	3.00
Consulting, training, and support	15%	3.00	3.00	3.00	5.00	3.00	5.00	5.00
Licensing/pricing models	10%	3.00	3.00	5.00	3.00	5.00	3.00	5.00
Partner ecosystem	10%	1.00	3.00	5.00	5.00	5.00	3.00	3.00
Market presence	0%	2.20	2.40	4.20	4.40	3.60	3.60	1.60
Customer installed base	40%	1.00	3.00	3.00	5.00	3.00	3.00	1.00
Average deal size	30%	5.00	3.00	5.00	3.00	5.00	5.00	3.00
Product revenue	30%	1.00	1.00	5.00	5.00	3.00	3.00	1.00

All scores are based on a scale of 0 (weak) to 5 (strong).
*Indicates a nonparticipating vendor

Методология составления отчетов The Forrester Wave предполагает распределение вендоров по 4 группам (лидеры, сильные игроки, соперники и претенденты), однако в 2020 году аналитики выделили только три группы.

- Лидеры (Leaders): Broadcom, CloudBees, Digital.ai, Flexagon, IBM, Microsoft.
- Сильные игроки (Strong Performers): Chef, GitLab, Harness, Micro Focus.
- Соперники (Contenders): Armory, Atlassian, Inedo, Octopus Deploy.

Эксперты отметили многофункциональность платформы CloudBees, которая известна благодаря поддержке open source проекта Jenkins. В IBM ICloud DevOps for Hybrid Deployment была подчеркнута хорошая поддержка различных платформ и упомянуты недостаточные возможности генерирования отчетности и неудобство использования. Высокую оценку получили облачные решения Microsoft Azure DevOps Pipelines и Azure DevOps Server. Automic Continuous Delivery от Broadcom названо подходящим для крупных компаний, которым нужна полнофункциональная платформа, поддерживающая унаследованные решения. Flexagon подходит для компаний, использующих Oracle. Digital.ai (бывшая XebiaLabs), находящаяся на июль 2020 года в процессе слияния с CollabNet, производителем VersionOn, отмечена как платформа с простым развертыванием и гибкостью,

предоставляющая хорошую аналитику о степени готовности релизов и подходящая для компаний с комплексной средой и унаследованными системами.

Среди сильных игроков можно выделить Chef, к сильным сторонам которого относится поддержка принципов «infrastructure-as-code» и «policy as code». Известный поставщик средств разработки Atlassian оказался в категории «Соперники» (Contenders) благодаря решению Bitbucket Pipelines. В этом решении отсутствуют многие возможности автоматизации релизов, однако оно может оказаться привлекательным для тех, кто уже инвестировал в продукты Atlassian.

Российские платформы

Платформа «Базис» - совместное предприятие «Облачная платформа», в которое входят 3 компании, представило единый спектр ИТ-решений.

Сегодня продуктовая линейка «Базис» включает:

- платформу виртуализации для построения облачной инфраструктуры;
- решение для организации инфраструктуры виртуальных рабочих столов;
- динамическую ИТ-инфраструктуру для автоматизации DevOps и CI/CD;
- программный комплекс по защите персональных данных и конфиденциальной информации в виртуальной инфраструктуре;
- аппаратный гипервизор.

«Базис» (СП «Облачная платформа») — разработчик программных продуктов для оказания облачных услуг и платформы динамической инфраструктуры. Компания образована в результате создания СП лидирующими компаниями ИТ-рынка — «Ростелеком», YADRO и «ГС-Инвест»/ «Рубитех». Результатом объединения стала уникальная интеграция экспертиз известных разработчиков и появление единого продуктового портфеля для рынка PaaS и IaaS.

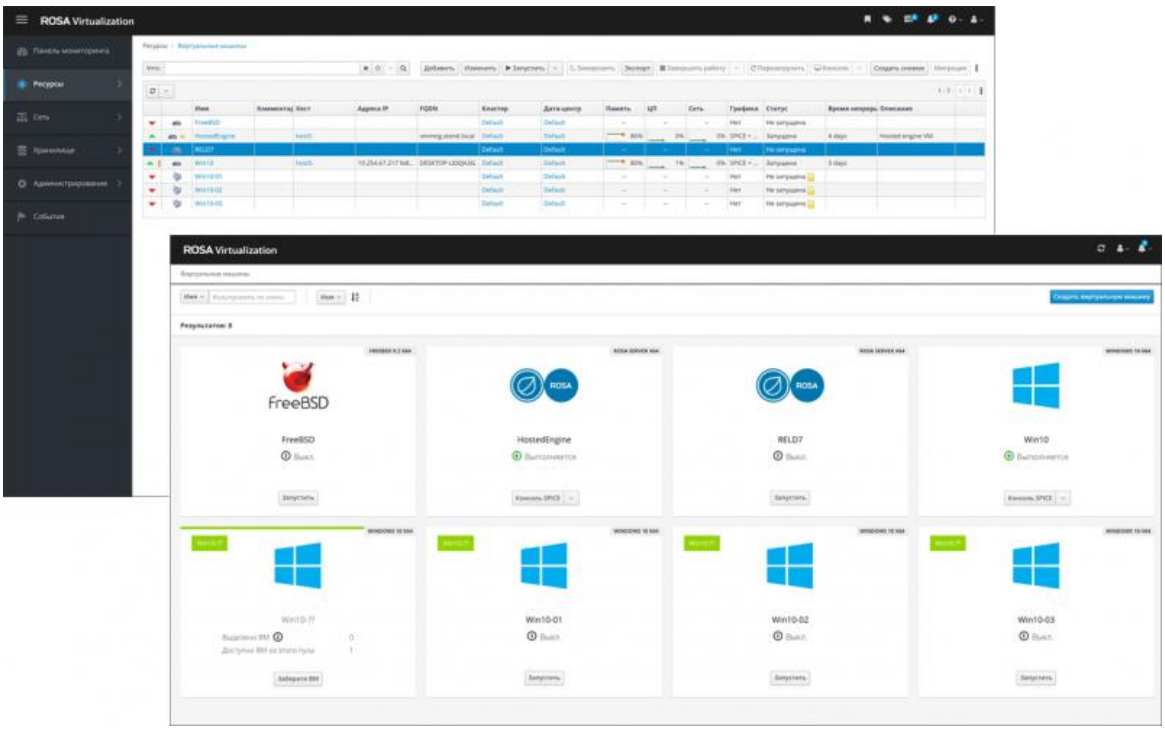
«Тионикс» — разработчик программных продуктов для оказания облачных услуг и платформы динамической инфраструктуры. Компания успешно развивается на российском рынке ИТ более восьми лет, является дочерним подразделением группы компаний «Ростелеком-ЦОД» — центром

компетенций ПАО «Ростелеком» по облачным технологиям. Все продукты компании включены в Единый реестр российских программ.

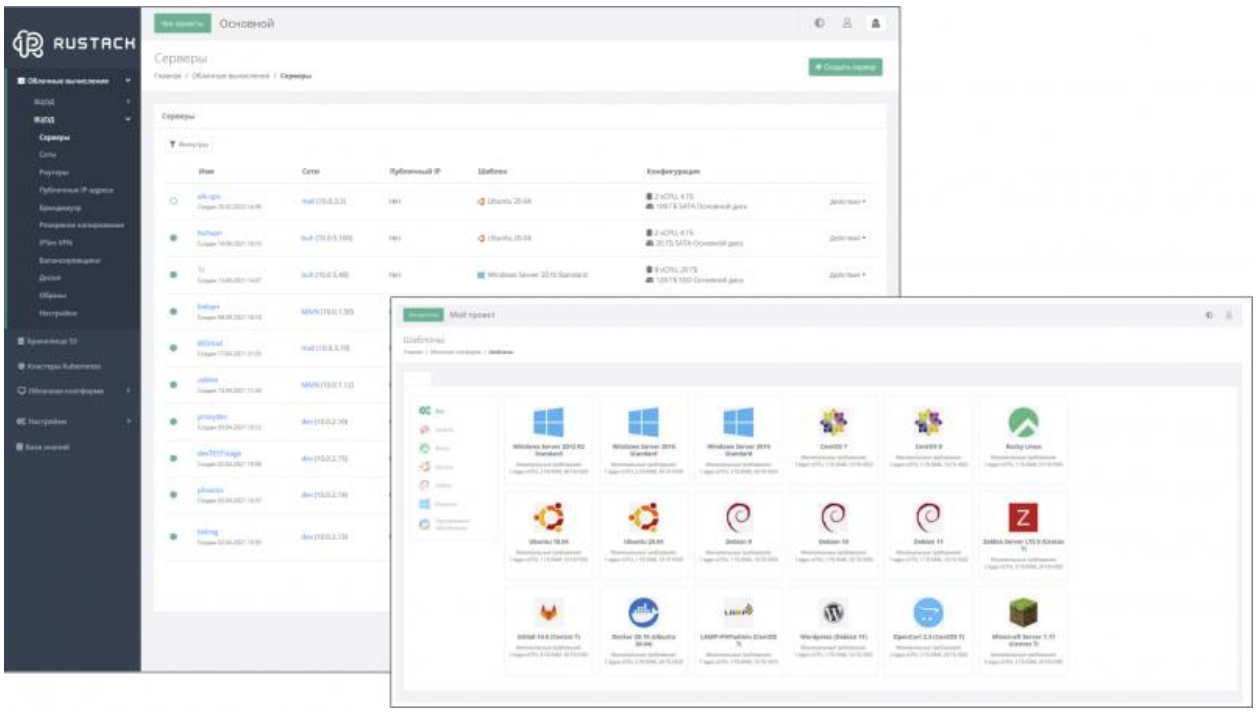
Digital Energy — российский разработчик программного обеспечения и поставщик облачных услуг, партнер и дочернее продуктовое подразделение компании YADRO, ведущей российской технологической компании разработчика и производитель вычислительных систем, платформ обработки и хранения данных.

«Скала Софтвер» — разработчик ПО виртуализации ИТ-инфраструктуры и рабочих станций пользователей, а также программно-определяемых хранилищ и мониторинга. До вхождения в СП «Облачная платформа», компания входила в группу ГС Инвест. ИТ-решения компании доказали свою эффективность и конкурентоспособность – более 7 лет компания развивает программное обеспечение, востребованное как государственными заказчиками, так и частными компаниями.

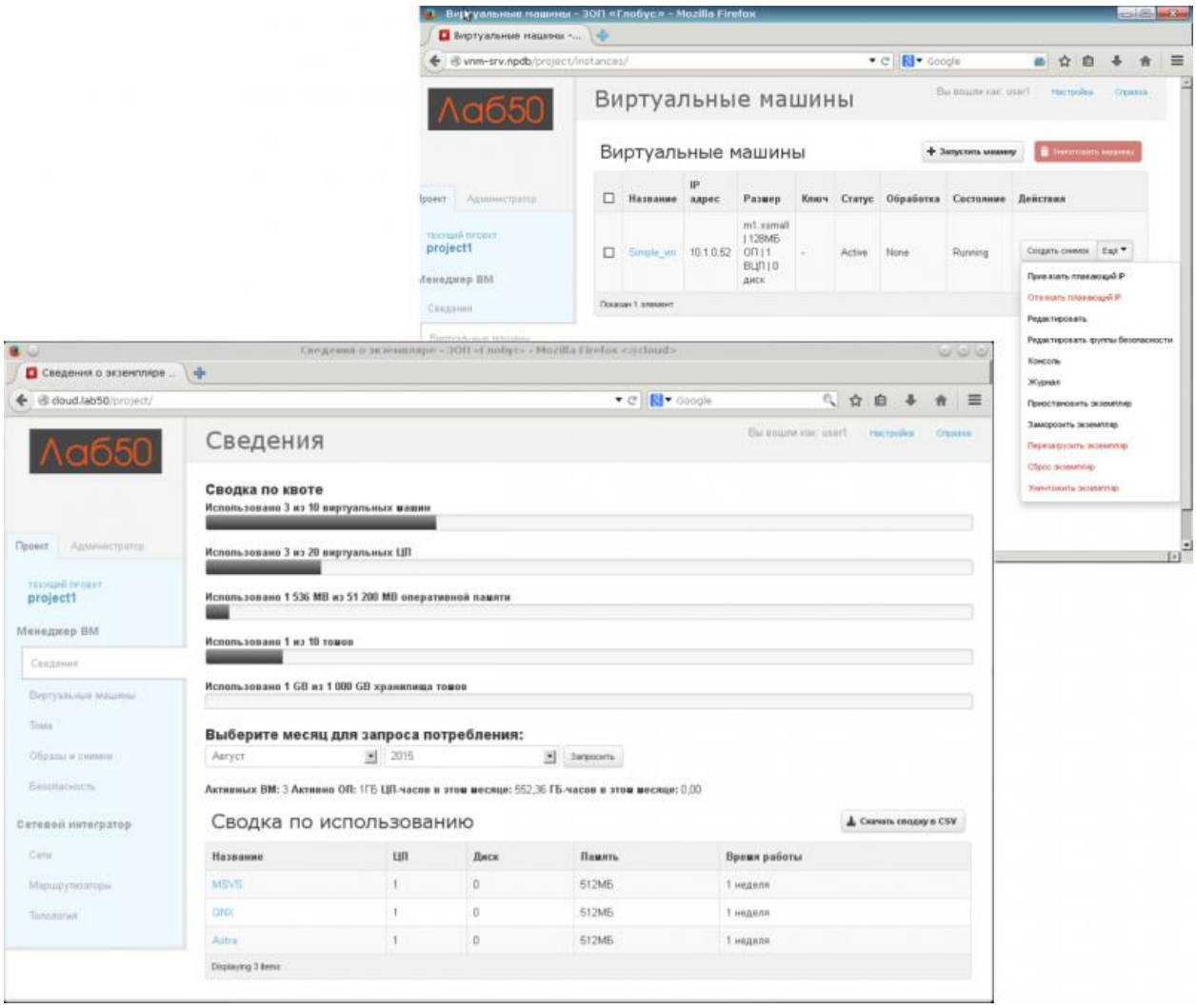
ROSA Virtualization (разработчик «НТЦ ИТ РОСА»). Платформа управления программными гипервизорами KVM и виртуальными машинами (ВМ) в облачной инфраструктуре. Решение поддерживает до тысячи ВМ и обеспечивает централизованное управление виртуализованной инфраструктурой с возможностью «живой миграции» (то есть без остановки работы) виртуальных машин между физическими серверами. В числе особенностей ROSA Virtualization значатся: возможность проброса периферийных USB- и PCI-устройств сервера, включая платы доверенной загрузки, в виртуальное окружение; средства контроля целостности ВМ; инструменты создания пулов ВМ и инфраструктуры виртуальных рабочих столов (Virtual Desktop Infrastructure, VDI); поддержка технологии NVIDIA GRID vGPU, позволяющей разделить физический графический процессор GPU на логические экземпляры и использовать их в виртуальной машине, обеспечивая таким образом полноценное графическое ускорение без привязки выделенной графической карты. Наличие сертификата ФСТЭК России допускает применение продукта в государственных информационных системах, в том числе работающих с персональными данными.



РУСТЭК (разработчик «РУСТЭК») — российская сервисная платформа виртуализации для создания и управления IT-инфраструктурой, входит в Единый реестр отечественного ПО. Система управления создана на базе OpenStack. В качестве гипервизора в платформе задействован KVM. Имеет специализированную ОС под задачи платформы, а также единый интерфейс для установки, обновления и масштабирования, API для интеграции с другими системами. Ролевая модель позволяет делегировать управление частями виртуальной инфраструктуры. Обеспечивает непрерывную работу всех сервисов. Предусмотрены механизмы как отказоустойчивости на уровне 1-го ЦОД, так и катастрофоустойчивость (возможность использования 3-ЦОДовой схемы). Имеется весь необходимый набор графических инструментов для самостоятельного развёртывания, конфигурирования и обслуживания платформы. Обновления платформы проходят в автоматизированном режиме. Обеспечивает централизованное управление виртуализованной инфраструктурой из личного кабинета пользователя, а также обладает удобной панелью управления виртуальными серверами и шаблонами операционных систем. На базе РУСТЭК возможно создать единое цифровое облачное пространство для компании любого уровня.



«Глобус» (разработчик «Лаборатория 50»). Платформа виртуализации на базе OpenStack и операционной системы Astra Linux Special Edition. Содержит дополнительные средства обеспечения безопасности и может быть использована для создания закрытых защищённых инфраструктурных cloud-сервисов и облачных хранилищ, соответствующих требованиям ФСТЭК России для систем класса АС, обрабатывающих данные с грифом «секретно» и «совершенно секретно». Программное обеспечение платформы ограничивает доступ к объектам облачной инфраструктуры (виртуальные машины, образы, и т. п.) в соответствии с мандатными и ролевыми правами пользователей.



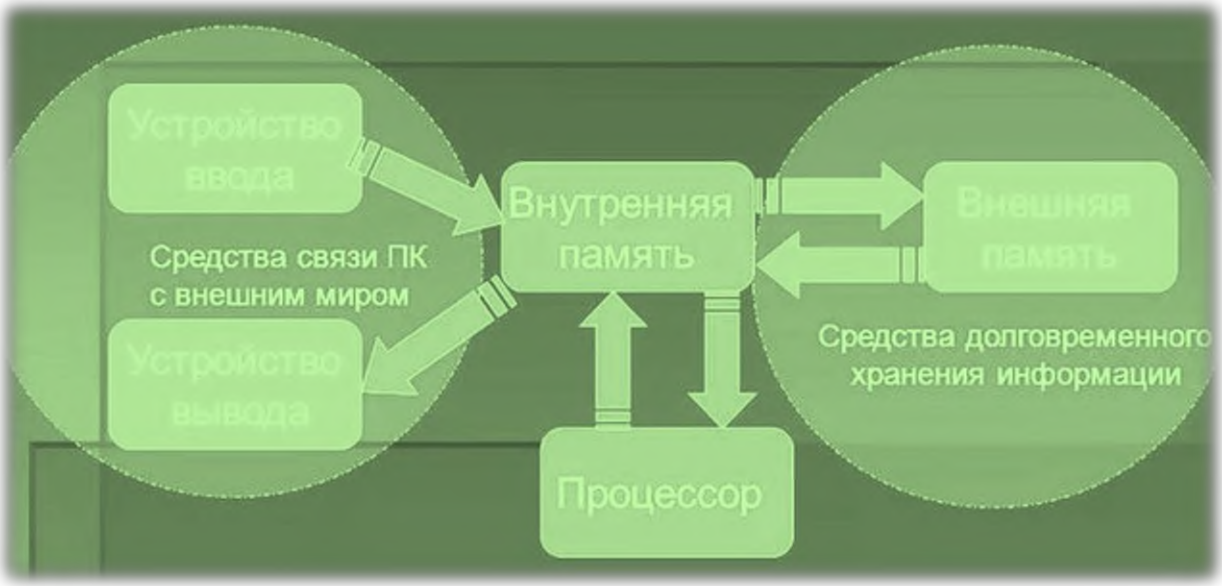
Альтернативные продукты на российском рынке: «Р-Виртуализация» (разработчик «Росплатформа»), «Брест» («РусБИТех-Астра»), «Аист» («Аэродиск»), «Ред Виртуализация» («Ред Софт»), ECP VeiL (НИИ «Масштаб»), Tionix Cloud Platform («Тионикс»).



МОДУЛЬ 2. ОСНОВЫ АДМИНИСТРИРОВАНИЯ OS

1. Принципы работы современных компьютеров: процессоры, память, накопители
2. Средства автоматизации и основные функции систем
3. Типы и назначение операционных систем.
4. Архитектура ОС на примере Linux
5. Процессы, управление процессами
6. Память, управление памятью
7. Шедулер
8. Дисковые системы
9. Файловые системы
10. Ядро операционной системы
11. Загрузка ОС
12. Типы дистрибутивов
13. Управление пакетами
14. Инициализация системы. Systemd, init-v
15. Управление пользователями
16. Кеширующий прокси-сервер (squid)
17. Производительность системы

1. Принципы работы современных компьютеров: процессоры, память, накопители



Суть построения компьютера похожа на строение человека. Центральный процессор, встроенная оперативная память и вместительный жесткий диск осуществляют функцию мозга. Материнская плата и встроенный чипсет олицетворяют кровеносную и нервную систему человека. Подключенная клавиатура, мышь и другие устройства ввода похожи на человеческое зрение, слух и другие органы восприятия окружающего мира. Экран, принтер и другие подключенные устройства для вывода информации сродни языку.

По сути, принцип работы компьютера можно изложить следующим образом:

Все составные части компьютера обладают определенной информацией. Под управлением процессора происходит обмен этими данными и их обработка.

Постоянная информация хранится на системном диске. При работе компьютера часть этих данных, необходимых для функционирования системы в данный момент, поступает в оперативную память. Сюда же загружаются данные с других устройств компьютера. Порядок обработки данных обеспечивает процессор. Результаты обработки данных выводятся на монитор или другие устройства.

Центральный процессор (CPU)

Основной составляющей любого компьютера является Central Processor Unit (CPU) или процессор. Существует большое разнообразие процессоров со своей определенной системой команд. Схему работы процессора можно представить таким образом: процессор получает определенные заранее последовательности, которые представляют собой определенный ряд чисел,

при этом каждое из них может быть расценено им в качестве команды, согласно заложенной в нем системой команд, или как ее еще называют - внутренней таблицей. В ней данные могут восприниматься по-разному, в зависимости от типа процессора. Стоит отметить, что интерпретация одного и того же числа в разных типах процессоров может быть различной, в том числе не содержать в себе команд. Это объясняется тем, что процессор принимает двоичные команды и программа, которая ориентирована на определенный тип процессора, не может быть прочитана и выполнена каким-либо другим его типом.

Память (memory)

После процессора стоит выделить еще одну важную составляющую часть любого компьютера – память, или запоминающее устройство. Принято выделять следующие виды памяти:

ОЗУ, которое применяется с целью записи данных на компьютер и их дальнейшего чтения (например, запись выходных результатов работы программы с целью дальнейшей работы с ними на каком-либо внешнем устройстве);

ПЗУ, которое хранит в памяти постоянные (неизменяемые в течение времени) данные.

Накопители

Отличительной особенностью ОЗУ от ПЗУ является их энергозависимость. Так, ПЗУ сохраняет все записанные в него данные не зависимо от поступления энергии в компьютер, в то время как в ОЗУ, при отключении электропитания или других проблем, связанных с электричеством, безвозвратно может их потерять.

Кроме вышеперечисленных принято выделять еще один тип памяти, получивший большое распространение совсем недавно, а именно флэш-память (flash memory). Она имеет ряд общих черт с ОЗУ и ПЗУ. Как и в ПЗУ при проблемах, связанных с питанием компьютера, данные, записанные во флэш-памяти, сохраняются. Кроме того, организация такого вида памяти позволяет осуществлять программную запись данных, как в ОЗУ. Во втором случае стоит отметить незначительный недостаток предоставленной возможности – запись данных происходит с более низкой скоростью, чем в ОЗУ.

Память в компьютере представлена в виде ячеек, которые служат для обращения процессора к конкретному месту в памяти. Для этого все ячейки памяти имеют свой собственный идентификатор – уникальный адрес, представленный в числовом виде, который организован в виде матрицы. Запрос к конкретной ячейке происходит в результате обращения к ней

процессора путем задания им номера строки и столбца матрицы, в которой она расположена. При этом для управления используют такие системные сигналы, такие как CAS и RAS, на которые можно наткнуться при настройке параметров BIOS, связанных с работой с памятью.

Старт любой программы начинается с необходимости загрузки ее данных в оперативную память, например с какого-либо внешнего устройства. При этом загрузка может быть как полной, так и частичной. После того, как программа загружена, процессор с помощью определенных функций: счетчика команд, распознает и приводит к выполнению команды, полученные им из ОЗУ, т. е. для выполнения этих операций у него есть отведенный для выполнения данных операций регистр, содержащий ячейки, в которых хранится следующая по порядку выполнения команда. Перед тем, как программа передана к исполнению, в счетчике ее команд записан адрес ячейки, в которой находится первая к исполнению команда, во время работы – данный счетчик с каждым обращением к ячейкам увеличивается, т. е. наполнение данного регистра возрастает до исполнения следующей по порядку программы.

Рассмотренная модель раскрывает процесс запуска и исполнения программы на компьютере достаточно грубо. Стоит отметить, что в современных реалиях процессоры расширили свою функциональность, а именно: появилась возможность выполнение следующих по порядку команд до того, как завершилась предыдущая; инициирование исполнения нескольких команд может происходить единовременно и т. д. Но, несмотря на это, основная идея работы остается неизменной.

Устройства ввода-вывода

Для организации «общения» с человеком есть необходимость в обязательном обеспечении компьютера устройствами, как ввода, так и вывода информации. Наиболее распространенным и являющимся и по настоящее время основным устройством, которое выступает клавиатура. Ввод информации осуществляется за счет клавиш, на которых представлены буквы, числа и другие символы. Занесение информации в этом случае происходит следующим образом: клавиша передает уникальный, соответствующий ей двоичный код в компьютер, после этого программа, находящаяся в ПЗУ, изменяет полученные ею коды в вид, который можно использовать в программах. Для отображения результатов, полученных в ходе работы за компьютером (например, текста, введенного с клавиатуры) нужно подсоединить к нему специальное устройство для вывода информации. Наиболее распространенным из таких устройств является монитор.

Для того чтобы вернуться к результатам проделанной ранее работы, необходимо его сохранить в какой-либо форме. Для этого были разработаны внешние накопители информации, позволяющие записать кроме основных результатов исполняемой программы, еще и ее текстовую составляющую (программный код). До изобретения таких накопителей люди активно использовали перфокарты. Примером их использования может быть запись в 80-х гг. для компьютеров обычных магнитофонных кассет, в этом случае «внешним накопителем информации» выступал магнитофон. В настоящее время для аналогичных целей применяют накопители расположенных на магнитных дисках, а также лазерные диски.

2. Средства автоматизации и основные функции систем

Что такое операционная система

Операционная система — самое важное программное обеспечение, благодаря которому пользователь может использовать мощности «железа» для решения конкретных задач. ОС координирует работу всех комплектующих и предоставляет каждой программе быстрый полный доступ к аппаратным ресурсам.

Операционная система, сокр. ОС (англ. operating system, OS) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений. Это определение применимо к большинству современных операционных систем общего назначения.

В большинстве вычислительных систем операционная система является основной, наиболее важной (а иногда и единственной) частью системного программного обеспечения. С 1990-х годов наиболее распространёнными операционными системами являются системы семейства Windows и системы класса UNIX (особенно Linux и Mac OS). С 2000-х большое распространение получили мобильные компьютеры (смартфоны и планшеты) и с ними ОС Android и iOS.

ОС является необходимой составляющей ПО ПК, без нее компьютер не может работать в принципе.

Операционная система есть в любом устройстве — планшете, компьютере, смартфоне, ноутбуке и т. д. Без нее не сможет функционировать ни один девайс, оставаясь просто набором комплектующих.

Чтобы пользователь мог свободно работать в различных приложениях, для операционных систем разработали графическую оболочку (интерфейс, GUI). Ранее для работы в ОС использовали интерфейс командной строки. Человек вводил особые команды, а компьютер воспроизводил их на черном экране. Сейчас этот способ неприемлем за счет своей замедленности и неудобства. Пользователи начали управлять машиной с помощью курсора и мыши только после внедрения графического интерфейса в Windows 95, что сразу обеспечило высокую популярность этой операционной системы.

Виды операционных систем

Покупая компьютер, вы получаете готовое к работе устройство с установленной операционной системой. По желанию ее можно усовершенствовать или заменить на другую, а также инсталлировать любую ОС поверх существующей. Графический интерфейс каждой системы (и даже ее версий) имеет различный стиль оформления. Каждый пользователь выбирает свой вариант для использования на персональном компьютере. Расскажем о трех самых популярных ОС: Windows, Linux и Mac OS.

Windows – самая популярная ОС

Рабочий стол Windows 10

Популярность ОС Windows от Microsoft подтверждена огромным количеством компьютерных пользователей — более чем 90%. Первая версия операционной системы появилась в 1985 г. под названием 1.0. По сути, это был усовершенствованный MS-DOS. Сегодня Microsoft выпустила Windows 10 и пока не собирается разрабатывать следующие версии. Компания создает только обновления, которые могут кардинально изменять интерфейс и функционал.

Преимущества ОС Windows:

Постоянная поддержка аппаратной части (видеокамер, аудио- и видеокарт и т. д.).

Простота инсталляции программного обеспечения (файл достаточно скачать и двойным кликом запустить установщик).

Понятный удобный интерфейс, который может освоить даже ребенок.

Большой выбор приложений для ОС Windows.

К недостаткам можно отнести длительный запуск ОС, постоянную инсталляцию обновлений, обеспечивающих безопасность, серьезные

проблемы при попадании вирусов, частую нестабильность системы и «пожирание» ресурсов.

Linux — надежная и безопасная ОС

Рабочий стол Ubuntu

ОС Linux основана на базе Unix, состоит из пакетов. Если ОС Windows отводит каждой программе свою папку, то Linux распределяет файлы по разным директориям. В одном месте лежат данные о запуске приложения, в другом — о конфигурации и т. д.

Преимущества ОС Linux:

Свободный доступ. Достаточно бесплатно скачать образ системы, записать его на флешку и инсталлировать на персональный компьютер.

Бесплатное ПО.

Преобразование софта. Продвинутые пользователи могут писать свои программы или изменять действующие.

Запуск программного обеспечения для Windows через специальные приложения — эмуляторы.

Оригинальность архитектуры ОС и трудности с созданием и активацией вредоносных кодов на Linux. Благодаря этой особенности операционная система отличается высокой безопасностью и служит надежным стражем значительной части серверов.

К минусам ОС Linux относятся небольшое количество поддерживаемых устройств (принтеров, аудио- и видеокарт), игр и приложений. Разнообразный софт остается в сфере действия Windows, т. к. производителям невыгодно создавать бесплатные программы. К тому же пользователям необходимо постоянно самообучаться и искать решения проблем с программной и аппаратной частью на различных тематических форумах.

Хотя Windows пользуется особой популярностью, Linux тоже имеет немало поклонников. С каждой новой версией системы разработчики увеличивают количество поддерживаемых устройств. Постепенно улучшают дистрибутивы: графический интерфейс становится более приятным, менеджеры пакетов — практичными, а программное обеспечение пополняется новыми продуктами

Mac OS — операционная система только для работы

Рабочий стол macOS Catalina

Apple разработала эту операционную систему для установки на свои же компьютеры. Как и Linux, она создана на основе Unix. Отличие от предыдущей ОС заключается в закрытом программном коде и платности продукта. В этом есть неоспоримые плюсы — разработчикам выгодно совершенствовать

старые и создавать новые приложения, ускорять их работу и повышать эффективность взаимодействия с аппаратной частью и другим софтом. Минус этой ОС достаточно ощутимый — для нее нет игр. Mac OS создана исключительно для работы, а не для отдыха.

Мобильные ОС

Техническая часть мобильных гаджетов очень сильно отличается от той, которой располагают компьютеры. Поэтому рабочие процессы на смартфонах, mp3-плеерах, планшетах и другие девайсах регулирует особое программное обеспечение: Android от Google, Windows Phone от Microsoft и iOS от Apple.

iOS

Эту операционную систему Apple устанавливает только на свои мобильные девайсы. Удобная, привлекательная, скоростная ОС. Многие пользователи, выбирая новый гаджет, отдают предпочтение именно этой качественной многофункциональной системе, для которой разработчики часто выпускают обновления. Закрытая iOS обладает высоким уровнем защиты. Система поддерживает работу более 1 млн приложений, но скачать их можно только в App Store.

Android

На ОС Android работает преобладающее количество современных мобильных гаджетов. Система является ответом на выход IOS. Android — открытая система, функциональная и очень удобная. Разработчики так же регулярно издают различные обновления, исправляющие ошибки и улучшающие работу пользователей. Для системы создано более миллиона приложений, которые можно устанавливать из Google Play и сторонних магазинов (последнее нельзя сделать на iOS).

Windows Phone

ОС имеет плавный, стабильный и быстрый интерфейс, отличную защиту от проникновения вирусов и регулярные обновления. Приложения из Windows Phone Store обладают оптимизированным кодом, благодаря чему отличаются высокой скоростью работы и экономичностью расходования заряда батареи. Microsoft интегрировал в ОС пакет Office 2013, позволяющий просматривать и редактировать на экране гаджета любой документ.

Но есть и некоторые минусы:

Недостаточное разнообразие софта в Windows Phone Store.

Минимум возможностей для персонализации экрана девайса.

Долгий путь к установке рингтона.

Нереализованный принцип многозадачности. Пользователь хоть и может запустить несколько приложений, но качественно функционировать будет только одно.

Если сложить все преимущества и недостатки, Windows Phone станет отличным выбором при решении большинства задач.

Типы разрядных операционных систем

Сначала дадим понятие разрядности. Это — число битов, которое способно одновременно обработать конкретное устройство. В нашем случае в роли последнего выступает операционная система. Сегодня используют 2 разрядности — 32- и 64-битную. Обозначают их, соответственно, x32 и x64. Первая одновременно обрабатывает 32 бита данных, вторая — 64.

Системы разных разрядностей отличаются и другими параметрами:

x32 задействует максимум 3,5 ГБ оперативной памяти, даже если объем последней значительно больше. x64 же поддерживает до 128 ГБ.

x64 работает и в 32-, и в 64-битном режиме, x32 — только в 32-битном.

Для x64 необходимы особые драйвера (которые можно так же легко найти).

Центральный процессор должен поддерживать 64-битные ОС, иначе они не будут работать.

Если на компьютере установлен многоядерный процессор (или процессоры), то x64 поднимет его производительность. Но, когда объем оперативной памяти составляет не более 4 ГБ, имеет смысл поставить x32 для экономии ресурсов ОЗУ. Многие пользователи считают, что x32 и x86 — это одна и та же система. Но это далеко не так. Сочетанием x86 обозначают архитектуру микропроцессора и аппаратную платформу. Она относится и к x32 (x86_32 bit), и к x64 (x86_64 bit).

Функции ОС

Операционная система эффективно распределяет ресурсы аппаратной части между установленным программным обеспечением, не допуская замедлений либо остановки процессов. ОС загружает процессор, управляет свободным пространством оперативной и постоянной памяти, инсталлирует, обновляет и удаляет различный софт (под руководством пользователя) и выполняет другие функции.

Сегодняшние операционные системы работают в режиме многозадачности — одновременно проводят множество процессов.

Какими должны быть ОС

Современные операционные системы должны отлично справляться с выполнением возложенных на них функций. Также пользователи ценят следующие особенности ОС:

Производительность. Обеспечение оптимального времени завершения операций и реагирования системы.

Надежность — работа без сбоев и отказов.

Расширяемость — возможность внедрения в систему новых модулей, реализующих дополнительные функции.

Совместимость — взаимодействие разных ОС или их версий. Новая версия может запускать приложения, разработанные для предыдущих вариаций. Или одна операционная система способна работать с софтом, созданным для другой (и наоборот).

Немаловажную роль играет и переносимость — способность ОС «переехать» на другую аппаратную платформу, претерпев минимум изменений.

Основные функции ОС:

Выполнение по запросу программ тех достаточно элементарных (низкоуровневых) действий, которые являются общими для большинства программ и часто встречаются почти во всех программах (ввод и вывод данных, запуск и остановка других программ, выделение и освобождение дополнительной памяти и др.).

Загрузка программ в оперативную память и их выполнение.

Стандартизованный доступ к периферийным устройствам (устройства ввода-вывода).

Управление оперативной памятью (распределение между процессами, организация виртуальной памяти).

Управление доступом к данным на энергонезависимых носителях (таких как жёсткий диск, оптические диски и др.), организованным в той или иной файловой системе.

Обеспечение пользовательского интерфейса.

Сетевые операции, поддержка стека сетевых протоколов.

Дополнительные функции:

Параллельное или псевдопараллельное выполнение задач (многозадачность).

Эффективное распределение ресурсов вычислительной системы между процессами.

Разграничение доступа различных процессов к ресурсам.

Организация надёжных вычислений (невозможности одного вычислительного процесса намеренно или по ошибке повлиять на вычисления в другом процессе), основана на разграничении доступа к ресурсам.

Взаимодействие между процессами: обмен данными, взаимная синхронизация.

Защита самой системы, а также пользовательских данных и программ от действий пользователей (злонамеренных или по незнанию) или приложений.

Многопользовательский режим работы и разграничение прав доступа.

Ядро — центральная часть операционной системы, управляющая выполнением процессов, ресурсами вычислительной системы и предоставляющая процессам координированный доступ к этим ресурсам. Основными ресурсами являются процессорное время, память и устройства ввода-вывода. Доступ к файловой системе и сетевое взаимодействие также могут быть реализованы на уровне ядра.

В состав ОС непременно входят:

Программный модуль, управляющий файловой системой: процесс работы компьютера сводится к обмену файлами между устройствами

Командный процессор – специальная программа, которая запрашивает у пользователя команды и выполняет их. Пользователь может дать команду запуска программы, выполнения какой-либо операции над файлами (копирование, удаление, переименование), вывода документа на печать и так далее. Операционная система должна эту команду выполнить

Драйверы – программы, которые управляют работой устройств. Каждому устройству соответствует свой драйвер. Технология «Plug and Play» (подключи и играй) позволяет автоматизировать подключение новых устройств. В процессе установки Windows определяет тип и конкретную модель установленного устройства и подключает необходимый для его функционирования драйвер. При включении компьютера произво-дится загрузка драйверов в оперативную память. Пользователь имеет возможность вручную установить или переустановить драйверы.

Программные модули графического интерфейса – программы, позволяющие пользователю вводить команды с помощью мыши.

Графический интерфейс пользователя (ГИП, ГПИ) — разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений.

В отличие от интерфейса командной строки, в ГПИ пользователь имеет произвольный доступ (с помощью устройств ввода — клавиатуры, мыши, джойстика и т. п.) ко всем видимым экранным объектам (элементам интерфейса) и осуществляет непосредственное манипулирование ими. Чаще всего элементы интерфейса в ГИ реализованы на основе метафор и отображают их назначение и свойства, что облегчает понимание и освоение программ неподготовленными пользователями.

Одним из требований к хорошему графическому интерфейсу программной системы является концепция «делай то, что я имею в виду» или DWIM (англ. Do What I Mean). DWIM требует, чтобы система работала предсказуемо, чтобы пользователь заранее интуитивно понимал, какое действие выполнит программа после получения его команды.

Утилиты – сервисные программы для обслуживания дисков (проверять, сжимать, дефрагментировать и тд), выполнения операций с файлами (архивировать, копировать и тд), и работы в компьютерных сетях

Справочная система – получение информации о функционировании ОС в целом и о работе её отдельных модулей

Файлы ОС хранятся во внешней, долговременной памяти (на жестком, гибком или лазерном диске). Но программы могут выполняться, только если они находятся в оперативной памяти, поэтому файлы ОС необходимо загрузить в оперативную память.

Загрузка ОС:

- Диск или флэш-накопитель, на котором находятся файлы операционной системы и с которого производится ее загрузка, называется системным.

- После включения ПК производится загрузка ОС с системного диска в оперативную память. Загрузка должна выполняться в соответствии с программой загрузки. Однако для того, чтобы компьютер выполнял какую-нибудь программу, эта программа должна уже находиться в оперативной памяти. Разрешение этого противоречия состоит в последовательной, поэтапной загрузке операционной системы.

- В ПЗУ содержатся программы тестирования ПК и первого этапа загрузки ОС — это BIOS (Basic Input/Output System — базовая система ввода/вывода).

- После включения питания процессор начинает выполнение программы самотестирования компьютера POST (Power-ON Self Test). Производится тестирование работоспособности процессора, памяти и других аппаратных средств компьютера

После проведения самотестирования специальная программа в BIOS, начинает поиск загрузчика ОС. Происходит поочередное обращение к имеющимся дискам и поиск на определенном месте (в первом загрузочном секторе диска) наличия специальной программы Master Boot (программы-загрузчика ОС).

- Master Boot загружается в ОС и ей передается управление работой компьютера. Программа ищет файлы операционной системы на системном диске и загружает их в оперативную память в качестве программных модулей

Инсталляция программ – процедура установки большинства программных продуктов, при которой используется специальная дистрибутивная копия.

Копирование программного продукта на жесткий диск называется установкой. Файл с программой имеет расширение .EXE .COM .BAT Он работает автономно или в сопровождении служебных файлов. Запустить программу – значит начать её работу. Но данный программный продукт должен быть совместим с аппаратными средствами

Итак, ОС выполняет две группы функций:

- предоставление пользователю или программисту вместо реальной аппаратуры компьютера расширенной виртуальной машины, с которой удобней работать и которую легче программировать;
- повышение эффективности использования компьютера путем рационального управления его ресурсами в соответствии с некоторым критерием.

Если говорить подробнее, то в функции ОС будут включены:

Управление процессами

Управление памятью

Управление файлами и внешними устройствами

Защита данных и администрирование

Интерфейс прикладного программирования

Пользовательский интерфейс

и т. д.

Если говорить о пользовательском интерфейсе, то операционная система должна обеспечивать удобный интерфейс не только для прикладных программ, но и для человека, работающего за терминалом.

В ранних операционных системах пакетного режима функции пользовательского интерфейса были сведены к минимуму и не требовали наличия терминала. Команды языка управления заданиями набивались на перфокарты, а результаты выводились на печатающее устройство.

Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов: алфавитно-цифровыми и графическими.

При работе за алфавитно-цифровым терминалом пользователь имеет в своем распоряжении систему команд, мощность который отражает функциональные возможности данной ОС. Обычно командный язык ОС позволяет запускать и останавливать приложения, выполнять различные операции с файлами и каталогами, получать информацию о состоянии ОС (количество работающих процессов, объем свободного пространства на дисках и т. п.), администрировать систему. Команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так называемого командного файла, содержащего некоторую последовательность команд.

Программный модуль ОС, ответственный за чтение отдельных команд или же последовательности команд из командного файла, иногда называют командным интерпретатором.

Ввод команды может быть упрощен, если операционная система поддерживает графический пользовательский интерфейс. В этом случае пользователь для выполнения нужного действия с помощью мыши выбирает на экране нужный пункт меню или графический символ.

3. Типы и назначение операционных систем

Типы операционной системы

Операционные системы существуют с самого первого поколения компьютеров, и они продолжают развиваться со временем. В этой главе мы обсудим некоторые из наиболее важных типов операционных систем, которые наиболее часто используются.

Пакетная операционная система

Пользователи пакетной операционной системы не взаимодействуют с компьютером напрямую. Каждый пользователь готовит свою работу на автономном устройстве, таком как перфокарты, и передает его оператору компьютера. Для ускорения обработки задания с аналогичными потребностями объединяются в группы и запускаются как группа. Программисты оставляют свои программы у оператора, а затем оператор сортирует программы с похожими требованиями в пакеты.

Проблемы с пакетными системами заключаются в следующем –

Отсутствие взаимодействия между пользователем и работой.

Процессор часто простаивает, потому что скорость механических устройств ввода / вывода ниже, чем у процессора.

Сложно предоставить желаемый приоритет.

Операционные системы с разделением времени

Разделение времени – это метод, который позволяет многим людям, находящимся на разных терминалах, одновременно использовать определенную компьютерную систему. Разделение времени или многозадачность является логическим продолжением мультипрограммирования. Время процессора, которое распределяется между несколькими пользователями одновременно, называется разделением времени.

Основное различие между многопрограммными пакетными системами и системами с разделением времени состоит в том, что в случае многопрограммных пакетных систем цель состоит в том, чтобы максимально использовать процессор, тогда как в системах с разделением времени цель состоит в том, чтобы минимизировать время отклика.

Несколько заданий выполняются центральным процессором путем переключения между ними, но переключения происходят так часто. Таким образом, пользователь может получить немедленный ответ. Например, при обработке транзакции процессор выполняет каждую пользовательскую программу в коротком пакете или такте вычислений. То есть, если присутствует n пользователей, то каждый пользователь может получить квант времени. Когда пользователь отправляет команду, время ответа составляет не более нескольких секунд.

Операционная система использует планирование ЦП и мультипрограммирование, чтобы предоставить каждому пользователю небольшую часть времени. Компьютерные системы, которые были разработаны в основном как пакетные системы, были преобразованы в системы с разделением времени.

Преимущества операционных систем с разделением времени, следующие:

Обеспечивает преимущество быстрого ответа.

Предотвращает дублирование программного обеспечения.

Уменьшает время простоя процессора.

Недостатки операционных систем с разделением времени, следующие:

Проблема надежности.

Вопрос безопасности и целостности пользовательских программ и данных.

Проблема передачи данных.

Распределенная операционная система

Распределенные системы используют несколько центральных процессоров для обслуживания нескольких приложений реального времени и нескольких пользователей. Задания по обработке данных распределяются между процессорами соответственно.

Процессоры связываются друг с другом через различные линии связи (например, высокоскоростные шины или телефонные линии). Они называются слабосвязанными системами или распределенными системами. Процессоры в распределенной системе могут различаться по размеру и функциям. Эти процессоры называются сайтами, узлами, компьютерами и т. д.

Преимущества распределенных систем, следующие:

Благодаря средству совместного использования ресурсов пользователь на одном сайте может использовать ресурсы, доступные на другом.

Ускорение обмена данными друг с другом по электронной почте.

Если в распределенной системе происходит сбой одного сайта, остальные сайты могут продолжать работать.

Лучший сервис для клиентов.

Снижение нагрузки на хост-компьютер.

Уменьшение задержек при обработке данных.

Сетевая операционная система

Сетевая операционная система работает на сервере и предоставляет серверу возможность управлять данными, пользователями, группами, безопасностью, приложениями и другими сетевыми функциями. Основная цель сетевой операционной системы – предоставить общий доступ к файлам и принтерам для нескольких компьютеров в сети, обычно в локальной сети (LAN), в частной сети или в других сетях.

Примеры сетевых операционных систем включают Microsoft Windows Server, UNIX, Linux, Mac OS, Novell NetWare и BSD.

Преимущества сетевых операционных систем, следующие:

Централизованные серверы очень стабильны.

Безопасность управляется сервером.

Обновления до новых технологий и оборудования могут быть легко интегрированы в систему.

Удаленный доступ к серверам возможен из разных мест и типов систем.

Недостатки сетевых операционных систем заключаются в следующем

Высокая стоимость покупки и запуска сервера.

Зависимость от центрального местоположения для большинства операций.

Требуется регулярное обслуживание и обновления.

Операционная система реального времени

Система реального времени определяется как система обработки данных, в которой интервал времени, необходимый для обработки и реагирования на входные данные, настолько мал, что он контролирует среду. Время, необходимое системе для ответа на ввод и отображение необходимой обновленной информации, называется временем отклика. Таким образом, в этом методе время отклика меньше по сравнению с онлайн-обработкой.

Системы реального времени используются, когда существуют жесткие требования к времени работы процессора или поток данных, а системы реального времени могут использоваться в качестве устройства управления в выделенном приложении. Операционная система реального времени должна иметь четко определенные фиксированные временные ограничения, в противном случае система выйдет из строя. Например, научные эксперименты, системы медицинской визуализации, промышленные системы управления, системы вооружения, роботы, системы управления воздушным движением и т. д.

Существует два типа операционных систем реального времени.

Жесткие системы реального времени

Жесткие системы реального времени гарантируют, что критические задачи будут выполнены вовремя. В жестких системах реального времени вторичное хранилище ограничено или отсутствует, а данные хранятся в ПЗУ. В этих системах виртуальная память почти никогда не обнаруживается.

Мягкие системы реального времени

Мягкие системы реального времени менее ограничены. Критическая задача в реальном времени получает приоритет над другими задачами и сохраняет приоритет до завершения. Мягкие системы реального времени имеют ограниченную полезность, чем жесткие системы реального времени. Например, мультимедиа, виртуальная реальность, передовые научные проекты, такие как подводные исследования и планетарные роверы, и т. Д.

Операционная система – Услуги

Операционная система предоставляет услуги как пользователям, так и программам.

Он предоставляет программам среду для выполнения.

Он предоставляет пользователям сервисы для удобного выполнения программ.

Ниже приведены несколько общих служб, предоставляемых операционной системой.

Выполнение программы

Операции ввода / вывода

Манипулирование файловой системой

связь

Обнаружение ошибок

Распределение ресурсов

защита

Выполнение программы

Операционные системы обрабатывают многие виды действий от пользовательских программ до системных программ, таких как спулер принтера, серверы имен, файловый сервер и т. Д. Каждое из этих действий инкапсулируется как процесс.

Процесс включает в себя полный контекст выполнения (код для выполнения, данные для манипулирования, регистры, используемые ресурсы ОС). Ниже приведены основные действия операционной системы в отношении управления программами.

- Загружает программу в память.
- Выполняет программу.
- Управляет выполнением программы.
- Предоставляет механизм для синхронизации процессов.
- Предоставляет механизм для коммуникации процесса.
- Предоставляет механизм для тупиковой обработки.

Операция ввода / вывода

Подсистема ввода / вывода состоит из устройств ввода / вывода и их соответствующего программного драйвера. Драйверы скрывают особенности конкретных аппаратных устройств от пользователей.

Операционная система управляет связью между пользователем и драйверами устройств.

- Операция ввода / вывода означает операцию чтения или записи с любым файлом или любым конкретным устройством ввода / вывода.
- Операционная система обеспечивает доступ к требуемому устройству ввода-вывода при необходимости.

Манипулирование файловой системой

Файл представляет собой коллекцию связанной информации. Компьютеры могут хранить файлы на диске (вторичное хранилище) для длительного хранения. Примеры носителей включают магнитную ленту, магнитный диск и дисководы оптических дисков, такие как CD, DVD. Каждый из этих носителей имеет свои собственные свойства, такие как скорость, емкость, скорость передачи данных и методы доступа к данным.

Файловая система обычно организована в каталоги для удобства навигации и использования. Эти каталоги могут содержать файлы и другие направления. Ниже приведены основные действия операционной системы в отношении управления файлами.

- Программа должна прочитать файл или записать файл.
- Операционная система дает разрешение программе для работы с файлом.
- Разрешение варьируется от только для чтения, чтения-записи, отказа и т. Д.
- Операционная система предоставляет пользователю интерфейс для создания / удаления файлов.
- Операционная система предоставляет пользователю интерфейс для создания / удаления каталогов.
- Операционная система предоставляет интерфейс для создания резервной копии файловой системы.

Связь

В случае распределенных систем, которые представляют собой совокупность процессоров, которые не разделяют память, периферийные устройства или часы, операционная система управляет связью между всеми процессами. Несколько процессов взаимодействуют друг с другом через линии связи в сети.

ОС обрабатывает стратегии маршрутизации и подключения, а также проблемы конкуренции и безопасности. Ниже приведены основные действия операционной системы в отношении связи:

Два процесса часто требуют передачи данных между ними

Оба процесса могут быть на одном компьютере или на разных компьютерах, но связаны через компьютерную сеть.

Связь может быть реализована двумя способами: либо с помощью общей памяти, либо путем передачи сообщений.

Обработка ошибок

Ошибки могут возникнуть в любое время и в любом месте. Ошибка может возникать в ЦП, в устройствах ввода-вывода или в оборудовании памяти. Ниже приведены основные действия операционной системы в отношении обработки ошибок:

ОС постоянно проверяет на возможные ошибки.

ОС предпринимает соответствующие действия для обеспечения правильных и согласованных вычислений.

Управление ресурсами

В случае многопользовательской или многозадачной среды ресурсы, такие как основная память, циклы ЦП и хранилище файлов, должны быть выделены для каждого пользователя или задания. Ниже приведены основные действия операционной системы в отношении управления ресурсами –

ОС управляет всеми видами ресурсов с помощью планировщиков.

Алгоритмы планирования ЦП используются для лучшего использования ЦП.

Защита

С учетом того, что компьютерная система имеет несколько пользователей и одновременно выполняет несколько процессов, различные процессы должны быть защищены от действий друг друга.

Под защитой понимается механизм или способ управления доступом программ, процессов или пользователей к ресурсам, определенным компьютерной системой. Ниже приведены основные действия операционной системы в отношении защиты:

ОС гарантирует, что весь доступ к системным ресурсам контролируется.

ОС обеспечивает защиту внешних устройств ввода-вывода от недопустимых попыток доступа.

ОС предоставляет функции аутентификации для каждого пользователя с помощью паролей.

Операционная система – Свойства

Пакетная обработка

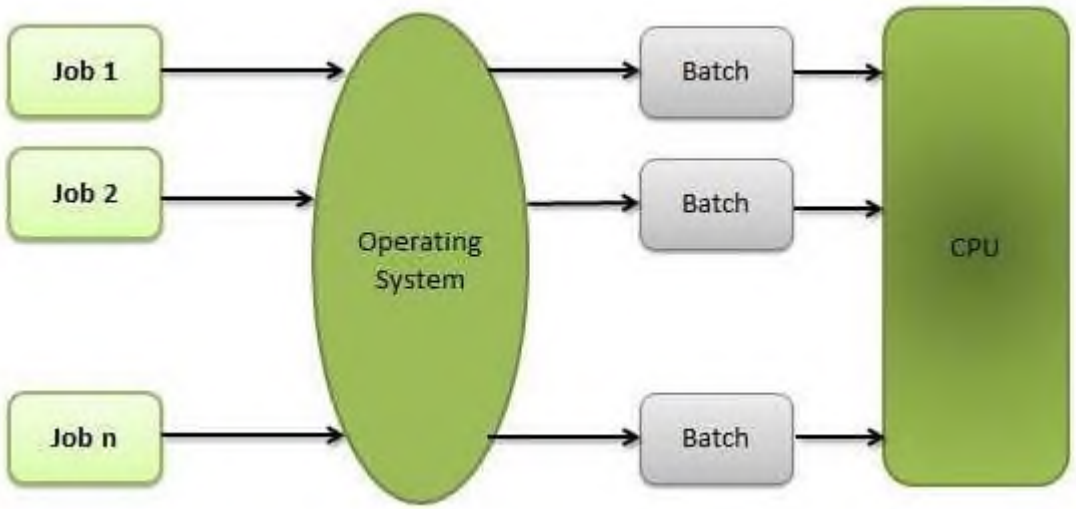
Пакетная обработка – это метод, при котором операционная система собирает программы и данные вместе в пакете до начала обработки. Операционная система выполняет следующие действия, связанные с пакетной обработкой:

ОС определяет задание, которое имеет предопределенную последовательность команд, программ и данных как единое целое.

ОС хранит ряд заданий в памяти и выполняет их без какой-либо ручной информации.

Задания обрабатываются в порядке поступления, т. Е. В порядке поступления.

Когда задание завершает свое выполнение, его память освобождается, а выходные данные задания копируются в выходную папку для последующей печати или обработки.



Преимущества

Пакетная обработка переносит большую часть работы оператора на компьютер.

Повышение производительности при запуске новой работы, как только завершится предыдущая, без какого-либо ручного вмешательства.

Пакетная обработка переносит большую часть работы оператора на компьютер.

Повышение производительности при запуске новой работы, как только завершится предыдущая, без какого-либо ручного вмешательства.

Недостатки

Сложно отлаживать программу.

Работа может войти в бесконечный цикл.

Из-за отсутствия схемы защиты одно пакетное задание может повлиять на ожидающие задания.

Многозадачность

Многозадачность – это когда одновременно выполняется несколько заданий процессором, переключаясь между ними. Переключения происходят настолько часто, что пользователи могут взаимодействовать с каждой программой во время ее работы. ОС выполняет следующие действия, связанные с многозадачностью –

Пользователь дает инструкции операционной системе или программе напрямую и получает немедленный ответ.

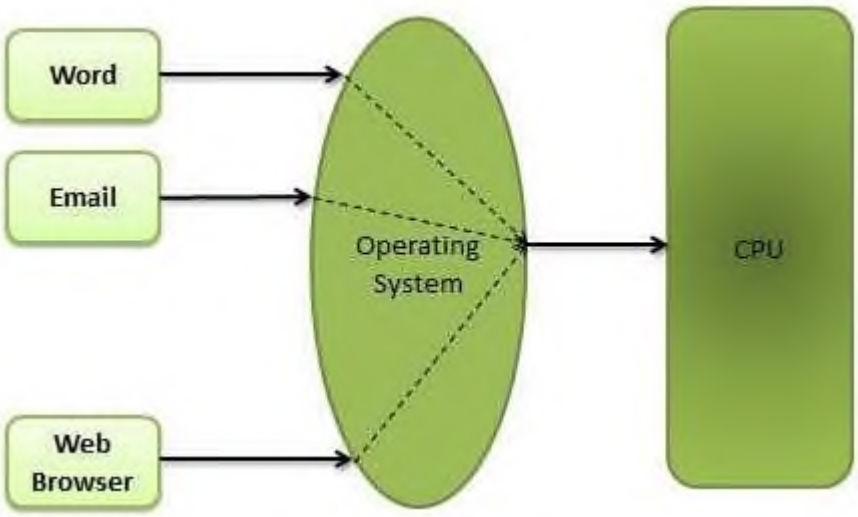
Операционная система обрабатывает многозадачность таким образом, что она может обрабатывать несколько операций / одновременно выполнять несколько программ.

Многозадачные операционные системы также известны как системы с разделением времени.

Эти операционные системы были разработаны для обеспечения интерактивного использования компьютерной системы по разумной цене.

Операционная система с общим временем использует концепцию планирования ЦП и мультипрограммирования, чтобы предоставить каждому пользователю небольшую часть ЦП с общим временем.

У каждого пользователя есть хотя бы одна отдельная программа в памяти.



Многозадачность

Программа, которая загружается в память и выполняется, обычно называется процессом.

Когда процесс выполняется, он обычно выполняется в течение очень короткого времени, прежде чем он либо завершится, либо должен будет выполнить ввод-вывод.

Поскольку интерактивный ввод-вывод обычно выполняется с меньшей скоростью, его выполнение может занять много времени. В течение этого времени процессор может использоваться другим процессом.

Операционная система позволяет пользователям одновременно использовать компьютер. Поскольку каждое действие или команда в системе с разделением по времени имеет тенденцию быть коротким, для каждого пользователя требуется только немного процессорного времени.

Поскольку система быстро переключает ЦП с одного пользователя / программы на другого, у каждого пользователя создается впечатление, что он

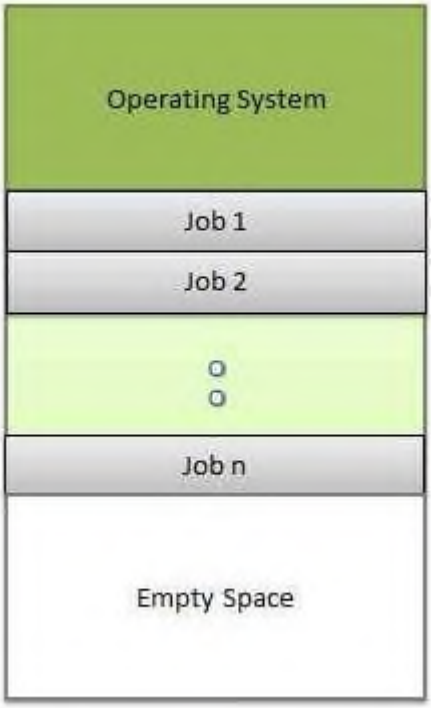
/ она имеет свой собственный ЦП, тогда как фактически один ЦП распределяется между многими пользователями.

Мультипрограммирование

Совместное использование процессора, когда две или более программ находятся в памяти одновременно, называется многопрограммным. Мультипрограммирование предполагает использование одного общего процессора. Мультипрограммирование увеличивает загрузку ЦП, организуя задания так, что ЦП всегда должен их выполнять.

На следующем рисунке показана схема памяти для многопрограммной системы.

Расположение памяти



ОС выполняет следующие действия, связанные с мультипрограммированием.

Операционная система хранит несколько заданий в памяти одновременно.

Этот набор заданий является подмножеством заданий, хранящихся в пуле заданий.

Операционная система выбирает и начинает выполнять одно из заданий в памяти.

Многопрограммные операционные системы контролируют состояние всех активных программ и системных ресурсов, используя программы управления памятью, чтобы гарантировать, что ЦП никогда не находится в режиме ожидания, если нет заданий для обработки.

Преимущества

- Высокая и эффективная загрузка процессора.
- Пользователь чувствует, что многим программам отводится процессор практически одновременно.

Недостатки

Планирование ЦП не требуется.

Чтобы разместить много заданий в памяти, требуется управление памятью.

Интерактивность

Интерактивность относится к способности пользователей взаимодействовать с компьютерной системой. Операционная система выполняет следующие действия, связанные с интерактивностью:

Предоставляет пользователю интерфейс для взаимодействия с системой.

Управляет устройствами ввода, чтобы принимать входные данные от пользователя. Например, клавиатура.

Управляет устройствами вывода, чтобы показать результаты пользователю. Например, Монитор.

Время отклика ОС должно быть коротким, поскольку пользователь отправляет и ждет результата.

Система реального времени

Системы реального времени обычно являются встроенными системами. Операционная система выполняет следующие действия, связанные с работой системы в режиме реального времени.

В таких системах операционные системы обычно считывают данные с датчиков и реагируют на них.

Операционная система должна гарантировать реакцию на события в течение фиксированных периодов времени для обеспечения правильной работы.

Распределенная среда

Распределенная среда относится к нескольким независимым процессорам или процессорам в компьютерной системе. Операционная система выполняет следующие действия, связанные с распределенной средой:

ОС распределяет логику вычислений между несколькими физическими процессорами.

Процессоры не делят память или часы. Вместо этого каждый процессор имеет свою собственную локальную память.

ОС управляет связью между процессорами. Они общаются друг с другом через различные линии связи.

Намоточные

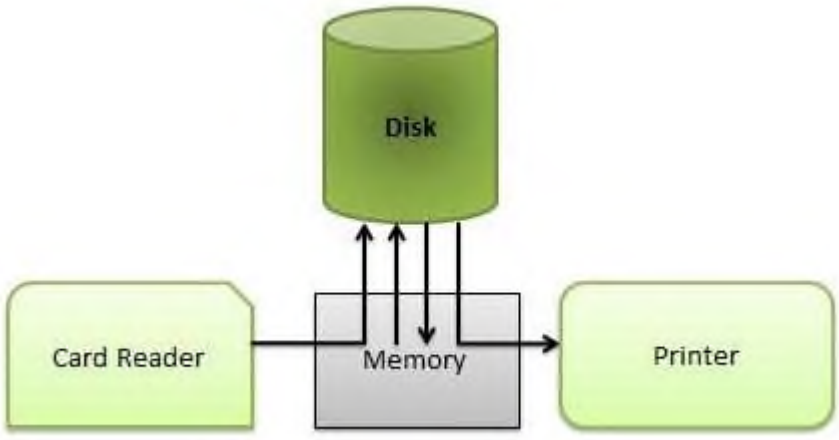
Буферизация является аббревиатурой для одновременных периферийных операций в режиме онлайн. Под спулингом понимается помещение данных различных заданий ввода / вывода в буфер. Этот буфер представляет собой специальную область в памяти или на жестком диске, которая доступна для устройств ввода-вывода.

Операционная система выполняет следующие действия, связанные с распределенной средой:

Управляет буферизацией данных устройств ввода-вывода, поскольку устройства имеют разные скорости доступа к данным.

Поддерживает буфер буфера, который обеспечивает станцию ожидания, где данные могут храниться, пока медленное устройство догоняет.

Поддерживает параллельные вычисления из-за процесса буферизации, поскольку компьютер может выполнять ввод / вывод параллельно. Становится возможным, чтобы компьютер считывал данные с ленты, записывал данные на диск и записывал на ленточный принтер, пока он выполняет свою вычислительную задачу.



Преимущества

Операция буферизации использует диск в качестве очень большого буфера.

Буферизация может перекрывать операции ввода-вывода для одного задания с процессорами для другого задания.

Операционная система – Процессы

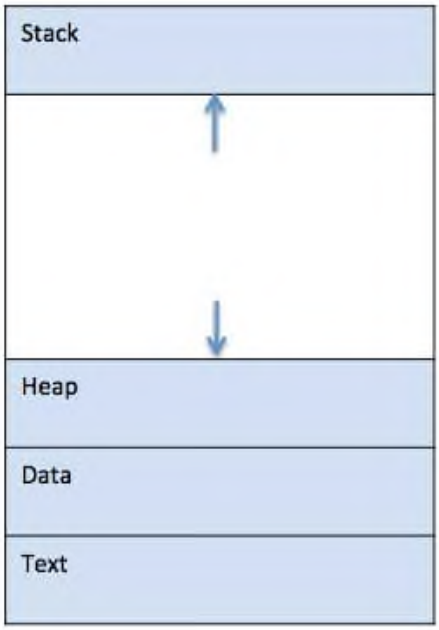
Процесс

Процесс – это в основном программа в исполнении. Выполнение процесса должно выполняться последовательно.

Процесс определяется как объект, который представляет собой базовую единицу работы, которая должна быть реализована в системе.

Проще говоря, мы пишем наши компьютерные программы в текстовом файле, и когда мы выполняем эту программу, она становится процессом, который выполняет все задачи, упомянутые в программе.

Когда программа загружается в память и становится процессом, ее можно разделить на четыре части: стек, куча, текст и данные. На следующем рисунке показана упрощенная схема процесса внутри основной памяти –



N	Компонент и описание
1	**стек** Стек процесса содержит временные данные, такие как параметры метода / функции, адрес возврата и локальные переменные.
2	**отвал** Это динамически выделяемая память для процесса во время его выполнения.
3	**Текст** Это включает текущую активность, представленную значением счетчика программы и содержимым регистров процессора.
4	**Данные** Этот раздел содержит глобальные и статические переменные.

Этот раздел содержит глобальные и статические переменные.

Стек

Стек процесса содержит временные данные, такие как параметры метода / функции, адрес возврата и локальные переменные.

Отвал

Это динамически выделяемая память для процесса во время его выполнения.

Текст

Это включает текущую активность, представленную значением счетчика программы и содержимым регистров процессора.

Данные

Этот раздел содержит глобальные и статические переменные.

Программа

Программа – это фрагмент кода, который может состоять из одной строки или миллионов строк. Компьютерная программа обычно пишется программистом на языке программирования. Например, вот простая программа, написанная на языке программирования C –

```
#include <stdio.h>

int main() {
   printf("Hello, World! \n");
   return 0;
}
```

Компьютерная программа – это набор инструкций, которые выполняют определенную задачу при выполнении компьютером. Когда мы сравниваем программу с процессом, мы можем заключить, что процесс является динамическим экземпляром компьютерной программы.

Часть компьютерной программы, которая выполняет четко определенную задачу, называется алгоритмом. Коллекция компьютерных программ, библиотек и связанных данных называется программным обеспечением.

Жизненный цикл процесса

Когда процесс выполняется, он проходит через различные состояния. Эти этапы могут отличаться в разных операционных системах, и названия этих состояний также не стандартизированы.

В общем случае процесс может иметь одно из следующих пяти состояний одновременно.

SN	Состояние и описание
1	**Начните**

	Это начальное состояние при первом запуске / создании процесса.
2	**готовы** Процесс ожидает назначения процессору. Готовые процессы ожидают, чтобы процессор был выделен им операционной системой, чтобы они могли работать. Процесс может войти в это состояние после состояния **запуска** или во время его выполнения, но прерывается планировщиком, чтобы назначить ЦП другому процессу.
3	**Бег** Как только процесс был назначен процессору планировщиком ОС, состояние процесса устанавливается на выполнение, и процессор выполняет свои инструкции.
4	**ждущий** Процесс переходит в состояние ожидания, если ему нужно дождаться ресурса, например, ждать ввода пользователя или ждать, пока файл станет доступным.
5	**Прекращено или Выход** Как только процесс завершает свое выполнение или он завершается операционной системой, он переводится в состояние завершения, где он ожидает удаления из основной памяти.

Начните

Это начальное состояние при первом запуске / создании процесса.

Готовы

Процесс ожидает назначения процессору. Готовые процессы ожидают, чтобы процессор был выделен им операционной системой, чтобы они могли работать. Процесс может войти в это состояние после состояния запуска или во время его выполнения, но прерывается планировщиком, чтобы назначить ЦП другому процессу.

Бег

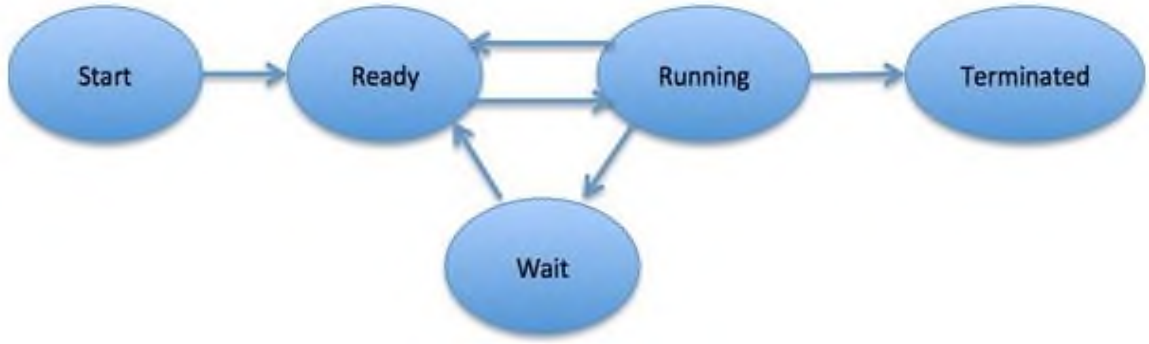
Как только процесс был назначен процессору планировщиком ОС, состояние процесса устанавливается на выполнение, и процессор выполняет свои инструкции.

Ждущий

Процесс переходит в состояние ожидания, если ему нужно дождаться ресурса, например, ждать ввода пользователя или ждать, пока файл станет доступным.

Прекращено или Выход

Как только процесс завершает свое выполнение или он завершается операционной системой, он переводится в состояние завершения, где он ожидает удаления из основной памяти.



Блок управления процессом (PCB)

Блок управления процессом – это структура данных, поддерживаемая операционной системой для каждого процесса. Печатная плата идентифицируется целочисленным идентификатором процесса (PID). PCB хранит всю информацию, необходимую для отслеживания процесса, как указано ниже в таблице –

N	Информация и описание
1	**Состояние процесса** Текущее состояние процесса, т. Е. Готов ли он, работает, ждет или что-то еще.
2	**Процессные привилегии** Это необходимо для разрешения / запрета доступа к системным ресурсам.
3	**Идентификатор процесса** Уникальная идентификация для каждого процесса в операционной системе.
4	**Указатель** Указатель на родительский процесс.
5	**Счетчик команд** Счетчик программ – это указатель на адрес следующей инструкции, которая будет выполнена для этого процесса.
6	**Регистры процессора** Различные регистры процессора, где процесс должен быть сохранен для выполнения в рабочем состоянии.

7	**Информация о планировании процессора** Приоритет процесса и другая информация о планировании, которая требуется для планирования процесса.
8	**Информация об управлении памятью** Это включает в себя информацию таблицы страниц, пределов памяти, таблицы сегментов в зависимости от памяти, используемой операционной системой.
9	**Бухгалтерская информация** Это включает количество ЦП, использованного для выполнения процесса, ограничения по времени, идентификатор выполнения и т. д.
10	**Информация о состоянии ввода-вывода** Это включает в себя список устройств ввода-вывода, выделенных для процесса

Это включает в себя список устройств ввода-вывода, выделенных для процесса.

Состояние процесса

Текущее состояние процесса, т. Е. Готов ли он, работает, ждет или что-то еще.

Процессные привилегии

Это необходимо для разрешения / запрета доступа к системным ресурсам.

Идентификатор процесса

Уникальная идентификация для каждого процесса в операционной системе.

Указатель

Указатель на родительский процесс.

Счетчик команд

Счетчик программ – это указатель на адрес следующей инструкции, которая будет выполнена для этого процесса.

Регистры процессора

Различные регистры процессора, где процесс должен быть сохранен для выполнения в рабочем состоянии.

Информация о планировании процессора

Приоритет процесса и другая информация о планировании, которая требуется для планирования процесса.

Информация об управлении памятью

Это включает в себя информацию таблицы страниц, пределов памяти, таблицы сегментов в зависимости от памяти, используемой операционной системой.

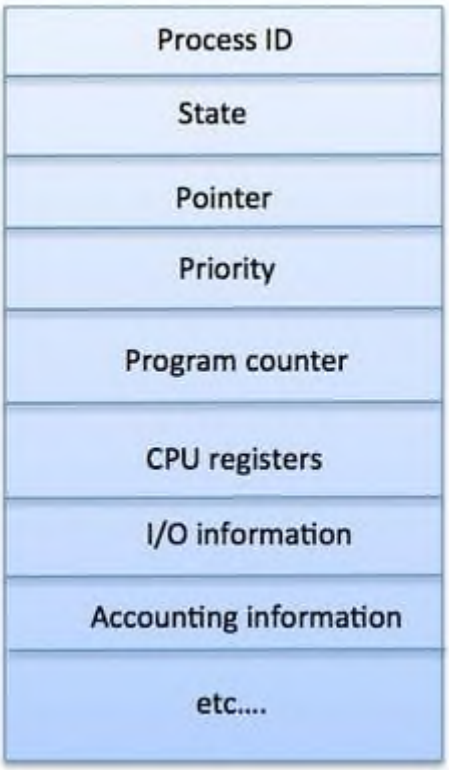
Бухгалтерская информация

Это включает количество ЦП, использованного для выполнения процесса, ограничения по времени, идентификатор выполнения и т. Д.

Информация о состоянии ввода-вывода

Это включает в себя список устройств ввода-вывода, выделенных для процесса.

Архитектура печатной платы полностью зависит от операционной системы и может содержать разную информацию в разных операционных системах. Вот упрощенная схема печатной платы –



Печатная плата поддерживается для процесса в течение всего срока его службы и удаляется после завершения процесса.

Операционная система – планирование процессов

Определение

Планирование процессов – это деятельность диспетчера процессов, которая обрабатывает удаление запущенного процесса из ЦП и выбор другого процесса на основе конкретной стратегии.

Планирование процессов является неотъемлемой частью многопрограммных операционных систем. Такие операционные системы

позволяют загружать в исполняемую память более одного процесса за один раз, и загруженный процесс совместно использует ЦП, используя временное мультиплексирование.

Очереди планирования процессов

ОС поддерживает все печатные платы в очередях планирования процессов. ОС поддерживает отдельную очередь для каждого из состояний процессов, а печатные платы всех процессов в одном и том же состоянии выполнения помещаются в одну и ту же очередь. Когда состояние процесса изменяется, его печатная плата отсоединяется от текущей очереди и перемещается в новую очередь состояний.

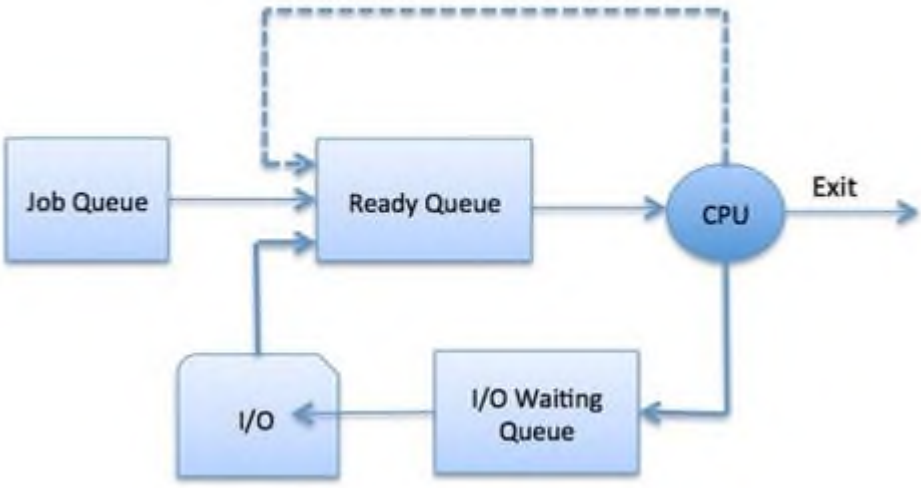
Операционная система поддерживает следующие важные очереди планирования процессов:

Очередь заданий – в этой очереди хранятся все процессы в системе.

Готовая очередь – эта очередь хранит набор всех процессов, находящихся в основной памяти, готовых и ожидающих выполнения. Новый процесс всегда помещается в эту очередь.

Очереди устройства – процессы, которые заблокированы из-за недоступности устройства ввода-вывода, составляют эту очередь.

Планирование процесса



ОС может использовать разные политики для управления каждой очередью (FIFO, Round Robin, Priority и т. Д.). Планировщик ОС определяет, как перемещать процессы между готовыми и запущенными очередями, которые могут иметь только одну запись на процессорное ядро в системе; на приведенной выше диаграмме он был объединен с процессором.

Модель двух состояний

Модель процесса с двумя состояниями относится к рабочим и неработающим состояниям, которые описаны ниже –

SN	Состояние и описание
1	**Бег** Когда создается новый процесс, он входит в систему как в рабочем состоянии.
2	**Не бегать** Процессы, которые не запущены, остаются в очереди, ожидая своей очереди на выполнение. Каждая запись в очереди является указателем на определенный процесс. Очередь реализуется с помощью связанного списка. Использование диспетчера заключается в следующем. Когда процесс прерывается, этот процесс передается в очередь ожидания. Если процесс завершен или прерван, процесс отбрасывается. В любом случае диспетчер затем выбирает процесс из очереди для выполнения.

Планировщики

Планировщики – это специальное системное программное обеспечение, которое обрабатывает планирование процессов различными способами. Их основная задача – выбрать задания для отправки в систему и решить, какой процесс запустить. Планировщики бывают трех типов –

- Долгосрочный планировщик
- Краткосрочный планировщик
- Среднесрочный планировщик

Долгосрочный планировщик

Это также называется планировщиком заданий. Долгосрочный планировщик определяет, какие программы допущены в систему для обработки. Он выбирает процессы из очереди и загружает их в память для выполнения. Процесс загружается в память для планирования ЦП.

Основной целью планировщика заданий является обеспечение сбалансированного сочетания заданий, таких как привязка ввода-вывода и привязка процессора. Он также контролирует степень мультипрограммирования. Если степень мультипрограммирования стабильна, то средняя скорость создания процесса должна быть равна средней скорости вылета процессов, покидающих систему.

В некоторых системах долгосрочный планировщик может быть недоступен или минимален. Операционные системы с разделением времени не имеют долгосрочного планировщика. Когда процесс меняет состояние с нового на готовое, тогда используется долгосрочный планировщик.

Краткосрочный планировщик

Он также называется планировщиком ЦП. Его основная цель – повысить производительность системы в соответствии с выбранным набором критериев. Это изменение состояния готовности в рабочее состояние процесса. Планировщик ЦП выбирает процесс среди процессов, готовых к выполнению, и выделяет ЦП одному из них.

Краткосрочные планировщики, также известные как диспетчеры, принимают решение о том, какой процесс выполнять дальше. Краткосрочные планировщики быстрее, чем долгосрочные.

Среднесрочный планировщик

Среднесрочное планирование является частью обмена. Удаляет процессы из памяти. Это уменьшает степень мультипрограммирования. Среднесрочный планировщик отвечает за обработку замененных процессов.

Работающий процесс может быть приостановлен, если он сделает запрос ввода-вывода. Приостановленные процессы не могут достичь прогресса. В этом случае, чтобы удалить процесс из памяти и освободить место для других процессов, приостановленный процесс перемещается во вторичное хранилище. Этот процесс называется обменом, а процесс называется обмен или выкатывание. Обмен может быть необходимым для улучшения технологического процесса.

Сравнение среди Планировщика

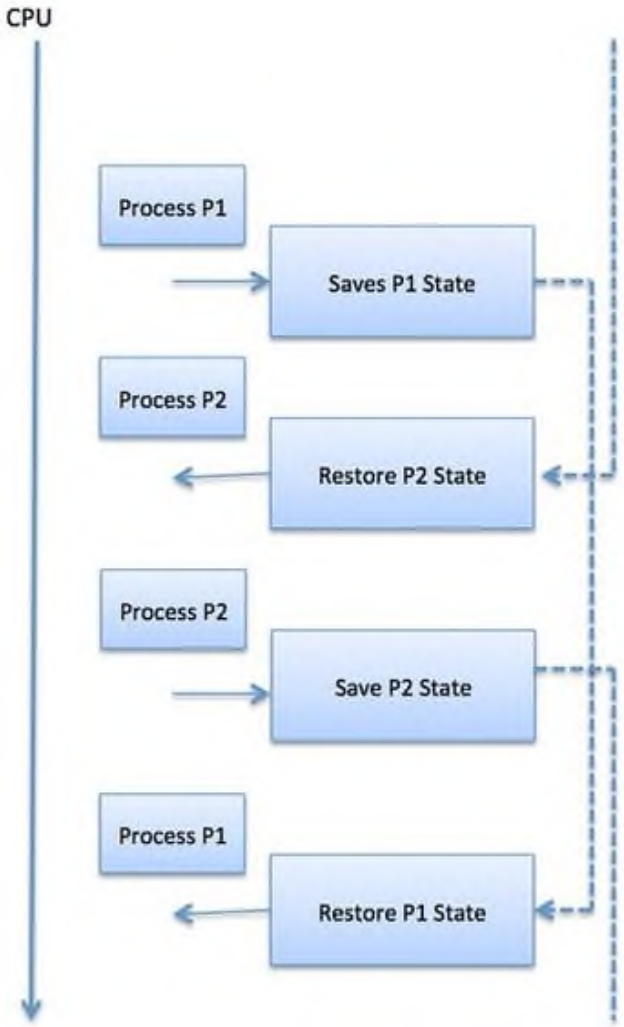
SN	Долгосрочный планировщик	Краткосрочный планировщик	Среднесрочный планировщик
1	Это планировщик работы	Это планировщик процессора	Это планировщик обмена процессами.
2	Скорость меньше, чем краткосрочный планировщик	Скорость самая быстрая среди двух других	Скорость находится между краткосрочным и долгосрочным планировщиком.
3	Контролирует степень мультипрограммирования	Это обеспечивает меньший контроль над степенью мультипрограммирования	Это уменьшает степень мультипрограммирования.
4	Она практически отсутствует или	Это также минимальная система разделения времени	Это часть систем разделения времени.

	минимальна по времени		
5	Он выбирает процессы из пула и загружает их в память для выполнения	Он выбирает те процессы, которые готовы выполнить	Это может повторно ввести процесс в память, и выполнение может быть продолжено.

Переключение контекста

Переключение контекста – это механизм для сохранения и восстановления состояния или контекста ЦП в блоке управления процессом, так что выполнение процесса может быть возобновлено с той же точки в более позднее время. Используя эту технику, переключатель контекста позволяет нескольким процессам совместно использовать один ЦП. Переключение контекста является неотъемлемой частью функций многозадачной операционной системы.

Когда планировщик переключает ЦП с выполнения одного процесса на выполнение другого, состояние текущего запущенного процесса сохраняется в блоке управления процессом. После этого состояние следующего процесса загружается с его собственной печатной платы и используется для настройки ПК, регистров и т. Д. В этот момент может начаться выполнение второго процесса.



Переключение контекста требует значительных вычислительных ресурсов, поскольку регистр и состояние памяти должны быть сохранены и восстановлены. Чтобы избежать количества времени переключения контекста, некоторые аппаратные системы используют два или более набора регистров процессора. Когда процесс переключается, следующая информация сохраняется для последующего использования.

- Счетчик команд
- Планирование информации
- Базовое и предельное значение регистра
- В настоящее время используется регистр
- Измененное состояние
- Информация о состоянии ввода / вывода
- Бухгалтерская информация

Алгоритмы планирования операционной системы

Планировщик процессов планирует различные процессы, которые должны быть назначены ЦПУ на основе определенных алгоритмов планирования. Существует шесть популярных алгоритмов планирования процессов, которые мы собираемся обсудить в этой главе:

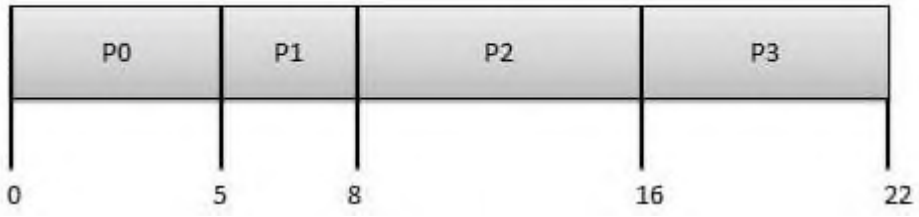
- Планирование «первым пришел – первым обслужен» (FCFS)
- Планирование Shortest-Job-Next (SJN)
- Приоритетное планирование
- Самое короткое оставшееся время
- Круглый Робин (RR) Планирование
- Планирование многоуровневых очередей

Эти алгоритмы являются либо не вытесняющими, ни вытесняющими. Непрерывающие алгоритмы разработаны таким образом, что, как только процесс входит в рабочее состояние, он не может быть прерван до тех пор, пока не завершит свое выделенное время, тогда как упреждающее планирование основано на приоритете, когда планировщик может выгрузить процесс с низким приоритетом в любое время, когда высокий приоритет процесс переходит в состояние готовности.

First Come First Serve (FCFS)

- Задания выполняются в порядке поступления.
- Это не упреждающий, упреждающий алгоритм планирования.
- Легко понять и реализовать.
- Его реализация основана на очереди FIFO.
- Низкая производительность, так как среднее время ожидания велико.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Время ожидания каждого процесса выглядит следующим образом –

Процесс	**Время ожидания: Время обслуживания – Время прибытия**
P0	0 – 0 = 0
P1	5–1 = 4

P2	8–2 = 6
P3	16–3 = 13

Среднее время ожидания: (0 + 4 + 6 + 13) / 4 = 5,75

Кратчайшая работа следующая (SJN)

Это также называется первой короткой работой, или SJF

Это не упреждающий, упреждающий алгоритм планирования.

Лучший подход для минимизации времени ожидания.

Легко реализовать в пакетных системах, где необходимое время процессора известно заранее.

Невозможно реализовать в интерактивных системах, где необходимое время ЦП неизвестно.

Процессор должен заранее знать, сколько времени займет процесс.

Дано: таблица процессов и время их прибытия, время выполнения

Процесс	**Время прибытия**	**Время исполнения**	**Время обслуживания**
P0	0	5	0
P1	1	3	5
P2	2	8	14
P3	3	6	8

Время ожидания каждого процесса выглядит следующим образом –

Процесс	**Время ожидания**
P0	0 – 0 = 0
P1	5–1 = 4
P2	14–2 = 12
P3	8–3 = 5

Среднее время ожидания: (0 + 4 + 12 + 5) / 4 = 21/4 = 5,25

Приоритетное планирование

Приоритетное планирование – это алгоритм без вытеснения и один из самых распространенных алгоритмов планирования в пакетных системах.

Каждому процессу назначается приоритет. Процесс с наивысшим приоритетом должен выполняться первым и так далее.

Процессы с одинаковым приоритетом выполняются по принципу «первым пришел – первым обслужен».

Приоритет может быть решен на основе требований к памяти, времени или любых других ресурсов.

Дано: таблица процессов, их время прибытия, время выполнения и приоритет. Здесь мы рассматриваем 1 является самым низким приоритетом.

Процесс	**Время прибытия**	**Время исполнения**	**Приоритет**	**Время обслуживания**
P0	0	5	1	0
P1	1	3	2	11
P2	2	8	1	14
P3	3	6	3	5

Время ожидания каждого процесса выглядит следующим образом –

Процесс	**Время ожидания**
P0	0 – 0 = 0
P1	11–1 = 10
P2	14–2 = 12
P3	5–3 = 2

Среднее время ожидания: (0 + 10 + 12 + 2) / 4 = 24/4 = 6

Самое короткое оставшееся время

Наименьшее оставшееся время (SRT) является преимущественной версией алгоритма SJN.

Процессор выделяется для задания, наиболее близкого к завершению, но ему может предшествовать более новое готовое задание с более коротким временем до завершения.

Невозможно реализовать в интерактивных системах, где необходимое время ЦП неизвестно.

Он часто используется в пакетных средах, где короткие задания должны отдавать предпочтение.

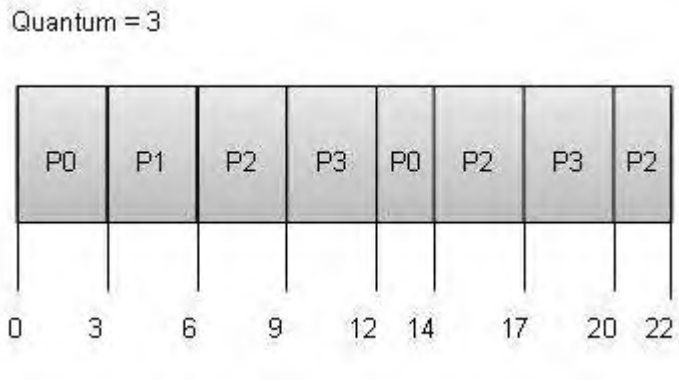
Круглый Робин Планирование

Round Robin – это алгоритм планирования вытесняющих процессов.

Каждому процессу предоставляется определенное время для выполнения, оно называется квантом.

Как только процесс выполняется в течение заданного периода времени, он прерывается, и другой процесс выполняется в течение заданного периода времени.

Переключение контекста используется для сохранения состояний вытесненных процессов.



Время ожидания каждого процесса выглядит следующим образом –

Процесс	**Время ожидания: Время обслуживания – Время прибытия**
P0	(0 – 0) + (12–3) = 9
P1	(3–1) = 2
P2	(6–2) + (14–9) + (20–17) = 12
P3	(9–3) + (17–12) = 11

Среднее время ожидания: (9 + 2 + 12 + 11) / 4 = 8,5

Планирование многоуровневых очередей

Многоуровневые очереди не являются независимым алгоритмом планирования. Они используют другие существующие алгоритмы для группировки и планирования заданий с общими характеристиками.

Для процессов с общими характеристиками поддерживается несколько очередей.

Каждая очередь может иметь свои собственные алгоритмы планирования.

Приоритеты присваиваются каждой очереди.

Например, задания, связанные с процессором, могут планироваться в одной очереди, а все задания, связанные с вводом / выводом, – в другой

очереди. Затем планировщик процессов поочередно выбирает задания из каждой очереди и назначает их ЦПУ на основе алгоритма, назначенного очереди.

Операционная система – многопоточность

Что такое тема?

Поток – это поток выполнения через код процесса, со своим собственным программным счетчиком, который отслеживает, какая инструкция должна выполняться в следующий раз, системные регистры, в которых хранятся текущие рабочие переменные, и стек, содержащий историю выполнения.

Поток делится со своими равноправными потоками такой информацией, как сегмент кода, сегмент данных и открытые файлы. Когда один поток изменяет элемент памяти сегмента кода, все остальные потоки видят это.

Нить также называется облегченным процессом. Потоки предоставляют способ повысить производительность приложений за счет параллелизма. Потоки представляют собой программный подход к повышению производительности операционной системы за счет сокращения служебных потоков, что эквивалентно классическому процессу.

Каждый поток принадлежит ровно одному процессу, и никакой поток не может существовать вне процесса. Каждый поток представляет отдельный поток управления. Потоки были успешно использованы при реализации сетевых серверов и веб-серверов. Они также обеспечивают подходящую основу для параллельного выполнения приложений на мультипроцессорах с общей памятью. На следующем рисунке показана работа однопоточного и многопоточного процесса.

Один против многопоточного процесса

Разница между процессом и потоком

SN	Процесс	Нить
1	Процесс тяжелый или ресурсоемкий.	Нить имеет малый вес и требует меньше ресурсов, чем процесс.
2	Процесс переключения требует взаимодействия с операционной системой.	Переключение потоков не требует взаимодействия с операционной системой.

3	В нескольких средах обработки каждый процесс выполняет один и тот же код, но имеет собственные ресурсы памяти и файлов.	Все потоки могут совместно использовать один и тот же набор открытых файлов, дочерних процессов.
4	Если один процесс заблокирован, то ни один другой процесс не может выполняться, пока первый процесс не будет разблокирован.	Пока один поток заблокирован и ожидает, второй поток в той же задаче может работать.
5	Несколько процессов без использования потоков используют больше ресурсов.	Многопоточные процессы используют меньше ресурсов.
6	В нескольких процессах каждый процесс работает независимо от других.	Один поток может читать, записывать или изменять данные другого потока.

Преимущества нити

- Потоки минимизируют время переключения контекста.
- Использование потоков обеспечивает параллелизм внутри процесса.
- Эффективное общение.
- Более экономично создавать и переключать контекстные потоки.
- Потоки позволяют использовать многопроцессорные архитектуры с большим масштабом и эффективностью.

Типы ниток

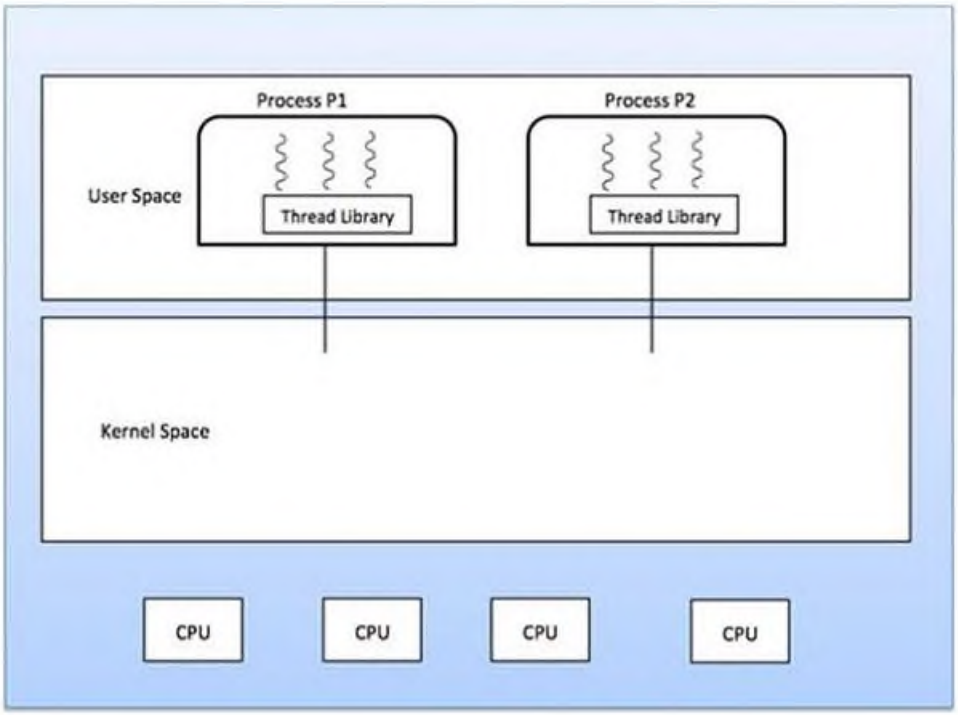
Потоки реализуются следующими двумя способами:

Потоки уровня пользователя – управляемые пользователем потоки.

Потоки уровня ядра – управляемые потоки операционной системы, работающие на ядре, ядре операционной системы.

Потоки уровня пользователя

В этом случае ядро управления потоками не знает о существовании потоков. Библиотека потоков содержит код для создания и уничтожения потоков, для передачи сообщений и данных между потоками, для планирования выполнения потоков и для сохранения и восстановления контекстов потоков. Приложение запускается с одного потока.



Преимущества

Переключение потоков не требует привилегий режима ядра.

Поток пользовательского уровня может работать в любой операционной системе.

Планирование может зависеть от приложения в потоке уровня пользователя.

Потоки пользовательского уровня быстро создаются и управляются.

Недостатки

В типичной операционной системе большинство системных вызовов блокируются.

Многопоточное приложение не может использовать преимущества многопроцессорной обработки.

Потоки уровня ядра

В этом случае управление потоками осуществляется ядром. В области приложения нет кода управления потоками. Потоки ядра поддерживаются непосредственно операционной системой. Любое приложение может быть запрограммировано на многопоточность. Все потоки в приложении поддерживаются в рамках одного процесса.

Ядро поддерживает контекстную информацию для процесса в целом и для отдельных потоков внутри процесса. Планирование Ядром осуществляется на основе потоков. Ядро выполняет создание потоков, планирование и управление в пространстве ядра. Потоки ядра обычно медленнее создаются и управляются, чем пользовательские потоки.

Преимущества

Ядро может одновременно планировать несколько потоков из одного процесса на несколько процессов.

Если один поток в процессе заблокирован, ядро может запланировать другой поток того же процесса.

Сами подпрограммы ядра могут быть многопоточными.

Недостатки

Потоки ядра обычно медленнее создаются и управляются, чем пользовательские потоки.

Передача управления из одного потока в другой в рамках одного и того же процесса требует переключения режима в ядро.

Многопоточные модели

Некоторые операционные системы предоставляют объединенный поток уровня пользователя и средство потока уровня ядра. Solaris является хорошим примером такого комбинированного подхода. В комбинированной системе несколько потоков в одном приложении могут работать параллельно на нескольких процессорах, и блокирующий системный вызов не должен блокировать весь процесс. Многопоточные модели бывают трех типов

Много ко многим отношениям.

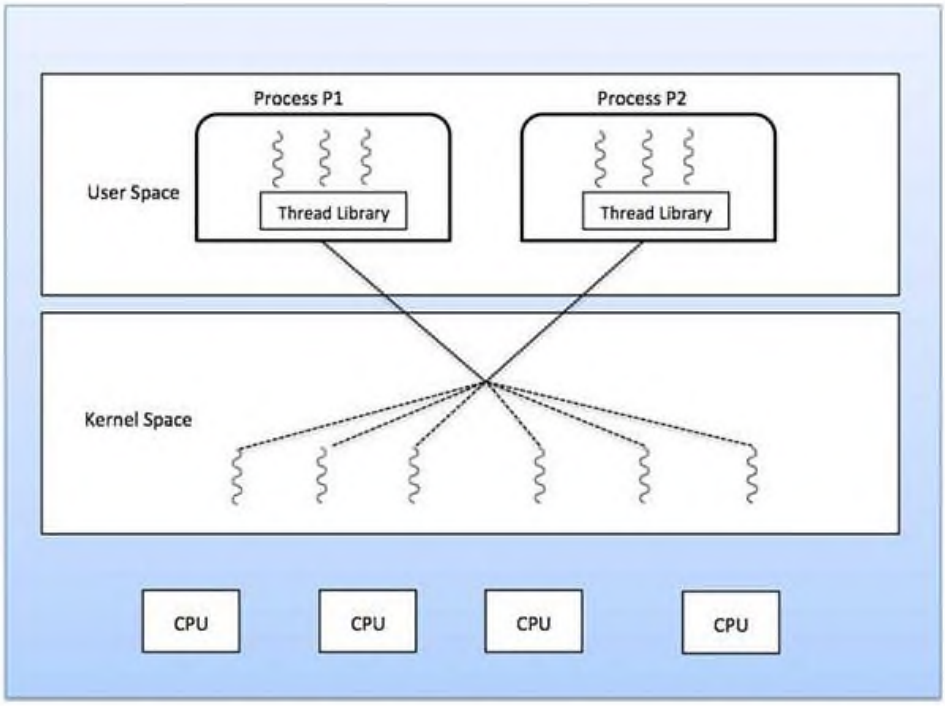
Много к одному отношения.

Отношения один к одному.

Модель много ко многим

Модель «многие ко многим» мультиплексирует любое количество пользовательских потоков на равное или меньшее количество потоков ядра.

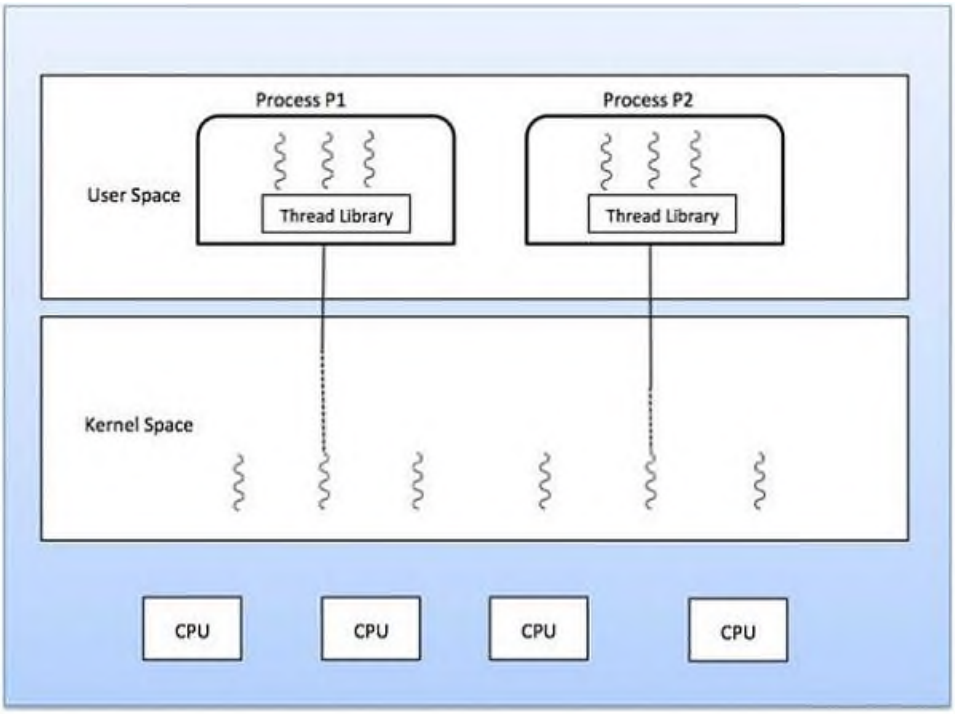
На следующей диаграмме показана модель многопоточности, при которой 6 потоков уровня пользователя мультиплексируются с 6 потоками уровня ядра. В этой модели разработчики могут создавать столько пользовательских потоков, сколько необходимо, и соответствующие потоки ядра могут работать параллельно на многопроцессорной машине. Эта модель обеспечивает наилучшую точность при параллельности, и когда поток выполняет системный вызов блокировки, ядро может запланировать выполнение другого потока.



Много к одной модели

Модель «многие к одному» отображает множество потоков уровня пользователя в один поток уровня ядра. Управление потоками осуществляется в пространстве пользователя библиотекой потоков. Когда поток выполняет системный вызов блокировки, весь процесс будет заблокирован. Только один поток может одновременно обращаться к Ядру, поэтому несколько потоков не могут работать параллельно на многопроцессорных системах.

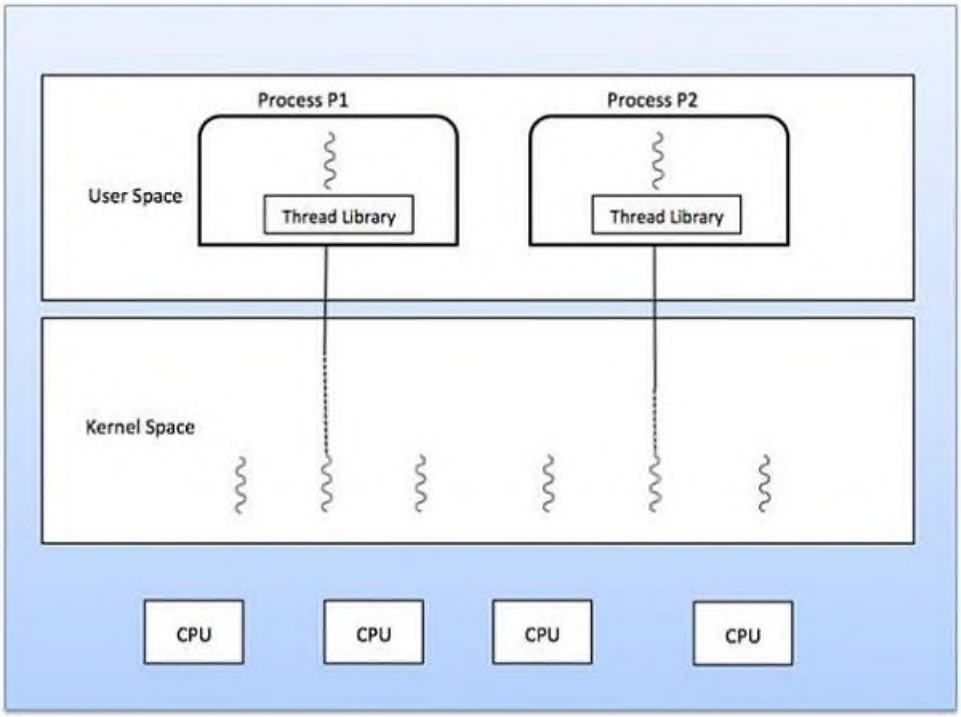
Если библиотеки потоков пользовательского уровня реализованы в операционной системе таким образом, что система их не поддерживает, то потоки ядра используют режимы отношения «многие к одному».



Модель один на один

Существует взаимно-однозначное отношение потока уровня пользователя к потоку уровня ядра. Эта модель обеспечивает больше параллелизма, чем модель «многие к одному». Это также позволяет другому потоку запускаться, когда поток выполняет блокирующий системный вызов. Он поддерживает несколько потоков для параллельного выполнения на микропроцессорах.

Недостатком этой модели является то, что для создания пользовательского потока требуется соответствующий поток ядра. OS / 2, Windows NT и Windows 2000 используют модель отношений один к одному.



Разница между пользовательским уровнем и потоком на уровне ядра

N	Потоки уровня пользователя	Нить на уровне ядра
1	Потоки пользовательского уровня быстрее создаются и управляются.	Потоки уровня ядра медленнее создаются и управляются.
2	Реализация осуществляется библиотекой потоков на уровне пользователя.	Операционная система поддерживает создание потоков ядра.
3	Поток пользовательского уровня является общим и может работать в любой операционной системе.	Поток уровня ядра специфичен для операционной системы.
4	Многопоточные приложения не могут использовать преимущества многопроцессорной обработки.	Сами подпрограммы ядра могут быть многопоточными.

Операционная система – Управление памятью

Управление памятью – это функциональность операционной системы, которая обрабатывает или управляет основной памятью и перемещает процессы назад и вперед между основной памятью и диском во время выполнения. Управление памятью отслеживает каждую область памяти, независимо от того, выделено ли это для какого-либо процесса или свободно. Он проверяет, сколько памяти должно быть выделено процессам. Он решает, какой процесс получит память и в какое время. Он отслеживает всякий раз, когда какая-то память освобождается или выделяется, и соответственно обновляет статус.

Этот материал научит вас основным понятиям, связанным с управлением памятью.

Адресное пространство процесса

Адресное пространство процесса – это набор логических адресов, на которые процесс ссылается в своем коде. Например, когда используется 32-битная адресация, адреса могут варьироваться от 0 до 0x7fffffff; то есть 2 ^ 31 возможных чисел, для общего теоретического размера 2 гигабайта.

Операционная система заботится о сопоставлении логических адресов с физическими адресами во время выделения памяти программе. В программе используются три типа адресов до и после выделения памяти:

Символьные адреса

Адреса, используемые в исходном коде. Имена переменных, константы и метки команд являются основными элементами символьного адресного пространства.

Относительные адреса

Во время компиляции компилятор преобразует символические адреса в относительные адреса.

Физические адреса

Загрузчик генерирует эти адреса в тот момент, когда программа загружается в основную память.

Виртуальные и физические адреса одинаковы в схемах привязки адресов во время компиляции и во время загрузки. Виртуальные и физические адреса различаются по схеме привязки адресов во время выполнения.

Множество всех логических адресов, сгенерированных программой, называется логическим адресным пространством. Набор всех физических адресов, соответствующих этим логическим адресам, называется физическим адресным пространством.

Отображение времени выполнения с виртуального на физический адрес выполняется блоком управления памятью (MMU), который является аппаратным устройством. MMU использует следующий механизм для преобразования виртуального адреса в физический адрес.

Значение в базовом регистре добавляется к каждому адресу, сгенерированному пользовательским процессом, который обрабатывается как смещение во время его отправки в память. Например, если значение базового регистра равно 10000, то попытка пользователя использовать адрес 100 местоположения будет динамически перераспределена в местоположение 10100.

Пользовательская программа имеет дело с виртуальными адресами; он никогда не видит реальные физические адреса.

Статическая и динамическая загрузка

Выбор между статической или динамической загрузкой должен быть сделан во время разработки компьютерной программы. Если вам нужно загружать свою программу статически, то во время компиляции все программы будут скомпилированы и связаны без какой-либо внешней зависимости от программы или модуля. Компоновщик объединяет объектную программу с другими необходимыми объектными модулями в абсолютную программу, которая также включает в себя логические адреса.

Если вы пишете динамически загружаемую программу, то ваш компилятор скомпилирует программу, и для всех модулей, которые вы хотите включить динамически, будут предоставлены только ссылки, а остальная часть работы будет выполнена во время выполнения.

Во время загрузки при статической загрузке абсолютная программа (и данные) загружаются в память для запуска выполнения.

Если вы используете динамическую загрузку, динамические процедуры библиотеки хранятся на диске в перемещаемой форме и загружаются в память только тогда, когда они необходимы программе.

Статическое и динамическое связывание

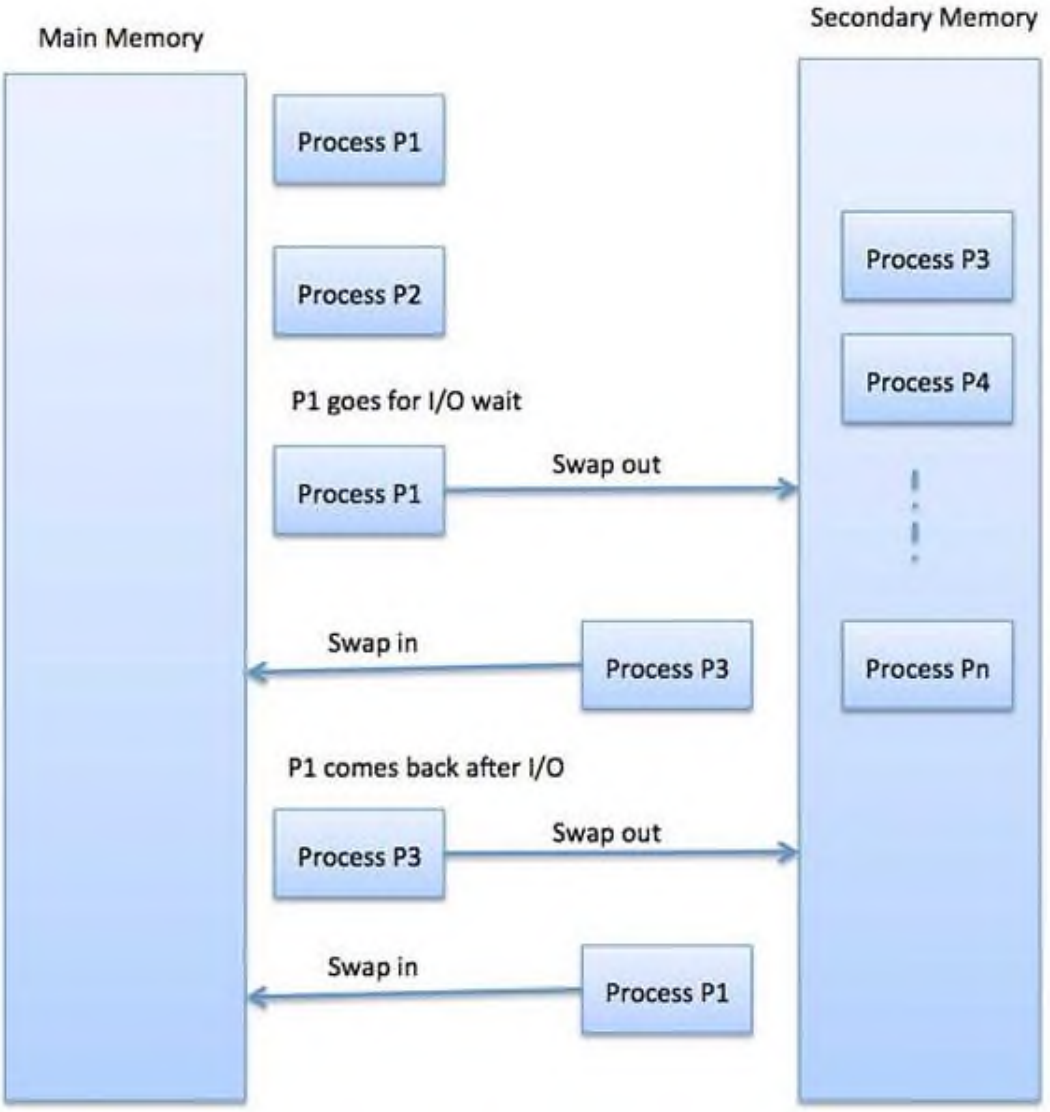
Как объяснено выше, когда используется статическое связывание, компоновщик объединяет все другие модули, необходимые программе, в одну исполняемую программу, чтобы избежать какой-либо зависимости во время выполнения.

Когда используется динамическое связывание, нет необходимости связывать реальный модуль или библиотеку с программой, скорее, ссылка на динамический модуль предоставляется во время компиляции и связывания. Динамические библиотеки ссылок (DLL) в Windows и общие объекты в Unix являются хорошими примерами динамических библиотек.

Перестановка

Обмен – это механизм, в котором процесс может быть временно выгружен из основной памяти (или перемещен) во вторичное хранилище (диск) и сделать эту память доступной для других процессов. Через некоторое время система переключает процесс обратно со вторичного хранилища на основную память.

Хотя на процесс подкачки обычно влияет производительность, но он помогает в параллельном запуске нескольких больших процессов, и именно поэтому своппинг также известен как метод сжатия памяти.



Общее время, необходимое для процесса подкачки, включает время, необходимое для перемещения всего процесса на вторичный диск, а затем для копирования процесса обратно в память, а также время, необходимое процессу для восстановления основной памяти.

Предположим, что пользовательский процесс имеет размер 2048 КБ, и на стандартном жестком диске, где произойдет обмен, скорость передачи данных составляет около 1 МБ в секунду. Фактический перенос процесса 1000K в или из памяти займет

```
2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds
```

Теперь, учитывая время на вход и выход, потребуется 4000 миллисекунд плюс другие издержки, когда процесс конкурирует за восстановление основной памяти.

Выделение памяти

Основная память обычно имеет два раздела –

Недостаточно памяти – Операционная система находится в этой памяти.

High Memory – пользовательские процессы хранятся в верхней памяти.

Распределение по одному разделу

В этом типе распределения схема регистров перемещения используется для защиты пользовательских процессов друг от друга, а также от изменения кода и данных операционной системы. Регистр перемещения содержит значение наименьшего физического адреса, тогда как регистр ограничения содержит диапазон логических адресов. Каждый логический адрес должен быть меньше регистра ограничения.

Распределение нескольких разделов

При таком типе распределения основная память делится на несколько разделов фиксированного размера, где каждый раздел должен содержать только один процесс. Когда раздел свободен, процесс выбирается из входной очереди и загружается в свободный раздел. Когда процесс завершается, раздел становится доступным для другого процесса.

Фрагментация

Поскольку процессы загружаются и удаляются из памяти, свободное пространство памяти разбивается на маленькие кусочки. Иногда бывает, что процессы не могут быть выделены блокам памяти, учитывая их небольшой размер, и блоки памяти остаются неиспользованными. Эта проблема известна как фрагментация.

Фрагментация бывает двух типов –

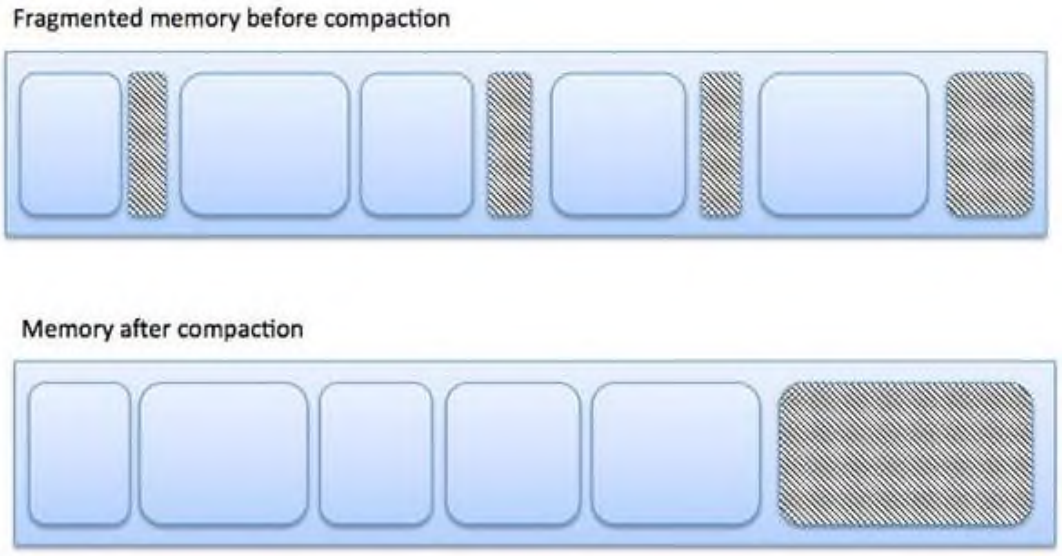
Внешняя фрагментация

Всего пространства памяти достаточно для удовлетворения запроса или для размещения в нем процесса, но он не является непрерывным, поэтому его нельзя использовать.

Внутренняя фрагментация

Блок памяти, назначенный процессу, больше. Некоторая часть памяти остается неиспользованной, так как она не может быть использована другим процессом.

На следующей диаграмме показано, как фрагментация может привести к потере памяти, а техника сжатия может использоваться для создания большей свободной памяти из фрагментированной памяти.



Внешняя фрагментация может быть уменьшена путем сжатия или перемешивания содержимого памяти, чтобы поместить всю свободную память вместе в один большой блок. Чтобы сделать уплотнение возможным, перемещение должно быть динамическим.

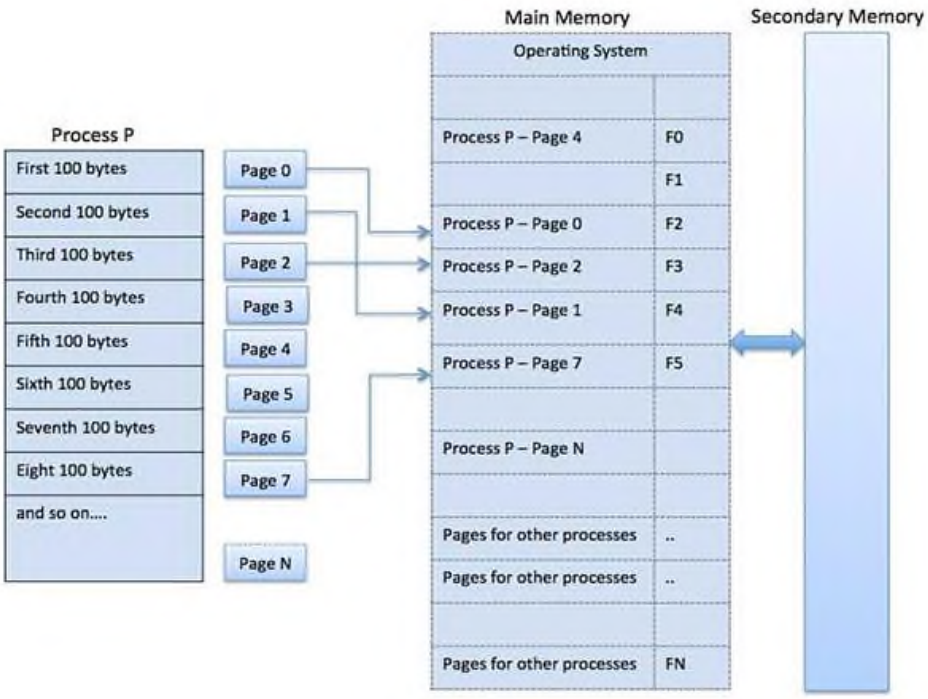
Внутренняя фрагментация может быть уменьшена путем эффективного назначения самого маленького раздела, но достаточно большого для процесса.

Paging

Компьютер может адресовать больше памяти, чем физически установлено в системе. Эта дополнительная память фактически называется виртуальной памятью, и это раздел жесткого диска, настроенный для эмуляции оперативной памяти компьютера. Техника подкачки играет важную роль в реализации виртуальной памяти.

Пейджинг – это метод управления памятью, при котором адресное пространство процесса разбивается на блоки одинакового размера, называемые страницами (размер равен степени 2, от 512 до 8192 байтов). Размер процесса измеряется количеством страниц.

Точно так же основная память делится на небольшие блоки фиксированного размера (физической) памяти, называемые кадрами, и размер кадра остается таким же, как и размер страницы, чтобы оптимально использовать основную память и избежать внешней фрагментации.



Перевод адреса

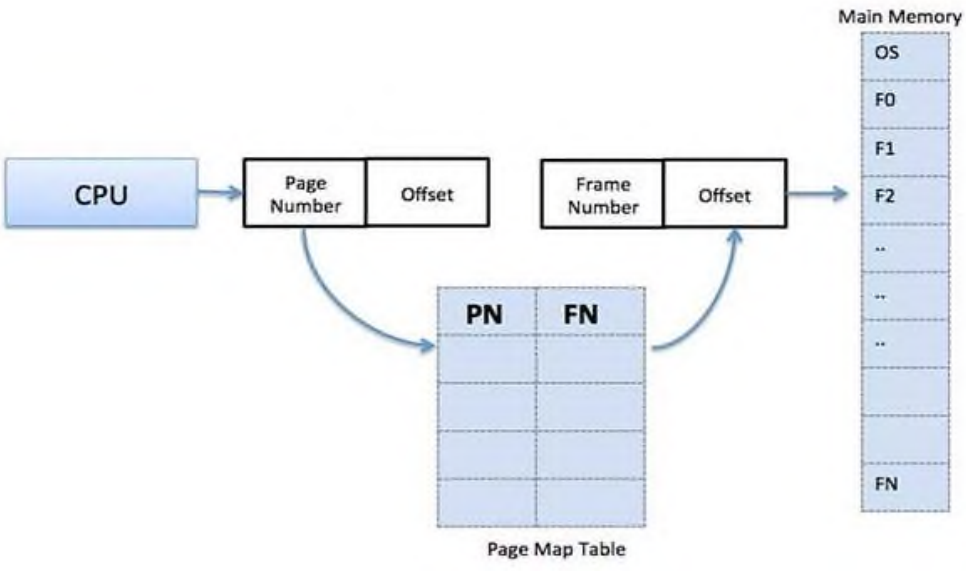
Адрес страницы называется логическим адресом и представлен номером страницы и смещением.

```
Logical Address = Page number + page offset
```

Адрес кадра называется физическим адресом и представлен номером кадра и смещением.

```
Physical Address = Frame number + page offset
```

Структура данных, называемая таблицей карты страниц, используется для отслеживания связи между страницей процесса и кадром в физической памяти.



Когда система выделяет фрейм для любой страницы, она преобразует этот логический адрес в физический адрес и создает запись в таблице страниц, которая будет использоваться при выполнении программы.

Когда процесс должен быть выполнен, его соответствующие страницы загружаются в любые доступные кадры памяти. Предположим, у вас есть программа объемом 8 КБ, но ваша память может вместить только 5 КБ в данный момент времени, тогда концепция пейджинга войдет в картину. Когда на компьютере заканчивается ОЗУ, операционная система (ОС) перемещает незанятые или ненужные страницы памяти во вторичную память, чтобы освободить ОЗУ для других процессов, и возвращает их, когда это необходимо программе.

Этот процесс продолжается в течение всего выполнения программы, где ОС продолжает удалять пустые страницы из основной памяти и записывать их во вторичную память и возвращать их, когда это требуется программой.

Преимущества и недостатки пейджинга

Вот список преимуществ и недостатков подкачки –

Пейджинг уменьшает внешнюю фрагментацию, но все еще страдает от внутренней фрагментации.

Пейджинг прост в реализации и считается эффективным методом управления памятью.

Благодаря одинаковому размеру страниц и фреймов, обмен становится очень простым.

Таблица страниц требует дополнительного места в памяти, поэтому может не подойти для системы с небольшим объемом ОЗУ.

Пейджинг уменьшает внешнюю фрагментацию, но все еще страдает от внутренней фрагментации.

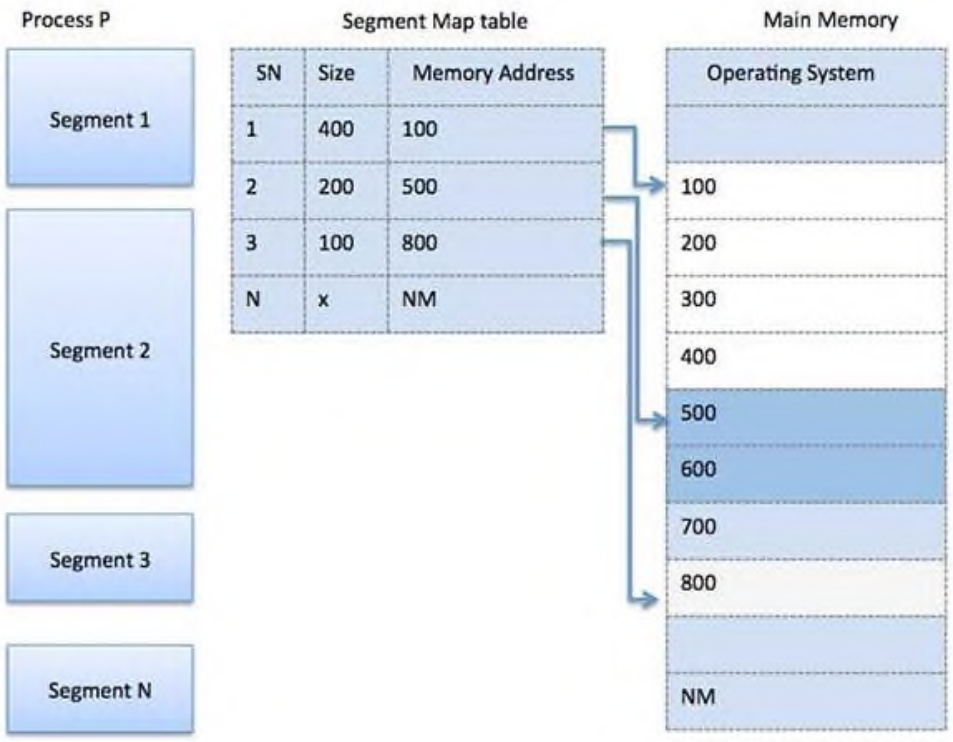
Сегментация

Сегментация – это метод управления памятью, при котором каждое задание делится на несколько сегментов разных размеров, по одному для каждого модуля, который содержит фрагменты, выполняющие связанные функции. Каждый сегмент на самом деле отличается логическим адресным пространством программы.

Когда процесс должен быть выполнен, его соответствующая сегментация загружается в несмежную память, хотя каждый сегмент загружается в непрерывный блок доступной памяти.

Управление памятью сегментирования очень похоже на пейджинг, но здесь сегменты имеют переменную длину, где, как и на страницах подкачки, имеют фиксированный размер.

Программный сегмент содержит основную функцию программы, вспомогательные функции, структуры данных и т. Д. Операционная система поддерживает таблицу карты сегментов для каждого процесса и список свободных блоков памяти, а также номера сегментов, их размер и соответствующие области памяти в основной памяти. Для каждого сегмента в таблице хранится начальный адрес сегмента и длина сегмента. Ссылка на ячейку памяти включает в себя значение, которое идентифицирует сегмент и смещение.



Операционная система – виртуальная память

Компьютер может адресовать больше памяти, чем физически установлено в системе. Эта дополнительная память фактически называется виртуальной памятью и представляет собой раздел жесткого диска, настроенный для эмуляции оперативной памяти компьютера.

Основным видимым преимуществом этой схемы является то, что программы могут быть больше, чем физическая память. Виртуальная память служит двум целям. Во-первых, это позволяет нам расширить использование физической памяти с помощью диска. Во-вторых, это позволяет нам иметь защиту памяти, потому что каждый виртуальный адрес транслируется в физический адрес.

Ниже приведены ситуации, когда не требуется полная загрузка всей программы в основную память.

Написанные пользователем процедуры обработки ошибок используются только тогда, когда в данных или вычислениях произошла ошибка.

Некоторые параметры и функции программы могут использоваться редко.

Многим таблицам назначается фиксированный объем адресного пространства, хотя фактически используется только небольшой объем таблицы.

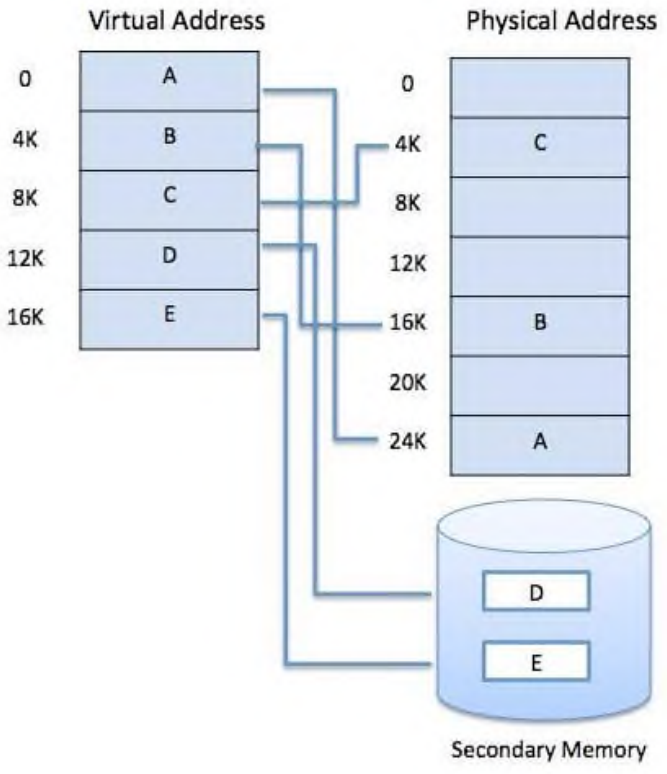
Возможность выполнения программы, которая только частично находится в памяти, может противостоять многим преимуществам.

Для загрузки или замены каждой пользовательской программы в память потребуется меньшее количество операций ввода-вывода.

Программа больше не будет ограничена объемом доступной физической памяти.

Каждая пользовательская программа может занимать меньше физической памяти, одновременно может выполняться больше программ с соответствующим увеличением загрузки ЦП и пропускной способности.

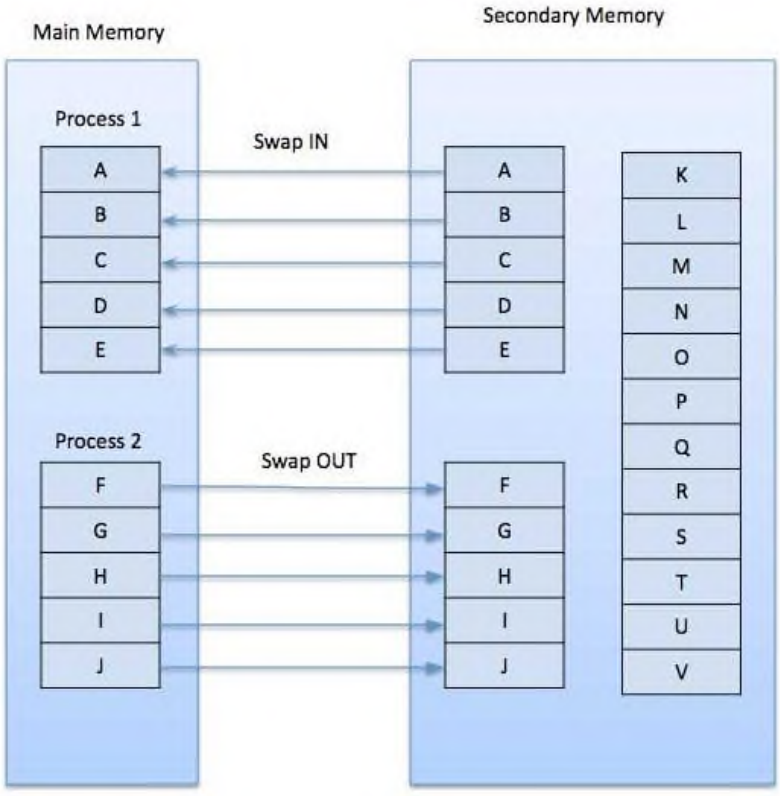
Современные микропроцессоры, предназначенные для общего назначения, блок управления памятью или MMU, встроен в аппаратное обеспечение. Работа MMU заключается в преобразовании виртуальных адресов в физические адреса. Основной пример приведен ниже –



Виртуальная память обычно реализуется подкачкой по требованию. Это также может быть реализовано в системе сегментации. Сегментация спроса также может использоваться для предоставления виртуальной памяти.

Пейджинг по требованию

Система подкачки по требованию очень похожа на систему подкачки с подкачкой, где процессы находятся во вторичной памяти, а страницы загружаются только по требованию, а не заранее. Когда происходит переключение контекста, операционная система не копирует ни одну из страниц старой программы на диск или любую страницу новой программы в основную память. Вместо этого она просто начинает выполнение новой программы после загрузки первой страницы и извлекает страницы программы, на которые они ссылаются.



При выполнении программы, если программа ссылается на страницу, которая недоступна в основной памяти, потому что она была заменена недавно, процессор обрабатывает эту недействительную ссылку на память как ошибку страницы и передает управление от программы операционной системе к потребовать страницу обратно в память.

Преимущества

Ниже приведены преимущества пейджинга по требованию:

- Большая виртуальная память.
- Более эффективное использование памяти.
- Нет ограничений по степени мультипрограммирования.

Недостатки

Количество таблиц и объем служебных данных процессора для обработки прерываний страниц больше, чем в случае простых методов постраничного управления.

Алгоритм замены страницы

Алгоритмы замены страниц – это методы, с помощью которых операционная система решает, какие страницы памяти следует поменять, записать на диск, когда необходимо выделить страницу памяти. Пейджинг происходит всякий раз, когда происходит сбой страницы, и свободная страница не может быть использована для целей распределения, так как страницы недоступны или количество свободных страниц меньше требуемых страниц.

Когда на страницу, которая была выбрана для замены и была выгружена, снова ссылаются, она должна считывать данные с диска, и это требует завершения ввода-вывода. Этот процесс определяет качество алгоритма замены страницы: чем меньше время ожидания для вставки страниц, тем лучше алгоритм.

Алгоритм замены страниц просматривает ограниченную информацию о доступе к страницам, предоставляемым аппаратным обеспечением, и пытается выбрать, какие страницы следует заменить, чтобы минимизировать общее количество пропусков страниц, в то же время сопоставляя его с затратами на первичное хранение и процессорное время алгоритма. сам. Есть много разных алгоритмов замены страниц. Мы оцениваем алгоритм, выполняя его для определенной строки ссылки на память и вычисляя количество ошибок страниц,

Ссылочная строка

Строка ссылок в памяти называется ссылочной строкой. Строки ссылок генерируются искусственно или путем отслеживания заданной системы и записи адреса каждой ссылки в памяти. Последний выбор дает большое количество данных, где мы отмечаем две вещи.

Для данного размера страницы нам нужно учитывать только номер страницы, а не весь адрес.

Если у нас есть ссылка на страницу p, то любые последующие ссылки на страницу p никогда не вызовут ошибку страницы. Страница p будет в памяти после первой ссылки; следующие ссылки не будут ошибаться.

Например, рассмотрим следующую последовательность адресов – 123,215,600,1234,76,96

Если размер страницы равен 100, тогда строка ссылки равна 1,2,6,12,0,0

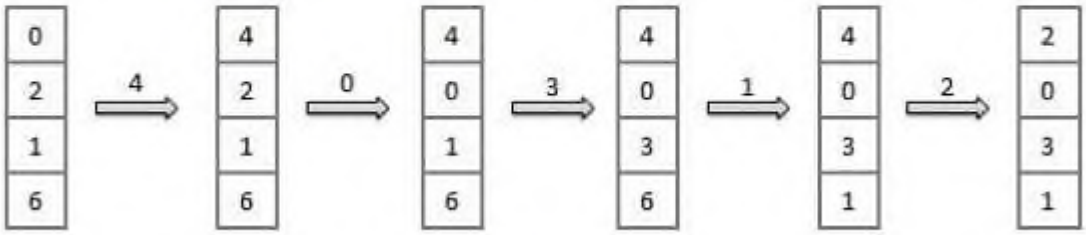
Алгоритм First In First Out (FIFO)

Самая старая страница в основной памяти – та, которая будет выбрана для замены.

Легко реализовать, вести список, заменять страницы из хвоста и добавлять новые страницы в заголовок.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x x



Fault Rate = 9 / 12 = 0.75

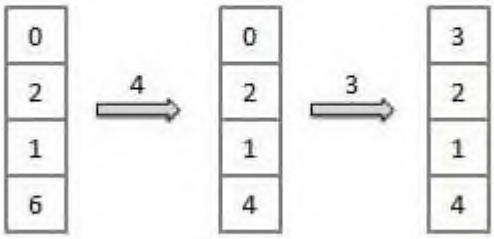
Оптимальный алгоритм страницы

Оптимальный алгоритм замены страниц имеет самую низкую частоту отказов страниц среди всех алгоритмов. Существует оптимальный алгоритм замены страниц, который называется OPT или MIN.

Замените страницу, которая не будет использоваться в течение самого длительного периода времени. Используйте время, когда страница будет использоваться.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x



Fault Rate = 6 / 12 = 0.50

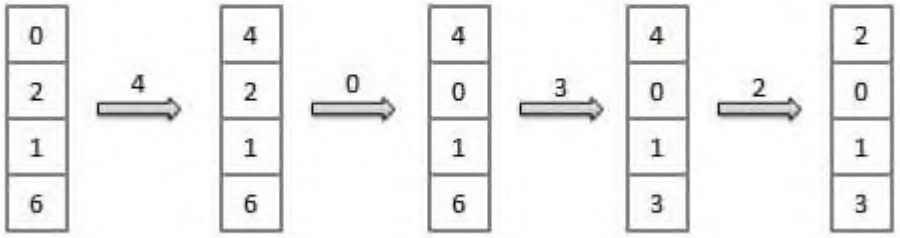
Наименее недавно использованный (LRU) алгоритм

Страница, которая долгое время не использовалась в основной памяти, будет выбрана для замены.

Легко реализовать, вести список, заменять страницы, оглядываясь в прошлое.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

Алгоритм буферизации страниц

Чтобы быстро начать процесс, сохраняйте пул бесплатных фреймов.

В случае ошибки страницы выберите страницу, которую необходимо заменить.

Запишите новую страницу в рамку свободного пула, отметьте таблицу страниц и перезапустите процесс.

Теперь запишите грязную страницу с диска и поместите фрейм, содержащий замененную страницу, в свободный пул.

Наименее часто используемый (LFU) алгоритм

Страница с наименьшим количеством считается той, которая будет выбрана для замены.

Этот алгоритм страдает от ситуации, в которой страница интенсивно используется на начальном этапе процесса, но затем никогда не используется снова.

Наиболее часто используемый (MFU) алгоритм

Этот алгоритм основан на аргументе, что страница с наименьшим количеством, вероятно, только что была введена и еще не использовалась.

Операционная система – Оборудование ввода / вывода

Одной из важных задач операционной системы является управление различными устройствами ввода-вывода, включая мышь, клавиатуры, сенсорную панель, дисководы, адаптеры дисплея, USB-устройства, битовый экран, светодиод, аналого-цифровой преобразователь, вкл / выкл. выключатель, сетевые соединения, аудио ввод / вывод, принтеры и т. д.

Система ввода-вывода должна принимать запрос ввода-вывода приложения и отправлять его на физическое устройство, затем принимать любой ответ, возвращаемый устройством, и отправлять его приложению. Устройства ввода / вывода можно разделить на две категории –

Блочные устройства – блочное устройство, с которым драйвер связывается, отправляя целые блоки данных. Например, жесткие диски, USB-камеры, Disk-On-Key и т. Д.

Символьные устройства – это устройство, с которым драйвер общается, отправляя и получая отдельные символы (байты, октеты). Например, последовательные порты, параллельные порты, звуковые карты и т. д.

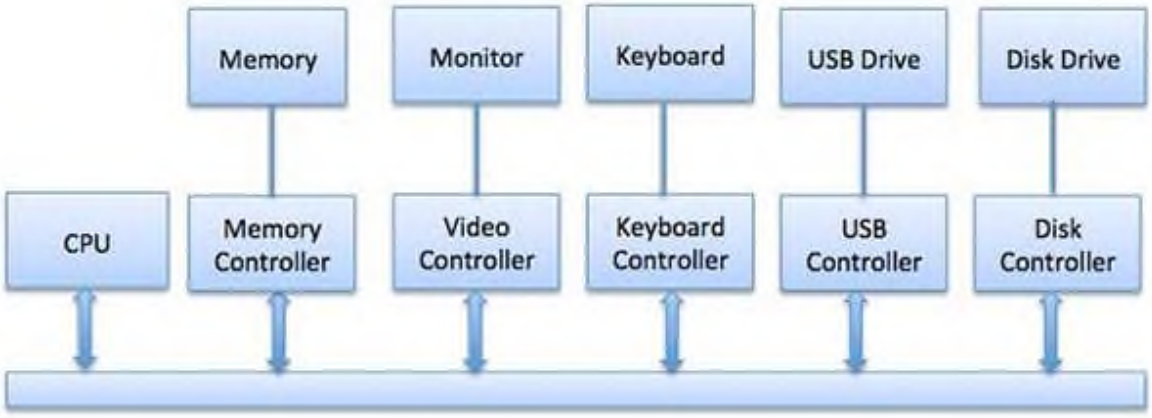
Контроллеры устройств

Драйверы устройств – это программные модули, которые можно подключить к ОС для работы с конкретным устройством. Операционная система получает помощь от драйверов устройств для обработки всех устройств ввода-вывода.

Контроллер устройства работает как интерфейс между устройством и драйвером устройства. Блоки ввода / вывода (клавиатура, мышь, принтер и т. Д.) Обычно состоят из механического компонента и электронного компонента, где электронный компонент называется контроллером устройства.

Для каждого устройства всегда есть контроллер устройства и драйвер устройства для связи с операционными системами. Контроллер устройства может обрабатывать несколько устройств. В качестве интерфейса его основной задачей является преобразование последовательного потока битов в блок байтов, выполнение коррекции ошибок по мере необходимости.

Любое устройство, подключенное к компьютеру, подключается через разъем и розетку, а разъем подключается к контроллеру устройства. Ниже приведена модель подключения ЦП, памяти, контроллеров и устройств ввода-вывода, в которой все ЦП и контроллеры устройств используют общую шину для связи.



Синхронный и асинхронный ввод-вывод

Синхронный ввод / вывод – в этой схеме выполнение ЦП ожидает, пока продолжается ввод / вывод

Асинхронный ввод-вывод – ввод-вывод выполняется одновременно с выполнением процессора

Связь с устройствами ввода / вывода

Процессор должен иметь возможность передавать информацию на устройство ввода-вывода и обратно. Существует три подхода для связи с процессором и устройством.

Специальная инструкция ввода / вывода

Отображение в памяти ввода-вывода

Прямой доступ к памяти (DMA)

Специальная инструкция ввода / вывода

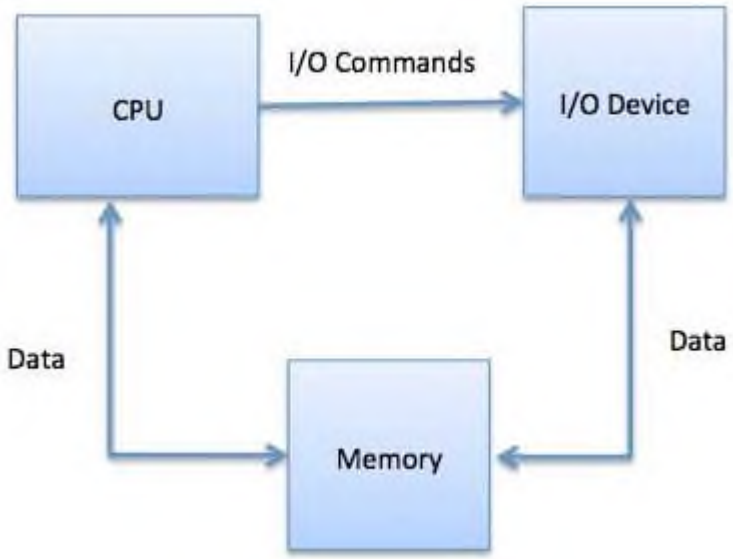
При этом используются инструкции процессора, специально предназначенные для управления устройствами ввода-вывода. Эти инструкции обычно позволяют отправлять данные на устройство ввода-вывода или считывать данные с устройства ввода-вывода.

Отображение в памяти ввода-вывода

При использовании сопоставленного с памятью ввода-вывода одно и то же адресное пространство совместно используется устройствами памяти и ввода-вывода. Устройство подключено напрямую к определенным местам в основной памяти, так что устройство ввода-вывода может передавать блок данных в / из памяти без прохождения через CPU.

Отображение в памяти ввода-вывода

При использовании отображения ввода-вывода в память ОС выделяет буфер в памяти и сообщает устройству ввода-вывода использовать этот буфер для отправки данных в ЦП. Устройство ввода-вывода работает асинхронно с ЦП, прерывает ЦП после завершения.



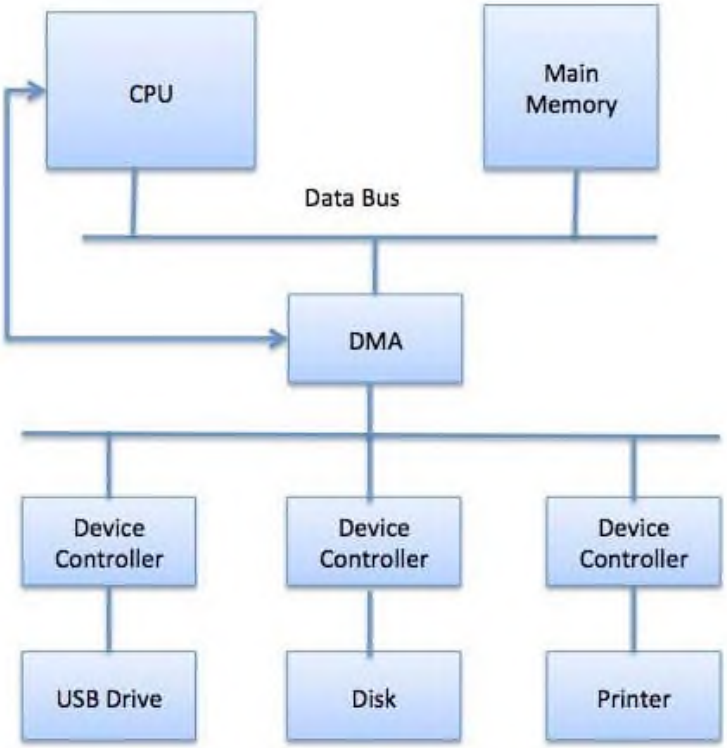
Преимущество этого метода состоит в том, что каждая команда, которая может получить доступ к памяти, может использоваться для управления устройством ввода-вывода. Операции ввода-вывода с отображением в память используются для большинства высокоскоростных устройств ввода-вывода, таких как диски, интерфейсы связи.

Прямой доступ к памяти (DMA)

Медленные устройства, такие как клавиатуры, будут генерировать прерывание для основного процессора после передачи каждого байта. Если бы быстрое устройство, такое как диск, генерировало прерывание для каждого байта, операционная система потратила бы большую часть своего времени на обработку этих прерываний. Таким образом, типичный компьютер использует аппаратное обеспечение прямого доступа к памяти (DMA), чтобы уменьшить эти издержки.

Прямой доступ к памяти (DMA) означает, что ЦП предоставляет полномочия модуля ввода / вывода для чтения или записи в память без участия. Модуль DMA сам управляет обменом данными между основной памятью и устройством ввода-вывода. CPU задействуется только в начале и в конце передачи и прерывается только после того, как весь блок был передан.

Для прямого доступа к памяти требуется специальное оборудование, называемое контроллером DMA (DMAC), которое управляет передачей данных и разрешает доступ к системной шине. Контроллеры запрограммированы с указателями источника и назначения (где для чтения / записи данных), счетчиками для отслеживания количества переданных байтов и настройками, которые включают в себя ввод / вывод и типы памяти, прерывания и состояния для циклов ЦП.



Операционная система использует оборудование DMA следующим образом:

шаг	Описание
1	Драйвер устройства получает указание перенести данные с диска на адрес буфера X.
2	Затем драйвер устройства дает команду контроллеру диска передать данные в буфер.
3	Контроллер диска начинает передачу DMA.
4	Контроллер диска отправляет каждый байт контроллеру DMA.
5	Контроллер DMA передает байты в буфер, увеличивает адрес памяти, уменьшает счетчик C, пока C не станет равным нулю.
6	Когда C становится равным нулю, DMA прерывает CPU, чтобы завершить передачу сигнала.

Опрос против прерывания ввода / вывода

Компьютер должен иметь способ обнаружения поступления любого типа ввода. Это может произойти двумя способами, известными как опрос и прерывание. Оба эти метода позволяют процессору иметь дело с событиями,

которые могут произойти в любое время и которые не связаны с процессом, который он выполняет в данный момент.

Опрос ввода / вывода

Опрос – это самый простой способ взаимодействия устройства ввода-вывода с процессором. Процесс периодической проверки состояния устройства, чтобы узнать, настало ли время для следующей операции ввода-вывода, называется опросом. Устройство ввода-вывода просто помещает информацию в регистр состояния, и процессор должен прийти и получить информацию.

В большинстве случаев устройства не будут требовать внимания, и когда это произойдет, придется подождать, пока их не запросит следующая программа опроса. Это неэффективный метод, и большая часть времени процессоров тратится на ненужные опросы.

Сравните этот метод с учителем, постоянно спрашивающим каждого ученика в классе, один за другим, нужна ли им помощь. Очевидно, что более эффективный метод будет для ученика информировать учителя, когда они нуждаются в помощи.

Прерывает ввод / вывод

Альтернативная схема для работы с вводом / выводом – это метод, управляемый прерываниями. Прерывание – это сигнал для микропроцессора от устройства, которое требует внимания.

Контроллер устройства помещает сигнал прерывания на шину, когда ему требуется внимание ЦП, когда ЦП получает прерывание. Он сохраняет свое текущее состояние и вызывает соответствующий обработчик прерываний, используя вектор прерывания (адреса подпрограмм ОС для обработки различных событий). Когда с устройством прерывания обработано, ЦПУ продолжает выполнять свою первоначальную задачу, как если бы оно никогда не прерывалось.

Операционная система – Программное обеспечение ввода / вывода

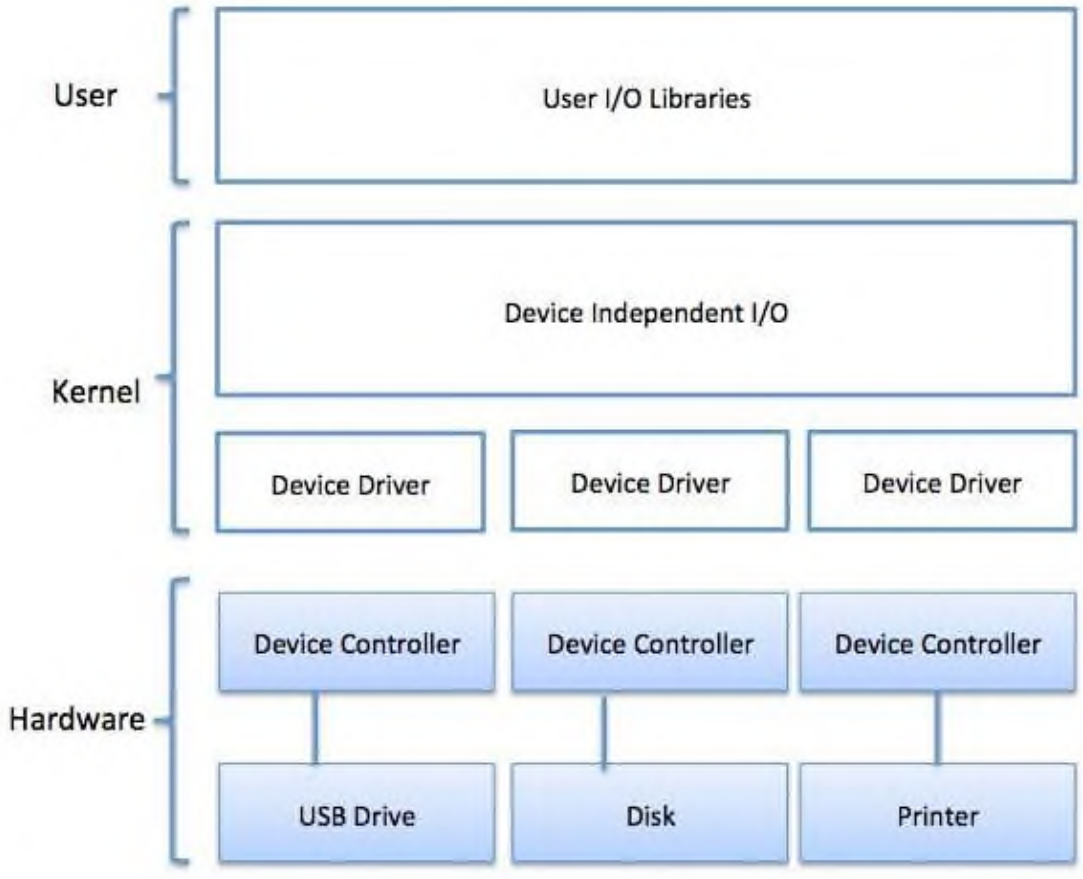
Программное обеспечение ввода / вывода часто организовано на следующих уровнях:

Библиотеки уровня пользователя – это обеспечивает простой интерфейс для пользовательской программы для выполнения ввода и вывода. Например, stdio – это библиотека, предоставляемая языками программирования C и C ++.

Модули уровня ядра – это обеспечивает драйвер устройства для взаимодействия с контроллером устройства и независимыми от устройства модулями ввода / вывода, используемыми драйверами устройства.

Аппаратное обеспечение – этот уровень включает в себя фактический аппаратный и аппаратный контроллер, который взаимодействует с драйверами устройства и обеспечивает работу оборудования.

Ключевой концепцией в разработке программного обеспечения ввода / вывода является то, что оно должно быть независимым от устройства, где должна быть возможность писать программы, которые могут получить доступ к любому устройству ввода / вывода без необходимости предварительно указывать устройство. Например, программа, которая считывает файл как ввод, должна иметь возможность считывать файл с дискеты, жесткого диска или компакт-диска без необходимости изменять программу для каждого отдельного устройства.



Драйверы устройств

Драйверы устройств – это программные модули, которые можно подключить к ОС для работы с конкретным устройством. Операционная система получает помощь от драйверов устройств для обработки всех устройств ввода-вывода. Драйверы устройств инкапсулируют код, зависящий от устройства, и реализуют стандартный интерфейс таким образом, чтобы код

содержал специфичные для устройства регистры чтения / записи. Драйвер устройства, как правило, записывается производителем устройства и поставляется вместе с устройством на компакт-диске.

Драйвер устройства выполняет следующие задания:

Чтобы принять запрос от устройства независимого программного обеспечения выше к нему.

Взаимодействовать с контроллером устройства для получения и ввода-вывода и выполнения необходимой обработки ошибок

Убедиться, что запрос выполнен успешно

Драйвер устройства обрабатывает запрос следующим образом. Предположим, что запрос приходит на чтение блока N. Если драйвер не используется в момент поступления запроса, он начинает выполнять запрос немедленно. В противном случае, если драйвер уже занят каким-либо другим запросом, он помещает новый запрос в очередь ожидающих запросов.

Обработчики прерываний

Обработчик прерываний, также известный как подпрограмма обслуживания прерываний или ISR, является частью программного обеспечения или, более конкретно, функцией обратного вызова в операционной системе или, более конкретно, в драйвере устройства, выполнение которого инициируется приемом прерывания.

Когда происходит прерывание, процедура прерывания делает все возможное, чтобы обработать прерывание, обновляет структуры данных и запускает процесс, который ожидал прерывания.

Механизм прерываний принимает адрес – число, которое выбирает конкретную процедуру / функцию обработки прерываний из небольшого набора. В большинстве архитектур этот адрес является смещением, хранящимся в таблице, называемой таблицей векторов прерываний. Этот вектор содержит адреса памяти специализированных обработчиков прерываний.

Независимое от устройства программное обеспечение ввода / вывода

Основная функция независимого от устройства программного обеспечения состоит в том, чтобы выполнять функции ввода-вывода, которые являются общими для всех устройств, и обеспечивать единый интерфейс с программным обеспечением уровня пользователя. Хотя написать полностью независимое от устройства программное обеспечение сложно, мы можем написать некоторые модули, которые являются общими для всех устройств. Ниже приведен список функций независимого от устройства программного обеспечения ввода-вывода.

Единый интерфейс для драйверов устройств

Именование устройства – Мнемонические имена сопоставляются с номерами устройств Major и Minor.

Защита устройства

Предоставление независимого от устройства размера блока

Буферизация, поскольку данные, поступающие с устройства, не могут быть сохранены в конечном месте назначения.

Распределение памяти на блочных устройствах

Распределение и выпуск выделенных устройств

Отчет об ошибках

User-Space I / O Software

Это библиотеки, которые предоставляют более богатый и упрощенный интерфейс для доступа к функциональности ядра или в итоге взаимодействуют с драйверами устройств. Большая часть программного обеспечения ввода-вывода пользовательского уровня состоит из библиотечных процедур, за некоторыми исключениями, таких как спулинговая система, которая является способом работы с выделенными устройствами ввода-вывода в многопрограммной системе.

Библиотеки ввода / вывода (например, stdio) находятся в пользовательском пространстве, чтобы предоставить интерфейс к независимому от устройства резидентному программному обеспечению ввода / вывода. Например, putchar (), getchar (), printf () и scanf () являются примером stdio библиотеки ввода-вывода пользовательского уровня, доступной в С-программировании.

Подсистема ввода / вывода ядра

Подсистема ввода / вывода ядра отвечает за предоставление многих услуг, связанных с вводом / выводом. Ниже приведены некоторые из предоставляемых услуг.

Планирование – ядро планирует набор запросов ввода-вывода для определения правильного порядка их выполнения. Когда приложение выполняет системный вызов блокирующего ввода-вывода, запрос помещается в очередь для этого устройства. Планировщик ввода-вывода ядра изменяет порядок очереди, чтобы повысить общую эффективность системы и среднее время отклика, испытываемое приложениями.

Буферизация – Подсистема ввода-вывода ядра поддерживает область памяти, известную как буфер, в котором хранятся данные, когда они передаются между двумя устройствами или между устройствами с операцией приложения. Буферизация выполняется для устранения несоответствия

скорости между производителем и потребителем потока данных или для адаптации между устройствами, которые имеют разные размеры передачи данных.

Кеширование – ядро поддерживает кеш-память, которая является областью быстрой памяти, в которой хранятся копии данных. Доступ к кэшированной копии более эффективен, чем доступ к оригиналу.

Буферизация и резервирование устройства. Буфер – это буфер, который содержит выходные данные для устройства, такого как принтер, который не может принимать чередующиеся потоки данных. Система буферизации копирует файлы очереди из очереди по очереди на принтер. В некоторых операционных системах спулинг управляется процессом системного демона. В других операционных системах это обрабатывается потоком ядра.

Обработка ошибок – операционная система, которая использует защищенную память, может защитить от многих видов аппаратных и прикладных ошибок.

Операционная система – Файловая система

Файл

Файл – это именованная коллекция связанной информации, которая записывается во вторичном хранилище, таком как магнитные диски, магнитные ленты и оптические диски. В общем случае файл представляет собой последовательность битов, байтов, строк или записей, значение которых определяется создателем и пользователем файлов.

Файловая структура

Структура файла должна соответствовать требуемому формату, понятному операционной системе.

Файл имеет определенную структуру в соответствии с его типом.

Текстовый файл – это последовательность символов, организованная в строки.

Исходный файл представляет собой последовательность процедур и функций.

Объектный файл – это последовательность байтов, организованных в блоки, которые понятны машине.

Когда операционная система определяет различные файловые структуры, она также содержит код для поддержки этих файловых структур. Unix, MS-DOS поддерживают минимальное количество файловых структур.

Тип файла

Тип файла относится к способности операционной системы различать различные типы файлов, такие как исходные текстовые файлы, двоичные файлы и т. Д. Многие операционные системы поддерживают многие типы файлов. Операционные системы, такие как MS-DOS и UNIX, имеют следующие типы файлов:

Обычные файлы

Это файлы, которые содержат информацию о пользователе.

Они могут иметь текст, базы данных или исполняемые программы.

Пользователь может применять к таким файлам различные операции, такие как добавление, изменение, удаление или даже удаление всего файла.

Каталог файлов

Эти файлы содержат список имен файлов и другую информацию, связанную с этими файлами.

Специальные файлы

Эти файлы также называются файлами устройств.

Эти файлы представляют физические устройства, такие как диски, терминалы, принтеры, сети, ленточные накопители и т. Д.

Эти файлы бывают двух типов –

Символьные специальные файлы – данные обрабатываются символьно, как в случае терминалов или принтеров.

Блочные специальные файлы – данные обрабатываются блоками, как в случае с дисками и лентами.

Механизмы доступа к файлам

Механизм доступа к файлу относится к способу доступа к записям файла. Есть несколько способов доступа к файлам –

Последовательный доступ

Прямой / Случайный доступ

Индексированный последовательный доступ

Последовательный доступ

Последовательный доступ – это доступ, к которому записи ведутся в некоторой последовательности, т. Е. Информация в файле обрабатывается по порядку, одна запись за другой. Этот метод доступа является наиболее примитивным. Пример: компиляторы обычно получают доступ к файлам таким способом.

Прямой / Случайный доступ

Организация файлов с произвольным доступом обеспечивает прямой доступ к записям.

Каждая запись имеет свой адрес в файле, с помощью которого к ней можно получить прямой доступ для чтения или записи.

Записи не должны быть в какой-либо последовательности в файле, и они не должны находиться в смежных местах на носителе данных.

Организация файлов с произвольным доступом обеспечивает прямой доступ к записям.

Индексированный последовательный доступ

Этот механизм построен на основе последовательного доступа.

Индекс создается для каждого файла, который содержит указатели на различные блоки.

Индекс ищется последовательно, а его указатель используется для прямого доступа к файлу.

Распределение пространства

Файлы выделяются дисковым пространством операционной системой. Операционные системы развертываются следующими тремя основными способами выделения дискового пространства для файлов.

Непрерывное распределение

Связанное распределение

Индексированное распределение

Непрерывное распределение

Каждый файл занимает непрерывное адресное пространство на диске.

Назначенный адрес диска в линейном порядке.

Легко реализовать.

Внешняя фрагментация является серьезной проблемой при таком типе распределения.

Связанное распределение

Каждый файл содержит список ссылок на блоки диска.

Каталог содержит ссылку / указатель на первый блок файла.

Нет внешней фрагментации

Эффективно используется в файле последовательного доступа.

Неэффективно в случае прямого доступа к файлу.

Индексированное распределение

Предоставляет решения проблем смежного и связанного размещения.

Создается индексный блок со всеми указателями на файлы.

Каждый файл имеет свой собственный индексный блок, в котором хранятся адреса дискового пространства, занимаемого файлом.

Каталог содержит адреса индексных блоков файлов.

Операционная система – Безопасность

Безопасность относится к обеспечению системы защиты ресурсов компьютерной системы, таких как ЦП, память, диск, программы программного обеспечения и, что наиболее важно, данных / информации, хранящихся в компьютерной системе. Если компьютерная программа запускается неавторизованным пользователем, он может нанести серьезный ущерб компьютеру или данным, хранящимся на нем. Таким образом, компьютерная система должна быть защищена от несанкционированного доступа, злонамеренного доступа к системной памяти, вирусов, червей и т. д.

Аутентификация

Одноразовые пароли

Программные угрозы

Системные угрозы

Классификация компьютерной безопасности

Аутентификация

Под аутентификацией понимается идентификация каждого пользователя системы и сопоставление исполняемых программ с этими пользователями. Операционная система несет ответственность за создание системы защиты, которая гарантирует, что пользователь, выполняющий определенную программу, является подлинным. Операционные системы обычно идентифицируют / аутентифицируют пользователей, используя следующие три способа:

Имя пользователя / пароль – пользователю необходимо ввести зарегистрированное имя пользователя и пароль в операционной системе для входа в систему.

Карта пользователя / ключ – пользователю необходимо ввести перфокарту в слот карты или ввести ключ, сгенерированный генератором ключей, в опции, предоставляемой операционной системой, для входа в систему.

Атрибут пользователя – шаблон отпечатка пальца / сетчатки глаза / подпись – пользователь должен передать свой атрибут через назначенное устройство ввода, используемое операционной системой для входа в систему.

Одноразовые пароли

Одноразовые пароли обеспечивают дополнительную безопасность наряду с обычной аутентификацией. В системе одноразовых паролей каждый раз, когда пользователь пытается войти в систему, требуется уникальный пароль. Если используется одноразовый пароль, его нельзя использовать снова. Одноразовый пароль реализован различными способами.

Случайные числа – пользователям предоставляются карточки с номерами, напечатанными вместе с соответствующими алфавитами. Система запрашивает числа, соответствующие нескольким случайно выбранным алфавитам.

Секретный ключ – пользователю предоставляется аппаратное устройство, которое может создать секретный идентификатор, сопоставленный с идентификатором пользователя. Система запрашивает такой секретный идентификатор, который должен генерироваться каждый раз перед входом в систему.

Сетевой пароль – некоторые коммерческие приложения отправляют одноразовые пароли пользователю на зарегистрированном мобильном телефоне / электронной почте, которые необходимо ввести до входа в систему.

Программные угрозы

Процессы операционной системы и ядро выполняют назначенную задачу в соответствии с инструкциями. Если пользовательская программа заставила этот процесс выполнять вредоносные задачи, она называется «Угрозы программе» . Одним из распространенных примеров программной угрозы является программа, установленная на компьютере, которая может сохранять и отправлять учетные данные пользователя по сети некоторому хакеру. Ниже приведен список некоторых известных программных угроз.

Троянский конь – такая программа перехватывает учетные данные пользователя и сохраняет их для отправки злоумышленнику, который впоследствии сможет войти на компьютер и получить доступ к системным ресурсам.

Дверь-ловушка – если программа, предназначенная для работы в соответствии с требованиями, имеет дыру в безопасности в своем коде и выполняет незаконные действия без ведома пользователя, то она вызывается для того, чтобы иметь ловушку.

Логическая бомба –это ситуация, когда программа ведет себя плохо только при соблюдении определенных условий, в противном случае она работает как настоящая программа. Это сложнее обнаружить.

Вирус – вирус, как следует из названия, может копировать себя в компьютерной системе. Они очень опасны и могут изменять / удалять пользовательские файлы, приводящие к краху систем. Вирус – это, как правило, небольшой код, встроенный в программу. Когда пользователь получает доступ к программе, вирус начинает внедряться в другие файлы / программы и может сделать систему непригодной для пользователя.

Системные угрозы

Под системными угрозами понимается злоупотребление системными службами и сетевыми подключениями, что создает пользователю проблемы. Системные угрозы могут использоваться для запуска программных угроз в полной сети, называемой программной атакой. Системные угрозы создают такую среду, в которой ресурсы операционной системы / пользовательские файлы используются не по назначению. Ниже приведен список некоторых известных системных угроз.

Worm –это процесс, который может снизить производительность системы, используя системные ресурсы до экстремальных уровней. Процесс червя генерирует несколько копий, каждая из которых использует системные ресурсы, не позволяя всем остальным процессам получать необходимые ресурсы. Черви-процессы могут даже отключить всю сеть.

Сканирование портов. Сканирование портов – это механизм или средство, с помощью которого хакер может обнаружить уязвимости системы, чтобы совершить атаку на систему.

Отказ в обслуживании – атаки типа «отказ в обслуживании» обычно не позволяют пользователю законно использовать систему. Например, пользователь может не иметь возможности пользоваться Интернетом, если отказ в обслуживании атакует настройки содержимого браузера.

Классификация компьютерной безопасности

Согласно Критериям оценки доверенных компьютерных систем Министерства обороны США, в компьютерных системах существует четыре классификации безопасности: A, B, C и D. Это широко используемые спецификации для определения и моделирования безопасности систем и решений безопасности. Ниже приводится краткое описание каждой классификации.

SN Тип классификации и описание

SN	**Тип классификации и описание**
1	**Наберите “A** Высший уровень. Использует формальные спецификации проекта и методы проверки. Обеспечивает высокую степень безопасности процесса.
2	**Тип Б**

	Обеспечивает обязательную защиту системы. Обладают всеми свойствами системы класса C2. Прикрепляет метку чувствительности к каждому объекту. Это трех типов. • **B1** – поддерживает метку безопасности каждого объекта в системе. Метка используется для принятия решений по контролю доступа. • **B2** – Расширяет метки чувствительности для каждого системного ресурса, такого как объекты хранения, поддерживает скрытые каналы и аудит событий. • **B3** – Позволяет создавать списки или группы пользователей для контроля доступа, чтобы предоставить доступ или отозвать доступ к указанному именованному объекту.
3	**Тип C** Обеспечивает защиту и ответственность пользователей, используя возможности аудита. Это двух типов. • **C1** – включает средства управления, чтобы пользователи могли защитить свою личную информацию и не дать другим пользователям случайно читать / удалять свои данные. Версии UNIX в основном относятся к классу Cl. • **C2** – добавляет управление доступом на индивидуальном уровне к возможностям системы уровня Cl.
4	**Тип D** Самый низкий уровень. Минимальная защита. MS-DOS, Window 3.1 попадают в эту категорию.

4. Архитектура операционной системы на примере Linux

Linux является одной из популярных версий операционной системы UNIX. Это открытый исходный код, так как его исходный код находится в свободном доступе. Это бесплатно для использования. Linux был разработан с учетом совместимости с UNIX. Его функциональный список очень похож на UNIX.

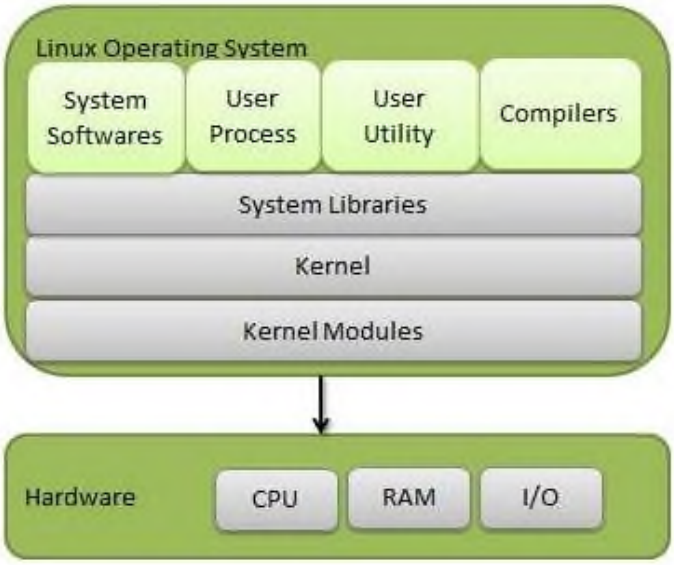
Компоненты системы Linux

Операционная система Linux состоит в основном из трех компонентов

Ядро – Ядро является основной частью Linux. Он отвечает за все основные виды деятельности этой операционной системы. Он состоит из различных модулей и напрямую взаимодействует с базовым оборудованием. Ядро предоставляет необходимую абстракцию для скрытия сведений об оборудовании низкого уровня в системных или прикладных программах.

Системная библиотека – Системные библиотеки – это специальные функции или программы, с помощью которых прикладные программы или системные утилиты получают доступ к функциям ядра. Эти библиотеки реализуют большинство функций операционной системы и не требуют прав доступа к коду модуля ядра.

System Utility – программы System Utility отвечают за выполнение специализированных задач индивидуального уровня.



Операционная система Linux

Режим ядра против режима пользователя

Код компонента ядра выполняется в специальном привилегированном режиме, называемом режимом ядра, с полным доступом ко всем ресурсам компьютера. Этот код представляет собой единый процесс, выполняется в одном адресном пространстве и не требует переключения контекста и, следовательно, является очень эффективным и быстрым. Ядро запускает каждый процесс и предоставляет системные сервисы процессам, обеспечивает защищенный доступ к оборудованию для процессов.

Код поддержки, который не требуется для запуска в режиме ядра, находится в системной библиотеке. Пользовательские программы и другие системные программы работают в режиме пользователя, который не имеет доступа к системному оборудованию и коду ядра. Пользовательские

программы / утилиты используют системные библиотеки для доступа к функциям ядра для получения низкоуровневых задач системы.

Основные характеристики

Ниже приведены некоторые важные функции операционной системы Linux.

Переносимость – Портативность означает, что программное обеспечение может работать на разных типах оборудования одинаково. Ядро Linux и прикладные программы поддерживают их установку на любую аппаратную платформу.

Открытый исходный код – исходный код Linux находится в свободном доступе, и это проект развития сообщества. Несколько групп работают в сотрудничестве, чтобы расширить возможности операционной системы Linux, и она постоянно развивается.

Многопользовательский – Linux является многопользовательской системой, что означает, что несколько пользователей могут одновременно получать доступ к системным ресурсам, таким как память / оперативная память / прикладные программы.

Мультипрограммирование – Linux – это мультипрограммная система, которая позволяет одновременно запускать несколько приложений.

Иерархическая файловая система – Linux предоставляет стандартную файловую структуру, в которой располагаются системные файлы / пользовательские файлы.

Shell – Linux предоставляет специальную программу-интерпретатор, которую можно использовать для выполнения команд операционной системы. Он может использоваться для выполнения различных типов операций, вызова прикладных программ. и т.п.

Безопасность – Linux обеспечивает безопасность пользователя, используя функции аутентификации, такие как защита паролем / контролируемый доступ к определенным файлам / шифрование данных.

Переносимость – Портативность означает, что программное обеспечение может работать на разных типах оборудования одинаково. Ядро Linux и прикладные программы поддерживают их установку на любую аппаратную платформу.

Открытый исходный код – исходный код Linux находится в свободном доступе, и это проект развития сообщества. Несколько групп работают в сотрудничестве, чтобы расширить возможности операционной системы Linux, и она постоянно развивается.

Многопользовательский – Linux является многопользовательской системой, что означает, что несколько пользователей могут одновременно получать доступ к системным ресурсам, таким как память / оперативная память / прикладные программы.

Мультипрограммирование – Linux – это мультипрограммная система, которая позволяет одновременно запускать несколько приложений.

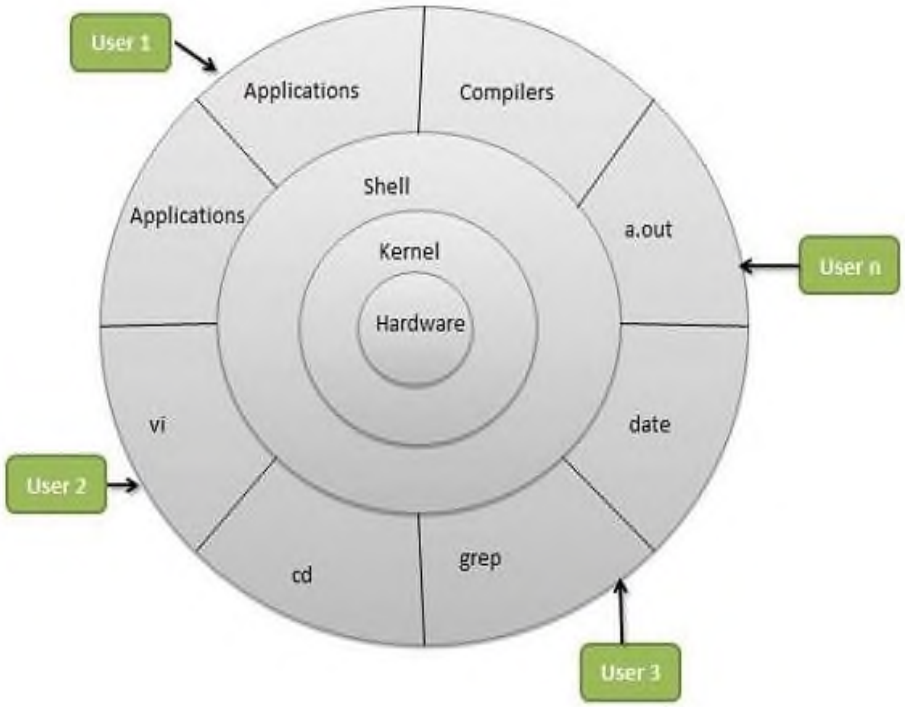
Иерархическая файловая система – Linux предоставляет стандартную файловую структуру, в которой располагаются системные файлы / пользовательские файлы.

Shell – Linux предоставляет специальную программу-интерпретатор, которую можно использовать для выполнения команд операционной системы. Он может использоваться для выполнения различных типов операций, вызова прикладных программ. и т.п.

Безопасность – Linux обеспечивает безопасность пользователя, используя функции аутентификации, такие как защита паролем / контролируемый доступ к определенным файлам / шифрование данных.

Архитектура

На следующем рисунке показана архитектура системы Linux –



Архитектура операционной системы Linux

Архитектура системы Linux состоит из следующих уровней:

Аппаратный уровень – Аппаратное обеспечение состоит из всех периферийных устройств (RAM / HDD / CPU и т. Д.).

Ядро – это основной компонент операционной системы, напрямую взаимодействующий с оборудованием, обеспечивающий низкоуровневые услуги для компонентов верхнего уровня.

Shell – интерфейс к ядру, скрывающий сложность функций ядра от пользователей. Оболочка принимает команды от пользователя и выполняет функции ядра.

Утилиты – служебные программы, которые предоставляют пользователю большую часть функциональных возможностей операционных систем.

5. Процессы, управление процессами в LINUX

В Linux для каждой отдельной программы, при ее запуске создается процесс. Неважно запускаете программу вы вручную самостоятельно или это делает система или ядро. Например, программа инициализации, которая запускается сразу после завершения загрузки ядра, тоже имеет свой процесс с идентификатором 0. Процессы в linux можно описать как контейнеры, в которых хранится вся информация о состоянии и выполнении программы. Если программа работает хорошо, то все нормально, но если она зависла или вам нужно настроить ее работу может понадобиться управление процессами в Linux.

Что такое процесс?

Начнем с того, что разберемся в терминах. По сути, процесс — это каждая программа. Для каждой запускаемой программы создается отдельный процесс. В рамках процесса программе выделяется процессорное время, оперативная память и другие системные ресурсы. У каждого процесса есть свой идентификатор, Proccess ID или просто PID, по ним, чаще всего и определяются процессы Linux. PID определяется неслучайно, программа инициализации получает PID 1, а каждая следующая запущенная программа - на единицу больше. Таким образом PID пользовательских программ доходит уже до нескольких тысяч.

На самом деле, процессы Linux не настолько абстрактны, какими они вам сейчас кажутся. Их вполне можно попытаться пощупать. Откройте ваш файловый менеджер, перейдите в корневой каталог, затем откройте папку /proc. Видите здесь кучу номеров? Так вот это все - PID всех запущенных процессов. В каждой из этих папок находится вся информация о процессе.

Например, посмотрим папку процесса 1. В папке есть другие под каталоги и много файлов. Файл cmdline содержит информацию о команде запуска процесса:

```
cat /proc/1/cmdline
/usr/lib/systemd/systemd
```

Поскольку у меня используется система инициализации Systemd, то и первый процесс запускается для нее. С помощью каталога /proc можно сделать все. Но это очень неудобно, особенно учитывая количество запущенных процессов в системе. Поэтому для реализации нужных задач существуют специальные утилиты. Перейдем к рассмотрению утилит, которые позволяют реализовать управление процессами в Linux.

В Linux есть очень большое количество утилит для решения различных задач по управлению процессами. Это и такие многофункциональные решения, как htop, top, а также простые утилиты, например, ps, kill, killall, who и т д. Мы остановимся на работе с программой htop и ее аналогами в форме утилит в стиле GNU, одна утилита - одна функция.

Давайте установим htop, если она у вас еще не установлена. В Ubuntu это делается так:

```
sudo apt install htop
```

В других дистрибутивах вам нужно просто использовать свой менеджер пакетов. Имя пакета такое же.

Посмотреть запущенные процессы

Это очень простая задача, и также просто она решается. Для этого существует множество утилит, начиная от обычной ps, до более продвинутых интерактивных top, htop и так далее.

Открыв htop, мы сразу видим список запущенных процессов. Конечно, здесь отображены не все процессы linux, их-то в системе очень много, вы уже знаете, все они на один экран не поместятся. По умолчанию выводятся процессы, запущенные от имени вашего пользователя:

```
htop
```

Вы можете увидеть такую информацию о процессе:

PID - идентификатор процесса

USER - пользователь, от которого был запущен процесс

PRI - приоритет процесса linux на уровне ядра (обычно NI+20)

NI - приоритет выполнения процесса от -20 до 19

S - состояние процесса

CPU - используемые ресурсы процессора

MEM - использованная память

TIME - время работы процесса

К отображению можно добавить и дополнительные параметры, но эти главные. Добавить параметры можно с помощью меню Setup. Там все очень просто, читайте подсказки и следуйте указаниям. Например, добавлен параметр PPID:

Очень важной особенностью программы есть то, что вы можете сортировать процессы в Linux по нужному параметру. Просто кликните по названию параметра, оно выделится зеленым и будет выполнена сортировка. Например, хотите посмотреть в каком порядке запускались процессы, сортируем по PID:

htop2

Также есть интересная возможность разместить процессы в виде дерева. Вы сможете увидеть, каким процессом был запущен тот или иной процесс. Для отображения дерева нажмите кнопку F5:

htop3

Почти те же действия вы можете выполнять с помощью программы ps. Только здесь нет такого удобного интерактивного режима. Все делается с помощью опций.

ps

Рассмотрим основные опции, которые будем использовать:

-e - вывести информацию обо всех процессах

-a - вывести информацию обо всех наиболее часто запрашиваемых процессах

-t - показывать только процессы из этого терминала

-p - показывать информацию только об указанном процессе

-u - показывать процессы только определенного пользователя

Одним словом, чтобы посмотреть все активные на данный момент процессы в linux, используется сочетание опций aux:

ps aux

Программа показывает все те же параметры, только здесь нет интерактивного интерфейса. Думаете здесь нельзя отсортировать процессы, но ошибаетесь, можно. Для этого есть опция sort. Вы можете сортировать их по любому полю, например:

ps aux --sort=%mem

Список будет отсортирован в обратном порядке, внизу значения больше, вверху - меньше. Если нужно в обратном порядке, добавьте минус:

ps aux --sort=-%cpu

ps1

В качестве поля для сортировки могут быть использованы приоритеты процессов Linux или любые другие параметры. Также вы можете обрезать вывод, если не нужно выводить всю информацию:

ps aux | tail

Казалось бы, у ps нет возможности стоить деревья процессов. Но не совсем, для этого существует отдельная команда:

pstree

pstree

Поиск процессов в linux

Список процессов, это хорошо. Но иногда, когда какой-нибудь процесс завис и нужно убить процесс Linux или нам нужно провести с ним какие-либо действия, нужно выделить этот процесс из списка, узнать его PID и информацию о нем.

Чтобы найти процесс linux в htop можно использовать кнопку F3. Нажмите F3 и наберите нужное слово. Дальше, чтобы перейти к следующему вхождению нажимайте F2 или Esc для завершения поиска:

htop4

Для поиска процессов в htop можно использовать также фильтр htop. Нажмите F4, введите слово и будут выведены только процессы linux, имя которых включает это слово.

htop6

В утилите ps фильтрации нет, но зато мы можем использовать утилиту grep, перенаправив вывод ps на нее чтобы найти процесс linux:

ps aux | grep chromium

Это очень часто употребляемая команда.

Изменение приоритета процессов

Приоритет процесса linux означает, насколько больше процессорного времени будет отдано этому процессу по сравнению с другими. Так мы можем очень тонко настроить какая программа будет работать быстрее, а какая медленнее. Значение приоритета может колебаться от 19 (минимальный приоритет) до -20 - максимальный приоритет процесса linux. Причем, уменьшать приоритет можно с правами обычного пользователя, но чтобы его увеличить нужны права суперпользователя.

В htop для управления приоритетом используется параметр Nice. Напомню, что Priv, это всего лишь поправка, она в большинстве случаев больше за Nice на 20. Чтобы изменить приоритет процесса просто установите на него курсор и нажимайте F7 для уменьшения числа (увеличения приоритета) или F8 - для увеличения числа.

Но и для решения этой задачи управления процессами Linux необязательно использовать htop. Вы можете сделать все и другими командами. Например, команда nice. С помощью нее вы можете указать приоритет для запускаемого процесса:

nice -n 10 apt-get upgrade

Или изменить приоритет для существующего по его pid:

renice -n 10 -p 1343

Завершение процессов в linux

Если процесс завис и не отвечает, его необходимо завершить. В htop, чтобы убить процесс Linux, просто установите курсор на процесс и нажмите F9:

htop7

Система для управления процессами использует определенные сигналы, есть сигналы, которые указывают процессу завершиться. Вот несколько основных сигналов:

SIGTERM - попросить процесс сохранить данные и завершится

SIGKILL - завершить процесс немедленно, без сохранения

Вообще сигналов есть несколько десятков, но мы не будем их рассматривать. Отправим сигнал SIGKILL:

Также можно воспользоваться утилитой kill:

kill -TERM 1943

Также можно уничтожить процесс по имени:

killall chromium

Ограничение процессов

Управление процессами в Linux позволяет контролировать практически все. Вы уже видели, что можно сделать, но можно еще больше. С помощью команды ulimit и конфигурационного файла /etc/security/limits.conf вы можете ограничить процессам доступ к системным ресурсам, таким как память, файлы и процессор. Например, вы можете ограничить память процесса Linux, количество файлов и т д.

Запись в файле имеет следующий вид:

<домен> <тип> <элемент> <значение>

домен - имя пользователя, группы или UID

тип - вид ограничений - soft или hard

элемент - ресурс, который будет ограничен

значение - необходимый предел

Жесткие ограничения устанавливаются суперпользователем и не могут быть изменены обычными пользователями. Мягкие, soft ограничения могут меняться пользователями с помощью команды ulimit.

Рассмотрим основные ограничения, которые можно применить к процессам:

nofile - максимальное количество открытых файлов

as - максимальное количество оперативной памяти

stack - максимальный размер стека

cpu - максимальное процессорное время

nproc - максимальное количество ядер процессора

locks - количество заблокированных файлов

nice - максимальный приоритет процесса

Например, ограничим процессорное время для процессов пользователя sergiy:

sergiy hard nproc 20

Посмотреть ограничения для определенного процесса вы можете в папке proc:

cat /proc/PID/limits

Max cpu time unlimited unlimited seconds

Max file size unlimited unlimited bytes

Max data size unlimited unlimited bytes

Max stack size 204800 unlimited bytes

Max core file size 0 unlimited bytes

Max resident set unlimited unlimited bytes

Max processes 23562 23562 processes

Max open files 1024 4096 files

Max locked memory 18446744073708503040 18446744073708503040 bytes

Max address space unlimited unlimited bytes

Max file locks unlimited unlimited locks

Max pending signals 23562 23562 signals

Max msgqueue size 819200 819200 bytes

Max nice priority 0 0

Max realtime priority 0 0

Max realtime timeout unlimited unlimited us

Ограничения, измененные, таким образом вступят в силу после перезагрузки. Но мы можем и устанавливать ограничения для текущего командного интерпретатора и создаваемых им процессов с помощью команды ulimit.

Вот опции команды:

-S - мягкое ограничение

-H - жесткое ограничение

-a - вывести всю информацию

-f - максимальный размер создаваемых файлов

-n - максимальное количество открытых файлов

-s - максимальный размер стека

-t - максимальное количество процессорного времени

-u - максимальное количество запущенных процессов

-v - максимальный объем виртуальной памяти

Например, мы можем установить новое ограничение для количества открываемых файлов:

ulimit -Sn 1024

Теперь смотрим:

ulimit -Sn

Установим лимит оперативной памяти:

ulimit -Sv 500000

6. Память, управление памятью в LINUX

Базовой единицей в организации памяти для систем UNIX/Linux является страница памяти. Обладающая размером от 4 Кбайт, которому соответствует объём физического пространства в оперативной или виртуальной (область подкачки на диске или другом устройстве хранения) памяти. При запуске процессов они запрашивают у системы (т. е. у ядра посредством соответствующих системных вызовов) память для своей работы. А в ответ на это ядро выделяет для них достаточное количество страниц памяти. Виртуальная память или как её ещё называют, «резервное ЗУ» (резервное запоминающее устройство) для страниц памяти. Которые содержат, к примеру, исходный текст исполняемого приложения, представляют собой обычные исполняемые файлы на диске. Равно как и для других файлов данных резервным ЗУ являются сами файлы. Информация о том, как взаимосвязаны страницы физической и виртуальной памяти хранится в соответствующих таблицах страниц памяти.

Для работы с памятью в Linux (как и в других UNIX-подобных системах) характерно такое явление как «страничный обмен» (paging). Оно заключается в том, что ядро выделяет процессам столько памяти, сколько им необходимо. В том смысле, чтобы её (памяти) всегда хватало. Это достигается за счёт расширения физической памяти за счёт виртуальной, т. е. «подкачки». Поскольку выполнение процессов должно происходить в реальной физической памяти. То ядро постоянно перемещает страницы памяти

процессов между физической и виртуальной памятью. Забегая вперёд, следует отметить, что в виртуальной памяти хранятся «неактивные» страницы. Которые не задействованы процессом в данный момент, но необходимые ему для полноценной работы впоследствии.

Основные принципы управления памятью в Linux

Первое, на что следует обратить внимание, это то, что ядро старается управлять памятью таким образом, чтобы недавно используемые процессом страницы находились в физической памяти. И в свою очередь, «неактивные» или редко используемые страницы перемещаются и хранятся в виртуальной памяти в области «подкачки». Такой механизм распределения памяти называется LRU (least recently used) — замещение наиболее редко используемых страниц.

Вторым важнейшим аспектом в работе памяти является использование кеш-буфера страниц. Это вытекает из работы алгоритма LRU, который довольно сложен в своей реализации. Поскольку следить за всеми обращениями к страницам — это в некоторых случаях, довольно ощутимые потери в производительности системы. Использование же страничного кеш-буфера куда проще в своей реализации при тех же самых результатах. К тому же данный подход имеет огромный модернизационный потенциал (в отличие от LRU) и алгоритмы анализа содержимого кеш-буфера (для определения, какие страницы должны быть перемещены из виртуальной памяти) постоянно совершенствуются. Что заметно сказывается на производительности и эффективности управления памятью.

Когда процессу не хватает памяти, то ядро начинает искать «занятые» страницы. Которые можно использовать для «голодающего» процесса. Обычно такими страницами являются те, что давно не были использованы. Ядро проверяет их на предмет модификации каким-либо процессом. Для этого существуют определённые признаки, при последнем обращении и если изменения были, то такие страницы помечаются ядром как «грязные». Т. е. такие, которые ещё нужны процессам. Для повторного использования памяти такие страницы сначала обязательно переносятся в виртуальную память. Все же остальные страницы являются «чистыми». И поэтому ядро их использует для предоставления другим или «голодающим» процессам.

Особенности управления памятью в Linux

Когда происходит обращение к страницам памяти, которые некоторое или долгое время не использовались, т. е. к «неактивным» страницам. То ядро выполняет с ними несколько важных задач:

- возвращает ссылки на эти страницы в соответствующей таблице страниц;
- сбрасывает в нулевое значение время «неиспользования» этих страниц;
- помечает эти страницы как «активные».

Со страницами, находящимися в виртуальной памяти не всё так однозначно. Дело в том, что для того, чтобы «активизировать» такие страницы, они должны быть предварительно прочитаны с диска.

Системное ядро комплектуется специализированными модулями. Которые содержат алгоритмы и даже целые технологии. С помощью которых система довольно эффективно «предсказывает», сколько может потребоваться памяти при разной степени активности и загруженности процессов. Эти алгоритмы имеют своей целью обеспечение процессов свободной памятью с максимальной эффективностью. Т. е. так, чтобы процессам как можно реже приходилось простаивать в «ожидании» выгрузки очередной страницы в свободную память. Таким образом, наблюдая за состоянием страничного обмена во время рабочей нагрузки системы, можно делать выводы о том, нужна ли ей дополнительная память. Если страничный обмен интенсивный — то однозначно следует установить дополнительные модули ОЗУ.

Если же происходит так, что процессам не хватает ни реальной физической, ни виртуальной памяти. Т. е. когда память полностью исчерпана, то система начинает завершать (а точнее уничтожать) целые процессы. Либо запрещает создание новых. Конечно, в этом случае в первую очередь уничтожаются наиболее «безболезненные» для системы процессы. Однако в таких случаях даже «на глаз» и по собственным ощущениям видно, что она большую часть времени тратит на управление памятью, а не на выполнение рабочих задач.

В Linux можно настроить параметр, который задаёт, насколько быстро ядро должно «отбирать» страницы памяти у процессов. Которым они менее нужны для процессов, которым они на данный момент необходимы. Этот параметр содержится в файле /proc/sys/vm/swappiness и по-умолчанию равен 60. Если задать его меньшим значением (например, 0). То ядро будет забирать страницы процесса в самую последнюю очередь. Используя вместо этого, любые другие варианты. Если это значение в пределах между 60 и 100. То страницы будут отбираться у процессов с более высокой вероятностью. Вариант с изменением данного параметра на самом деле говорит о том, что

необходимо либо снизить нагрузку на систему. Адаптировав её для других менее производительных задач, либо увеличить объём ОЗУ.

Заключение

В заключение следует отметить, что схема работы и управления с памятью в Linux не так уж и сложна. Гораздо сложнее специфические задачи. Такие как анализ содержимого кеш-буфера страниц и его интеллектуальное использование. Но это уже работа программистов и разработчиков. В свою очередь понимание основ управления памятью помогает лучше распоряжаться ресурсами системы. Что для системных администраторов очень важно.

7. Шедулер в Linux

Планирование – это процесс распределения ресурсов системы для выполнения задач. Мы рассмотрим вариант, в котором ресурсом является одно или несколько ядер процессора, а задачи представлены потоками или процессами, которые нужно выполнить.

Само планирование осуществляется планировщиком, который нацелен:

- Максимизировать пропускную способность, то есть количество задач, выполняемых за единицу времени.
- Минимизировать время ожидания, то есть время, прошедшее с момента готовности процесса до начала его выполнения.
- Минимизировать время ответа, то есть время, прошедшее с момента готовности процесса до завершения его выполнения.
- Максимизировать равнодоступность, то есть справедливое распределение ресурсов между задачами.

Типы процессов в Linux

В Linux процессы делятся на два типа:

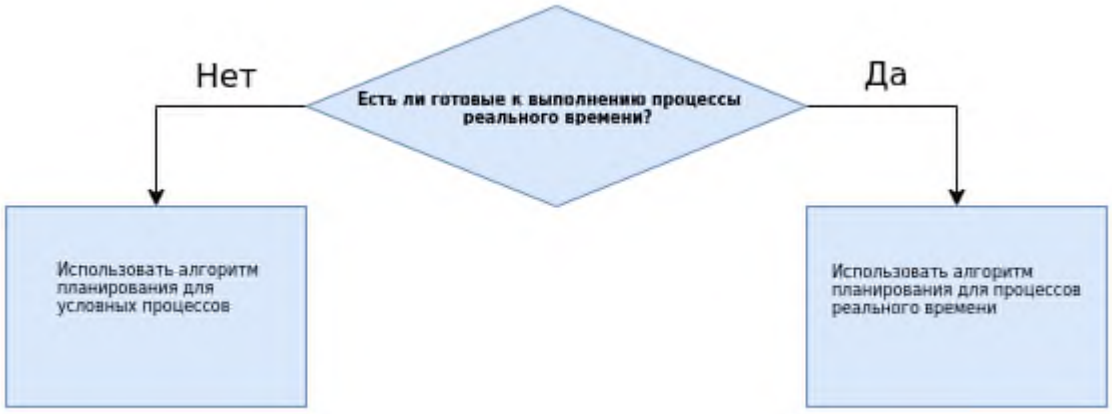
- Процессы реального времени.
- Условные процессы.

Процессы реального времени должны вписываться в границы времени ответа, независимо от загрузки системы. Иначе говоря, такие процессы являются срочными и ни при каких условиях не откладываются.

В качестве примера можно привести процесс переноса, отвечающий за распределение рабочей нагрузки между ядрами ЦПУ.

Условные же процессы не ограничиваются строгими рамками времени ответа и в случае занятости системы могут подвергаться задержкам.

У каждого типа процессов есть свой алгоритм планирования. При этом пока есть готовые к выполнению процессы реального времени, выполняться будут они, оставляя условные процессы в ожидании.



Планирование в реальном времени

В случае с планированием в реальном времени используются две политики, SCHED_RR и SCHED_FIFO.

Политика определяет количество выделяемого процессу времени, а также принцип организации очереди на выполнение.

Суть в том, что готовые к выполнению процессы хранятся в очереди, откуда выбираются планировщиком на основе той или иной политики.

SCHED_FIFO

В данной политике планировщик выбирает процесс, ориентируясь на время его поступления (FIFO = первым вошел, первым вышел).

Процесс с политикой планирования SCHED_FIFO может «освободить» ЦПУ в нескольких случаях:

Процесс ожидает, к примеру, операции ввода/вывода, после чего по возвращению в состояние «готов» помещается в конец очереди.

Процесс уступил ЦПУ через системный вызов sched_yield, после чего он тут же возвращается в конец очереди.

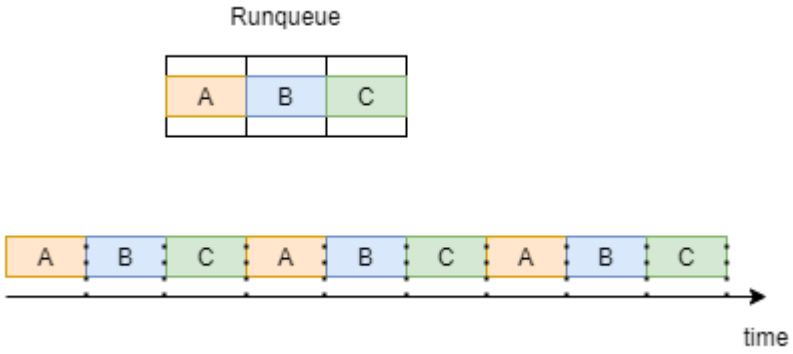
SCHED_RR

SCHED_RR подразумевает циклическое планирование.

В этой политике каждый процесс в очереди получает интервал времени (квант) и выполняется в свою очередь (исходя из приоритета) по циклическому принципу.

Для лучшего понимания рассмотрим пример, где в очереди находятся три процесса, A B C, все из которых работают по политике SCHED_RR.

Как показано ниже, каждый процесс получает квант времени и выполняется в свою очередь. После однократного выполнения всех процессов они повторяются в той же последовательности.



Обобщение по планированию в реальном времени

Процесс реального времени может планироваться по двум разным политикам, SCHED_FIFO и SCHED_RR.

Политика влияет на принцип работы очереди и определяет, сколько времени нужно выделить тому или иному процессу.

Условное планирование

Здесь мы знакомимся с Completely Fair Scheduler (CFS, абсолютно справедливый планировщик), представляющим алгоритм планирования условных процессов, появившийся в версии Linux 2.6.23.

CFS фокусируется на одной из них – он стремится к максимальной равноправности процессов, то есть обеспечивает выделение всем процессам равных квантов времени ЦПУ.

Обратите внимание, что процессы с повышенным приоритетом все равно могут получать на обработку больше времени.

Для лучшего понимания принципа работы CFS нужно познакомиться с новым термином – виртуальное время выполнения (vruntime).

Виртуальное время выполнения

Виртуальное время выполнения процесса – это количество времени, потраченного именно на выполнение, без учета любых ожиданий.

Как было сказано, CFS стремится быть максимально справедливым, в связи с чем по очереди планирует готовый к выполнению процесс с минимальным виртуальным временем.

CFS задействует переменные, содержащие максимальное и минимальное виртуальное время выполнения, и чуть позже станет ясно зачем.

CFS —Абсолютно справедливый планировщик

Прежде чем перейти к принципу работы этого алгоритма, нужно понять, какие структуры данных он использует.

CFS задействует красно-черное дерево, представляющее бинарное дерево поиска – то есть добавление, удаление и поиск выполняются за O(logN), где N выражает количество процессов.

Ключом в этом дереве выступает виртуальное время выполнения процесса. Новые процессы или процесс, возвращающиеся из ожидания в состояние готовности, добавляются в дерево с ключом vruntime = min_vruntime. Это очень важный момент, который позволяет избежать дефицита внимания ЦПУ для старых процессов.

Вернемся к самому алгоритму. В первую очередь он устанавливает себе лимит времени – sched_latency.

В течение этого времени алгоритм стремится выполнить все готовые процессы – N. Это означает, что каждый процесс получит интервал времени равный временному лимиту, поделенному на количество процессов: Qi = sched_latency/N.

Когда процесс исчерпывает свой интервал (Qi), алгоритм выбирает в дереве следующий процесс с наименьшим виртуальным временем.

Рассмотрим ситуацию, которая может стать проблематичной для такой схемы работы алгоритма.

Предположим, что алгоритм выбрал лимит времени 48мс при наличии 6 процессов – в этом случае каждый процесс получит на выполнение по 8мс.

Но что произойдет, если система окажется перегружена процессами? Предположим, что лимит времени остается равен 48мс, но теперь у нас 32 процесса. В результате каждый получит уже всего по 1.5мс на выполнение, что приведет к замедлению работы всей системы.

Почему?

Все дело в переключении контекста, которое подразумевает сохранение состояния процесса или потока с последующим его восстановлением и продолжением выполнения.

Каждый раз, когда процесс исчерпывает свое время на выполнение, и планируется очередной процесс, активируется переключение контекста, которое также занимает некоторое время.

Предположим, что на него уходит 1мс. В первом примере, где каждому процессу у нас отводилось по 8мс, это вполне допустимо. Так мы тратим 1мс на переключение контекста и 7мс на фактическое выполнение процесса.

А вот во втором примере на выполнение каждого процесса останется уже всего по 0.5мс – то есть большая часть времени уходит на переключение контекста, отсюда и проблема с выполнением.

Для того, чтобы исправить ситуацию, мы вводим новую переменную, которая определит минимальную протяженность кванта времени выполнения – min_granularity.

Представим, что min_granularity = 6мс, и вернемся к нашему примеру, где лимит времени равен 48мс при наличии 32 процессов.

С помощью той же формулы, что и прежде, мы получаем по 1.5мс на каждый процесс, но теперь такой вариант не допускается, так как min_granularity задает минимальный квант времени, который должен получить каждый процесс.

В данном случае, где Qi < min_granularity, мы берем Qi равным min_granularity, то есть 6мс, и соответствующим образом изменяем временной лимит. В результате он составит Qi x N = 6мс x 32 = 192мс.

На данный момент отличия между CFS и RR могут оказаться недостаточно наглядны, поскольку они оба определяют временные интервалы и организуют порядок выполнения процессов. Для лучшего обобщения и понимания различий между этими алгоритмами я приведу краткую таблицу:

RR – циклический список CFS – абсолютно справедливый планировщик

Квант времени статичен и не зависит от количества процессов в системе.

Квант времени динамичен и может изменяться в соответствии с количеством процессов в системе.

По исчерпанию процессом его кванта времени, RR выбирает очередной процесс с наименьшим виртуальным временем из циклического списка.

По исчерпанию процессом его кванта времени, CFS выбирает очередной процесс с наименьшим виртуальным временем из красно-черного дерева.

8. Дисковые системы

Диск подключают или делают его разметку на серверах — как правило из-под консоли. Из-под консоли, удаленно, делят Диск на разделы, форматируют в определенную файловую систему.

Так же есть те, кто предпочитает, именно в консоли работать с дисками. Делать все необходимые манипуляции с ними, обосновывая это тем, что так проще и надежнее.

Если вы используете Linux с графическим интерфейсом, то вы можете воспользоваться программой GParted.

Каждый выбирает то, что ему больше подходит, главное, чтоб была решена поставленная задача.

Прежде чем приступить к работе с диском нужно посмотреть, как определяет его система, а также всю структуру дисков в системе.

Смотрим информацию о дисках в системе

Прежде чем смотреть подключенные диски и структуру разделов, необходимо знать, как они обозначаются, при просмотре информации в консоли.

В зависимости от интерфейса подключения их обозначают:

sd — подключены по SCSI

SCSI (англ.Small Computer System Interface, произносится «скази» (встречается вариант as-si-as-ai)) представляет собой набор стандартов для физического подключения и передачи данных между компьютерами и периферийными устройствами.

hd — устройства ATA

ATA (англ. Advanced Technology Attachment) или IDE (англ. Integrated Drive Electronics) — параллельный интерфейс подключения накопителей (гибких дисков, жёстких дисков и оптических дисководов) к компьютеру. В 1990-е годы был стандартом на платформе IBM PC; в настоящее время вытеснен своим последователем — SATA

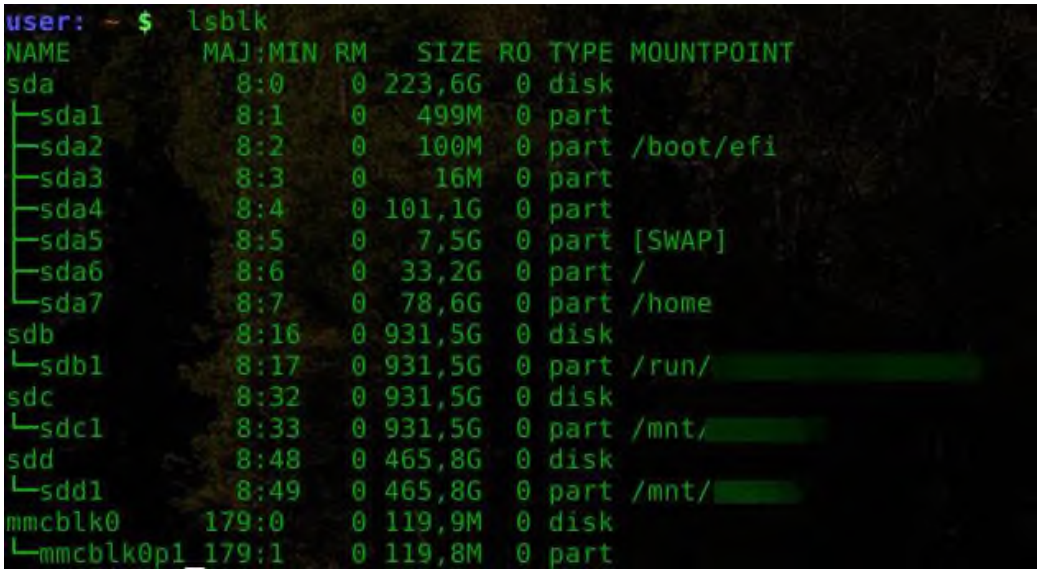
vd — виртуальные устройста

mmcblk — обозначают карты памяти в картридере

В современных реалиях, на сегодняшний день, используется как правило SCSI интерфейс, даже флешки и старые диски, путем использования переходника, используют именно этот тип подключения.

Наглядно все показывает команда

lsblk



Так же команда lsblk может показать полезные данные:

lsblk -o +UUID

Покажет UUID файловой системы (универсальный уникальный идентификатор)

lsblk -o +RM

Покажет подключаемые устройства — накопители USB

lsblk -d -o +MODEL

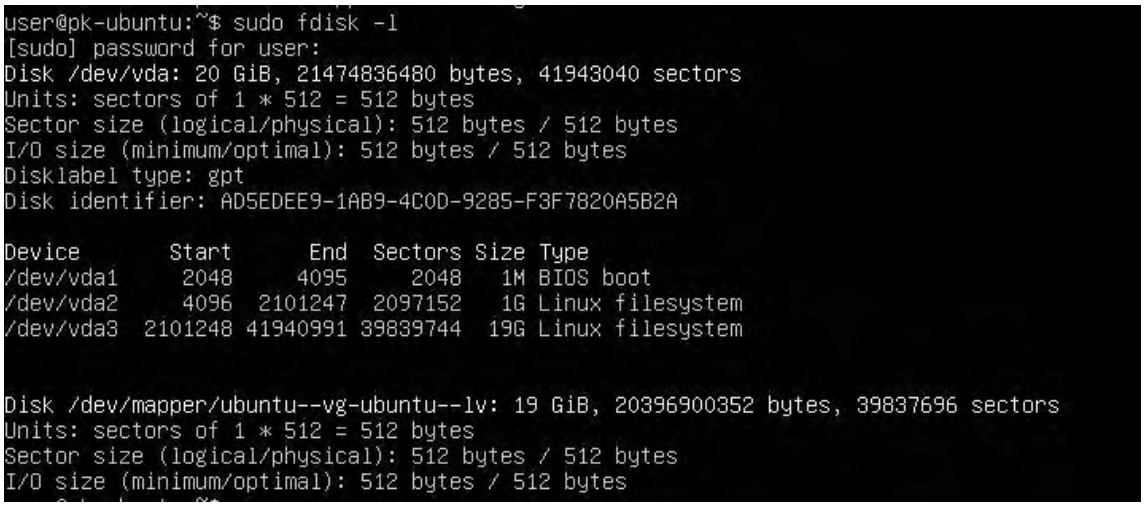
Так же выведет ко всем параметрам еще и модель диска, а это иногда очень нужно бывает.

Обо всех дополнительных опциях команды lsblk можете узнать, введя команду:

lsblk --help

Следующая команда выведет более подробную информацию

sudo fdisk -l



Следующая команда

sudo parted -l

```
user@pk-ubuntu:~$ sudo parted -l
[sudo] password for user:
Model: Linux device-mapper (linear) (dm)
Disk /dev/mapper/ubuntu--vg-ubuntu--lv: 20,4GB
Sector size (logical/physical): 512B/512B
Partition Table: loop
Disk Flags:

Number  Start  End     Size    File system  Flags
 1      0,00B  20,4GB  20,4GB  ext4


Warning: Unable to open /dev/sr0 read-write (Read-only file system).  /dev/sr0
has been opened read-only.
Model: QEMU QEMU DVD-ROM (scsi)
Disk /dev/sr0: 1016MB
Sector size (logical/physical): 2048B/2048B
Partition Table: mac
Disk Flags:

Number  Start   End     Size    File system  Name   Flags
 1      2048B   6143B   4096B                Apple
 2      75,8MB  78,3MB  2458kB               EFI


Model: Virtio Block Device (virtblk)
Disk /dev/vda: 21,5GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start   End     Size    File system  Name  Flags
 1      1049kB  2097kB  1049kB                     bios_grub
 2      2097kB  1076MB  1074MB  ext4
 3      1076MB  21,5GB  20,4GB
```

Эта команда выводит информацию о разделах файловой системы еще и показывая UUID, вот в таком виде

sudo blkid

```
user@pk-ubuntu:~$ sudo blkid
[sudo] password for user:
/dev/vda2: UUID="621777a3-ad69-44b1-8f51-dc34992639e9" TYPE="ext4" PARTUUID="10acea2d-6f6a-4fd0-b52c-5e8496dc4cc2"
/dev/vda3: UUID="HUH2mN-VPmx-UEtw-hoe8-zO5t-cb4c-5uV9MC" TYPE="LVM2_member" PARTUUID="5cb24db7-4194-4a9b-9aca-8b57a6c45c1c"
/dev/mapper/ubuntu--vg-ubuntu--lv: UUID="9aafc4ee-1414-4f13-b560-e491ebea16da" TYPE="ext4"
/dev/sr0: UUID="2021-09-15-20-41-58-00" LABEL="Ubuntu-Server 18.04.6 LTS amd64" TYPE="iso9660" PTUUID="7a857666" PTTYPE="dos"
/dev/vda1: PARTUUID="c9b2df12-5cfa-4e44-a735-3f0bd9e35542"
user@pk-ubuntu:~$
```

Работаем с разделами дисков.

Теперь, когда мы можем посмотреть и узнать имена наших дисков можно поработать и с разделами.

Первым представлю вам команду, которая «визуально» дает представление о том, что делаешь, подойдет тем кто больше предпочитает графический интерфейс.

Команда cfdisk

Для ее использования нужно ввести команду и затем указать с каким диском будем работать

sudo cfdisk имя_диска

Вот пример, результат ввода команды **sudo cfdisk /dev/vda1**



после ввода команды вы выбираете действие из предложенных внизу, последней командой будет Write — она запишет все произведенные изменения.

Команда fdisk

Эта команда содержит множество опций. Выше мы рассматривали один из ее параметров, а именно вывод списка дисков

fdisk - l

Команда fdisk используется в командном режиме, чтоб его запустить нужно набрать команду и указать в конце имя диска с которым хотите работать

fdisk имя_диска

После ввода команды система будет ждать команду на следующее действие



Список команд можно посмотреть, введя букву **«m»**

Все команды представлены в виде одной буквы, вот их список:

a — сделать диск загрузочным

b — редактировать bsd-метку диска

c — установить флаг совместимости с DOS

d — удалить раздел

l — перечислить известные типы разделов

m — вывести это меню

n — создать новый раздел

o — создать новую пустую таблицу разделов DOS

p — вывести таблицу разделов (аналогична fdisk -l)

q — выйти без сохранения

s — создать новую метку диска Sun

t — изменить системный идентификатор раздела

u — изменить единицы отображения/ввода

v — проверить таблицу разделов

w — записать таблицу на диск и выйти

x — дополнительные функции (только для экспертов)

Если например вы хотите создать диск то после первой команды вводим «**n**«

— далее указываем какой это раздел, будут подсказки.

— далее попросят указать номер первого и последнего цилиндра, лучше первый оставить по умолчанию, а второй указать цифрами, какой размер вам нужен, например +50G , прибавится 50 гигабайт.

Размер указывается опциями **K** — килобайты, **M** — мегабайты, **G** — гигабайты

Диск — форматируем и монтируем

Отформатировать диск вы можете командой **mkfs**

sudo mkfs.файловая система /раздел/диск

файловая система как правило **ext4** или **ntfs**

далее указываете диск, например **/dev/vd1**

В результате команда будет выглядеть так: **sudo mkfs.ext4 /dev/vd1**

Создаем новую директорию

Чтоб смонтировать созданный диск вам понадобится вначале создать директорию, в которую вы будете монтировать ваш диск.

Содать директорию поможет команда **mkdir**

sudo mkdir новая_директория

Например, создаем директорию **shara** в директориии **mnt**, команда будет выглядеть так s**udo mkdir /mnt/shara**

Монтируем диск в созданную директорию

Для монтирования применяется команда **mount**

sudo mount имя_диска директория_куда_монтируем

Пример, если монтировать **/dev/vd1** в кдиректорию **/mnt/shara** — команда будет выглядеть так:

sudo mount /dev/vd1 /mnt/shara

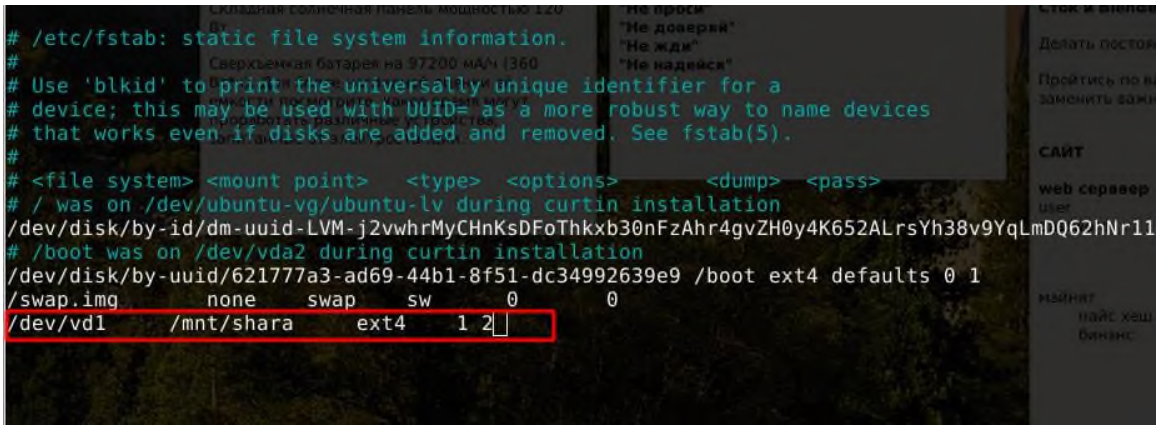
Сохраняем смонтированный диск

Для этого нужно добавить в файл **/etc/fstab** запись

/dev/vd1 /mnt/shara ext4 1 2

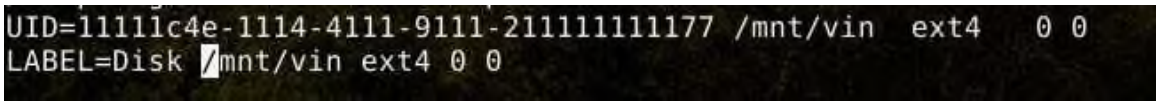
Файл можно открыть в блокноте nano, тогда команда будет выглядеть так

sudo nano /etc/fstab



Разберем состав записи в файле fstab для монтирования диска

— Первым записан диск, который нужно смонтировать, его можно указать не только в таком формате, но и указав его Label или UUID, записи будут выглядеть так



Предпочтительнее указывать UUID, это не даст перепутать диск в системе, в случае если вы его переподключите. например, имена могут поменяться.

— Вторым пунктом записан путь к директории, в которую монтируется диск **/mnt/shara** или **/mnt/vin** — у вас это может быть совсем другая директория, расположенная в совершенно любом месте.

— Третьим указывают файловую систему. В нашем случае это ext4, но может быть любая.

— Предпоследняя цифра это 0, 1 или 2. Обозначают в какую очередь проверять устройство на ошибки. щ- не проверять, 1- в первую, 2 — во вторую.

— Последняя цифра указывает нужно ли делать резервную копию раздела, всегда 0 или 1 (да или нет).

9. Файловые системы

Независимо от назначения операционной системы Linux она имеет четкую структуру расположения файлов. Для пользователей Windows,

недавно перешедших на Линукс, это может вызвать некоторые затруднения, поскольку у таких пользователей, как правило, отсутствует четкое понимание функций каждого каталога. Далее мы подробно разберем структуру файловой системы Linux-дистрибутивов.

/ (корень)

Представляет собой одновременно и корневой каталог, и всю файловую систему на конкретном компьютере. Дело в том, что в Linux отсутствуют привычные для Windows имена дисков. Дерево всех файлов и каталогов на всех носителях начинается именно с корня. От него уже «растут» остальные директории, в том числе ссылающиеся на другие физические диски.

/bin

Данный каталог содержит пользовательские бинарные (запускаемые) файлы. Фактически это программы, готовые для запуска пользователем до подключения директории /usr.

/sbin

Здесь хранятся системные исполняемые файлы, запускаемые в процессе загрузки системы. В этот же каталог сохраняются программы, требующие прав суперпользователя (root).

/etc

Данная директория предназначена для конфигурационных файлов всех установленных в системе программ. Здесь же находятся различные скрипты автозагрузки. Если пользователю необходимо глобально внести изменения в конфигурацию, он вводит пароль суперпользователя и изменяет файлы в этом каталоге.

/dev

Содержит файлы устройств. Важно помнить, что в Linux-системах любое аппаратное устройство (включая периферийное) представлено в виде файла. Поэтому подключенные к компьютеру и обнаруженные системой клавиатуру, мышь, флеш-накопители, жесткие диски следует искать в данной директории. Соответственно, обращаться к ним нужно как к файлам.

/proc

Все происходящие в реальном времени процессы также в виде файлов отображаются в этой директории. За динамическое создание этой подсистемы отвечает ядро Linux. В частности, здесь можно получить информацию об использовании системных ресурсов.

/var

В данный каталог помещаются часто изменяемые файлы — различные базы данных, кэш программ, системные журналы и т. п. В свою очередь

директория содержит важные поддиректории: /var/log (логи); /var/lib (библиотеки); /var/lock (файлы блокировки); /var/mail (почтовые сообщения); /var/spool (файлы печати); /var/run (номера процессов).

/tmp

Здесь расположены временные файлы, созданные как пользователями, так и самой системой. Соответственно, у пользователей есть право изменять содержимое этого каталога. Поскольку файлы временные, при каждой перезагрузке или каждом выключении компьютера они удаляются.

/usr

Это главный каталог всех пользовательских системных ресурсов, как правило, имеющий наибольший размер в системе. Содержит следующие подкаталоги:

- /usr/bin/ (исполняемые файлы пользовательских программ, не участвующих в загрузке системы);
- /usr/sbin/ (исполняемые файлы системных программ, требующих прав суперпользователя для выполнения);
- /usr/lib/ (библиотечные файлы для программ из обоих вышеназванных каталогов);
- /usr/local (пользовательские исполняемые файлы программ, а также необходимые для работы библиотеки и конфигурационные файлы).
- /home

Это домашний каталог пользователя, где в соответствующем пользовательском подкаталоге хранятся все личные данные, локальные файлы настроек и т. д. В Windows есть аналогия в виде личной папки для различных документов. В Linux же при монтировании этой директории на отдельный раздел все содержимое останется в целости и сохранности, если понадобится переустанавливать всю систему.



Структура файловой системы Linux

/boot

Содержит файлы загрузчика системы, образы ядра и временной файловой системы, а также соответствующие конфигурационные файлы.

/lib

Служит для хранения системных библиотек.

/opt

Сюда устанавливаются различные сторонние программы, в том числе несвободные (проприетарные). Это платный софт, игры, драйвера и пр.

/mnt

Предназначен для монтирования носителей или дополнительных файловых систем. Они отображаются в виде подкаталогов.

/media

Сюда также монтируются устройства, но исключительно внешние — USB-накопители, внешние жесткие диски и оптические приводы.

/srv

Это серверный каталог, содержащий в себе файлы, необходимые для работы сервисов и серверов (например, веб-сервера Apache).

/run

В данной директории хранятся временные PID-файлы процессов. Отличие от /var/run заключается в том, что в данном случае создается специальная файловая система TmpFS, поэтому каталог очищается при каждой перезагрузке.

/sys

Содержит базовую информацию о системе, предоставляемой ядром. Для этого формируются соответствующие подкаталоги. Фактически это отдельная файловая система, создаваемая ядром, которая дает возможность управлять множеством системных параметров (например, разделом подкачки swap).

10. Ядро операционной системы Linux

Линус Торвальдс создал именно ядро Linux, что позволило создателю пользоваться всеми возможностями систем Unix, но в некоммерческих целях.

Какие же функции возложены на этот самый главный компонент операционной системы? Итак, ядро:

управляет памятью (выделяет в памяти сегмент для работы программы, очищает память, ограничивает ее);

управляет процессами (отвечает за запуск и закрытие процессов, предоставление им ресурсов);

управляет аппаратной частью компьютера (с помощью встроенных драйверов для определенного оборудования обеспечивает работу операционной системы);

обеспечивает взаимодействие между процессами, службами и программами (дает возможность программному обеспечению нормально функционировать).



Ядро операционной системы Linux

Ядро фактически скрыто от обычного пользователя — оно не отображается в виде каких-то графических элементов, с ним нельзя взаимодействовать напрямую с клавиатуры. Тут можно провести аналогию с человеческим мозгом. Мы не можем им управлять осознанно. Но именно наше «ядро» обеспечивает нормальную работу всего организма.

Среди особенностей операционной системы Linux — ее многоликость. Ведь на основе ядра этой системы создано огромное количество разных и часто внешне непохожих друг на друга дистрибутивов — Debian, Ubuntu, Fedora, Red Hat, Arch Linux и т. д.

Само по себе ядро также бывает трех типов: монолитное, гибридное и микроядро. Рассмотрим каждую разновидность, начиная с последней.

Микроядро (microkernel) представляет собой несколько обособленных модулей, подгружаемых по мере необходимости.

Достоинства:

сравнительно более высокая надежность (в теории);

низкое потребление памяти;

модульность.

Недостатки:

низкий отклик, вызванный постоянными вызовами отдельных модулей;

необходимость в дополнительных драйверах, обеспечивающих доступ к «железу».

Монолитное ядро (monolithic) по своей структуре полностью противоположно микроядру. Оно преимущественно полностью загружается в память компьютера. Это положительно сказывается на скорость работы.

Достоинства:

прямой доступ к «железу»;

простота обмена данными между процессами;

высокий отклик процессов.

Недостатки:

большой размер;

менее безопасная структура в сравнении с микроядром;

требовательность к оперативной памяти.

Гибридное ядро (hybrid) комбинирует в себе характеристики двух предыдущих архитектур. Включает в себя модули, запуск которых в пространстве ядра ускоряет работу операционной системы.

Достоинства:

возможность для пользователя выбрать пространство работы отдельных модулей ядра;

меньший размер в сравнении с монолитным ядром;

гибкость в настройке.

Недостатки:

низкая скорость обработки;

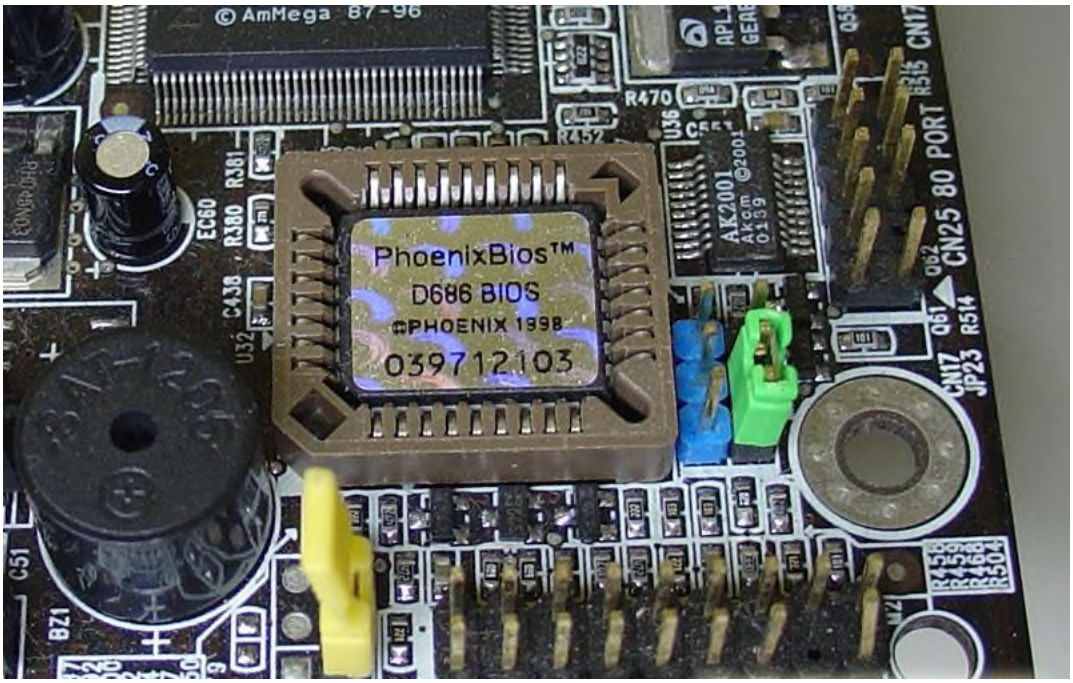
необходимость в фирменных драйверах для аппаратных устройств.

В Linux используется монолитное ядро с некоторыми элементами микроядерной архитектуры. На практике это выражается в том, что операционная система работает полностью в пространстве ядра, а для работы отдельных устройств могут подгружаться дополнительные драйверы.

11. Загрузка ОС linux

BIOS

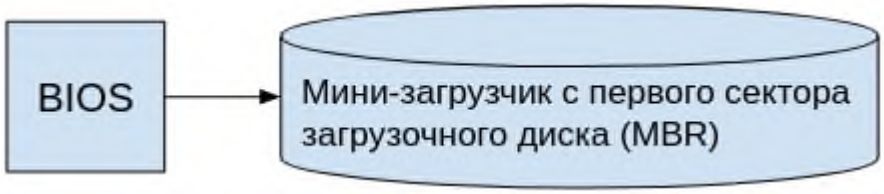
BIOS расшифровывается как **Basic Input/Output System**, что в переводе – **Базовая Система Ввода/Вывода**. Это специальная микросхема, которая находится на материнской плате компьютера.



Когда компьютер только включается, то выполняется код из микросхемы **BIOS.** Вначале этот код выполняет тестирование системы **POST (power-on self test)**, этим тестированием проверяется аппаратная часть системы. А затем из всех дисков определяется загрузочный.

BIOS умеет работать только с дисками размеченными **MBR** способом. На первом секторе такого диска находится так называемый **MBR** (master boot record). Сектор диска имеет размер 512 байт. В эти 512 байт помещается маленький загрузчик и таблица разделов. Кстати, если диск не загрузочный, то вместо загрузчика там может находиться код, который сообщает, что это не загрузочный диск (**No bootable device**).

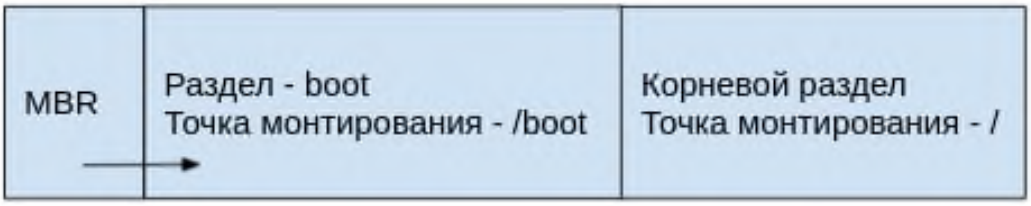
Так вот, **BIOS** передает управление маленькому загрузчику из **MBR** на загрузочном диске.



Загрузчик MBR и GRUB 2

Код загрузчика из MBR очень маленький и он способен лишь найти и запустить следующий загрузчик.

Следующий загрузчик больше и умнее. Бывают разные загрузчики для Linux, но самым популярным является **GRUB 2**. Он позволяет пользователю выбрать операционную систему для загрузки. А также умеет загружать операционные системы с логических разделов. **GRUB 2** расположен на разделе жесткого диска в каталоге **/boot** и довольно часто этот каталог выносят на отдельный раздел.



Давайте посмотрим, что находится в этом каталоге на **Debian 10**:

```
alex@deb-11:~$ ls -l /boot
итого 68836
-rw-r--r-- 1 root root   236056 дек  8 19:21 config-5.10.0-10-amd64
-rw-r--r-- 1 root root   236055 сен 30 22:36 config-5.10.0-9-amd64
drwxr-xr-x 5 root root     4096 дек 20 10:14 grub
-rw-r--r-- 1 root root 28167156 дек 20 10:12 i104
-rw-r--r-- 1 root root 28150099 дек 17 17:57 initrd.img-5.10.0-9-amd64
-rw-r--r-- 1 root root       83 дек  8 19:21 System.map-5.10.0-10-amd64
-rw-r--r-- 1 root root       83 сен 30 22:36 System.map-5.10.0-9-amd64
-rw-r--r-- 1 root root  6841280 дек  8 19:21 vmlinuz-5.10.0-10-amd64
-rw-r--r-- 1 root root  6833568 сен 30 22:36 vmlinuz-5.10.0-9-amd64
```

Ubuntu 20.04 в этом каталоге содержит примерно то же самое, но ядро новее:

```
alex@ubuntu:~$ ls -l /boot/
total 98424
-rw------- 1 root root  4754490 Sep 24 13:50 System.map-5.4.0-89-generic
-rw-r--r-- 1 root root   237884 Sep 24 13:50 config-5.4.0-89-generic
drwxr-xr-x 4 root root     4096 Nov  6 10:47 grub
lrwxrwxrwx 1 root root       27 Nov  6 10:46 initrd.img -> initrd.img-5.4.0-89-generic
-rw-r--r-- 1 root root 83998753 Nov  6 10:54 initrd.img-5.4.0-89-generic
lrwxrwxrwx 1 root root       27 Nov  6 10:46 initrd.img.old -> initrd.img-5.4.0-89-generic
lrwxrwxrwx 1 root root       24 Nov  6 10:46 vmlinuz -> vmlinuz-5.4.0-89-generic
-rw------- 1 root root 11780352 Sep 24 13:35 vmlinuz-5.4.0-89-generic
lrwxrwxrwx 1 root root       24 Nov  6 10:46 vmlinuz.old -> vmlinuz-5.4.0-89-generic
```

Разберем некоторые файлы в этом каталоге. Здесь находится:

- подкаталог с загрузчиком – **grub**;
- ядра разных версий – файлы **vmlinuz**;
- архивы с временной файловой системой и утилитами необходимыми для загрузки ядра – файлы **initrd.img**;
- остальные файлы я рассматривать пока не буду.

После того, как **GRUB 2** получил управление, он загружает **ядро Linux** (файл **vmlinuz**). При этом **GRUB 2** загружает в память файловую систему из файла **initrd.img**. А затем запускает ядро, и при запуске, передаёт ему некие параметры.

Вы можете спросить, зачем вообще нужен образ корневой файловой системы с утилитами **initrd.img**? Он необходим, потому что, когда **GRUB 2** начинает загружать систему **корневая файловая система** еще **не смонтирована**. Поэтому **GRUB 2** использует **initrd** как временную корневую файловую систему, которая загружается из файла в оперативную память.

Initrd – представляет собой файл, содержащий загружаемые модули ядра и минимальный набор утилит для загрузки этих модулей. Он сжимается и загружается в оперативную память с помощью загрузчика. Ядро получает доступ к нему, как если бы была установлена файловая система.

Вы можете использовать команду **lsinitramfs** чтобы перечислить содержимое файла **initrd.img**.

Вот пример для **Debian 11**:

```
alex@deb-11:~$ lsinitramfs -l /boot/initrd.img-5.10.0-10-amd64 | head -n 20
drwxr-xr-x   7 root     root            0 Dec 20 10:12 .
lrwxrwxrwx   1 root     root            7 Dec 20 10:12 bin -> usr/bin
drwxr-xr-x   3 root     root            0 Dec 20 10:12 conf
-rw-r--r--   1 root     root           16 Dec 20 10:12 conf/arch.conf
drwxr-xr-x   2 root     root            0 Dec 20 10:12 conf/conf.d
-rw-r--r--   1 root     root           49 Dec 17 17:57 conf/conf.d/resume
-rw-r--r--   1 root     root         1365 Jan 14  2021 conf/initramfs.conf
drwxr-xr-x   5 root     root            0 Dec 20 10:12 etc
-rw-r--r--   1 root     root            0 Dec 20 10:12 etc/fstab
-rw-r--r--   1 root     root         1622 Dec 20 10:12 etc/ld.so.cache
-rw-r--r--   1 root     root           34 Jul 29  2019 etc/ld.so.conf
drwxr-xr-x   2 root     root            0 Dec 17 17:56 etc/ld.so.conf.d
-rw-r--r--   1 root     root           44 Jul 29  2019 etc/ld.so.conf.d/libc.conf
-rw-r--r--   1 root     root          100 Oct  2 15:47 etc/ld.so.conf.d/x86_64-linux-gnu.conf
drwxr-xr-x   2 root     root            0 Dec 20 10:12 etc/modprobe.d
lrwxrwxrwx   1 root     root           12 Dec 20 10:12 etc/mtab -> /proc/mounts
drwxr-xr-x   2 root     root            0 Dec 20 10:12 etc/udev
-rw-r--r--   1 root     root          305 Feb  2  2021 etc/udev/udev.conf
-rwxr-xr-x   1 root     root         6301 Jan 14  2021 init
lrwxrwxrwx   1 root     root            7 Dec 20 10:12 lib -> usr/lib
```

А вот пример для **Ubuntu 20.04**:

```
alex@ubuntu:~$ lsinitramfs -l /boot/initrd.img | head -n 20
drwxr-xr-x   3 root     root            0 Dec 18  2019 .
drwxr-xr-x   3 root     root            0 Dec 18  2019 kernel
drwxr-xr-x   3 root     root            0 Dec 18  2019 kernel/x86
drwxr-xr-x   2 root     root            0 Dec 18  2019 kernel/x86/microcode
-rw-r--r--   1 root     root        30546 Dec 18  2019 kernel/x86/microcode/AuthenticAMD.bin
drwxr-xr-x   2 root     root            0 Apr 25  2021 kernel
drwxr-xr-x   2 root     root            0 Apr 25  2021 kernel/x86
drwxr-xr-x   2 root     root            0 Apr 25  2021 kernel/x86/microcode
drwxr-xr-x   2 root     root            0 Apr 25  2021 kernel/x86/microcode/.enuineIntel.align.0123456789abc
-rw-r--r--   1 root     root      4609024 Apr 25  2021 kernel/x86/microcode/GenuineIntel.bin
drwxr-xr-x   9 root     root            0 Nov  6 13:04 .
-rw-r--r--   1 root     root           29 Nov  6 13:04 .random-seed
lrwxrwxrwx   1 root     root            7 Nov  6 13:04 bin -> usr/bin
drwxr-xr-x   3 root     root            0 Nov  6 13:04 conf
-rw-r--r--   1 root     root           16 Nov  6 13:04 conf/arch.conf
drwxr-xr-x   2 root     root            0 Nov  6 13:04 conf/conf.d
-rw-r--r--   1 root     root         1807 Feb 15  2020 conf/initramfs.conf
-rw-r--r--   1 root     root           73 Nov  6 13:04 conf/modules
drwxr-xr-x   2 root     root            0 Nov  6 13:04 cryptroot
-rw-r--r--   1 root     root            0 Nov  6 13:04 cryptroot/crypttab
```

Ядро Linux и процесс инициализации

Ядро при запуске использует временную файловую систему из файла **initrd.img** для того, чтобы загрузиться полностью.

А дальше ядро запускает процесс инициализации операционной системы. Процесс инициализации является первым процессом в системе, так как ядро запускалось не как процесс. Каким именно будет процесс инициализации зависит от используемой системы инициализации. Существуют разные системы инициализации, но в **Debian 11** и **Ubuntu 20.04** используется система инициализации под названием **SystemD**.

То есть, первым процессом при запуске этих двух систем является **systemd**. Чтобы это проверить выполним следующие две команды, которые на обоих системах выполняются одинаково:

```
alex@deb-11:~$ ps -p 1
    PID TTY          TIME CMD
      1 ?        00:00:01 systemd

alex@deb-11:~$ ls -l /sbin/init
lrwxrwxrwx 1 root root 20 июл 13 20:29 /sbin/init -> /lib/systemd/systemd
```

Первой командой мы смотрим, какой процесс в системе работает под номером 1 – это **systemd**. Второй командой смотрим на что ссылается файл, который традиционно считается файлом системы инициализации – а ссылается он на **/lib/systemd/systemd**.

И наконец, система инициализации запускает все остальные процессы в системе.

Итог по загрузке с BIOS

BIOS	Код в микросхеме на материнской плате. Позволяет настраивать некоторые особенности работы компьютера. А для загрузки системы использует MBR на жестком диске
MBR	Первый сектор на диске, который использует таблицу разделов MBR. Имеет размер 512 байт. Хранит загрузчик и таблицу разделов. Этот загрузчик способен запустить более функциональный загрузчик
GRUB 2	Более функциональный загрузчик, который способен загрузить ядро системы (Linux или Windows). Имеет собственную командную строку и множество настроек
Kernel	Ядро Linux. Оно запускается не как процесс. После загрузки оно запускает систему инициализации как первый процесс в системе
Init	Система инициализации. Это первый процесс в системе, именно он запускает все остальные процессы

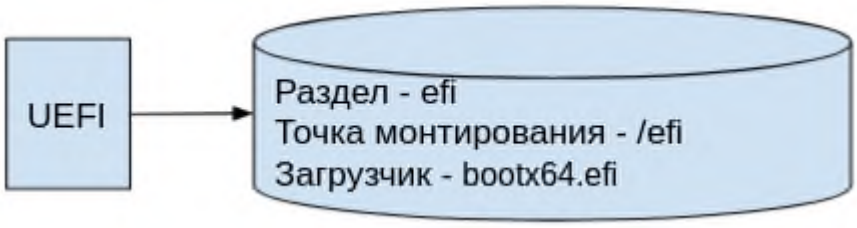
Алгоритм загрузки linux (UEFI)

Сейчас на смену **BIOS** пришёл **UEFI (Unified Extensible Firmware Interface)**. **UEFI** содержит множество преимуществ перед **BIOS**, одна из которых — это поддержка **GPT**.

Раньше диски разбивались на разделы с помощью **MBR**, при этом диск не мог быть больше 2 ТВ, а разделов не могло быть больше четырёх. При этом таблица разделов и первичный загрузчик помещались на первый сектор диска в **MBR**. А **GPT** позволяет использовать очень большие диски и помещает загрузчик на специальный раздел, а также позволяет создавать очень много разделов.

Диск, разбитый на разделы с помощью таблицы разделов **GPT** в первом секторе, продолжает хранить **MBR** запись. Это делается для того, чтобы старые системы с **BIOS** могли использовать **GPT диски**.

UEFI не использует загрузчик из **MBR**, вместо этого используется загрузчик со специального раздела на диске, который отформатирован в **FAT32** и называется **ESP** или **EFI**. Это специальный загрузчик от **UEFI**. Обычно это файл **/efi/boot/bootx64.efi**.



Secure Boot

UEFI имеет специальный режим, который называется **Secure Boot**. Если этот режим включен, то загрузчик без специальной подписи не будет работать. Windows и некоторые системы Linux имеют такие подписи, а для других систем приходится этот режим отключать.

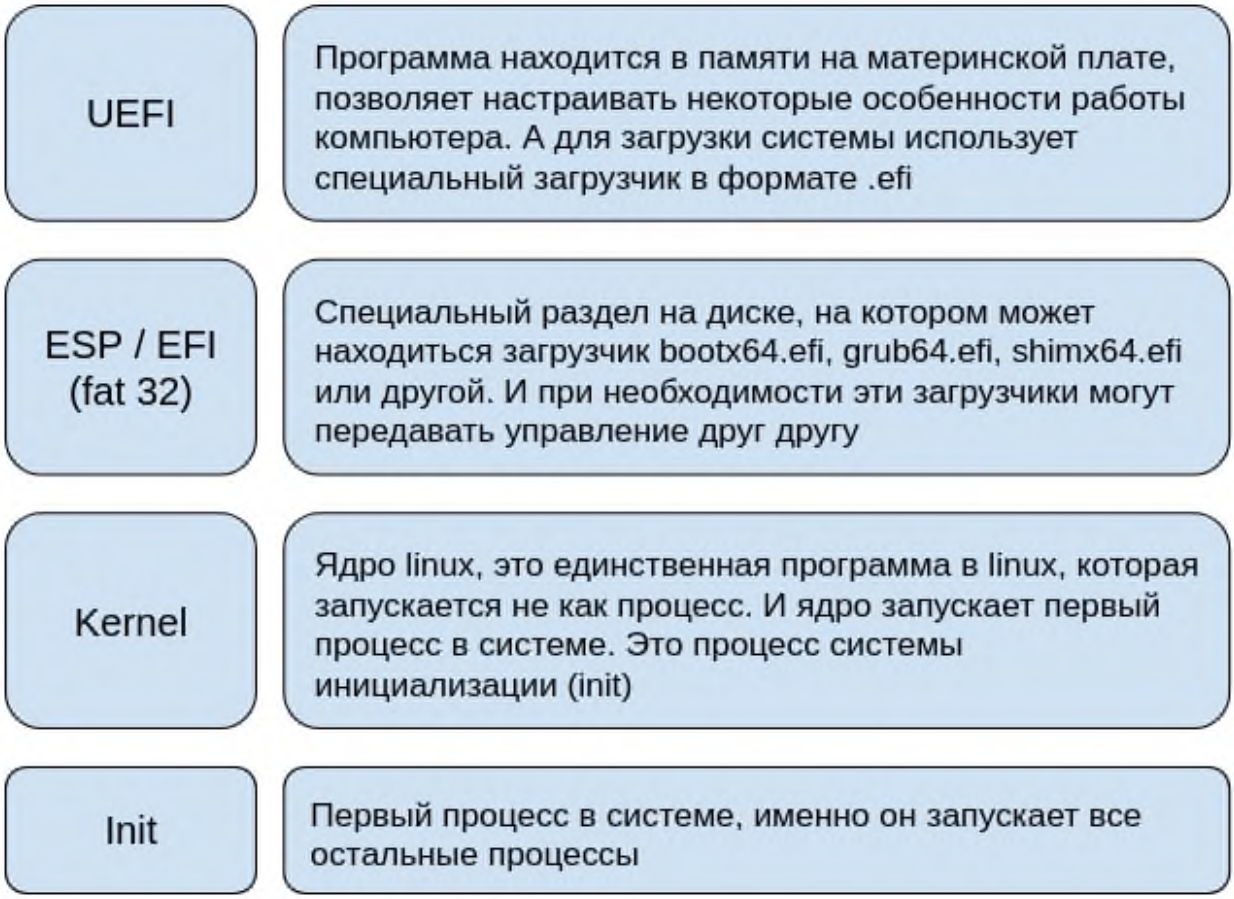
Или можно не отключать **Secure Boot**, а просто очистить ключи производителя и добавить туда ключи от необходимого загрузчика. Например, в **Ubuntu** – **GRUB 2** подписан подписью от **Canonical**, именно эту подпись и следует добавить в **Secure Boot**.

Дальнейшая загрузка

А дальше все происходит так же, как и в системах с **BIOS**. Загрузчик запускает **GRUB 2**, хотя может сразу запустить и **ядро Linux**, но тогда мы теряем гибкость **GRUB 2**.

GRUB 2 в свою очередь запускает **ядро Linux**. Дальше **ядро** запускает **систему инициализации**. А уже **система инициализации** запускает все остальные процессы в системе.

Итог по загрузке с UEFI



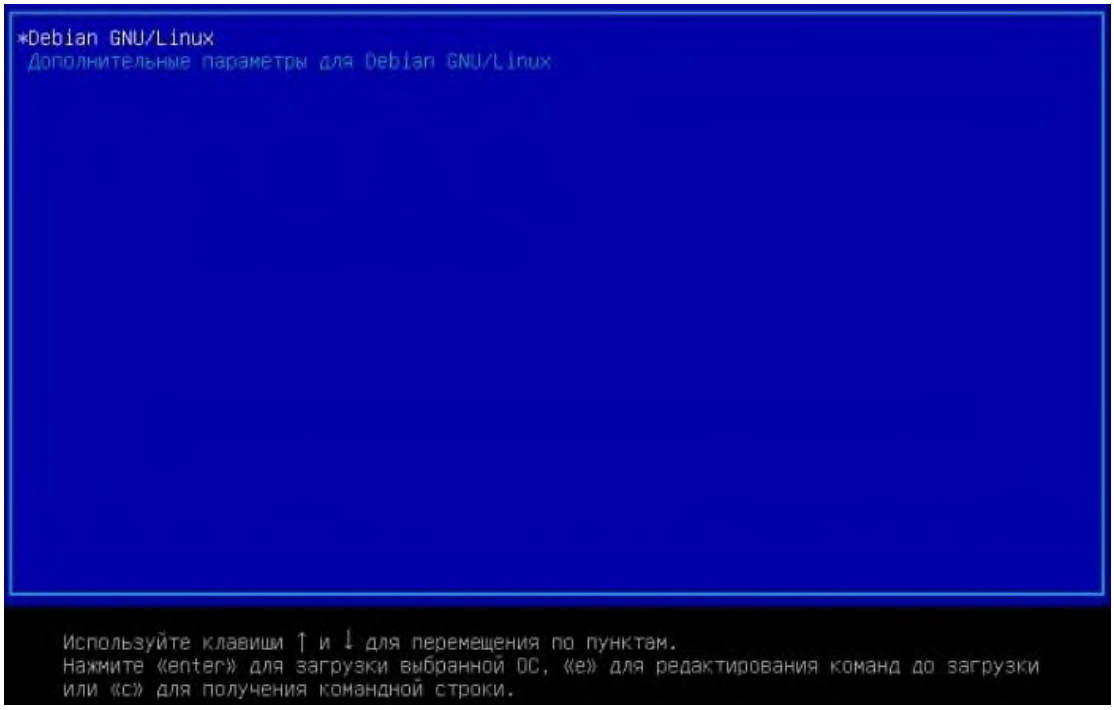
Немного про GRUB 2

GRUB 2 в настоящее время является основным загрузчиком для большинства систем Linux. Он позволяет запустить не только **ядро Linux**, но и **Windows** систему. Он имеет собственную командную строку с помощью, которой можно, например устранить некоторые ошибки при загрузке.

Существует две версии загрузчика **GRUB 2**: для **BIOS** и для **UEFI**. Но для **UEFI** он не имеет цифровой подписи для режима **Secure Boot**.

Ubuntu, кстати, поддерживает **Secure Boot**. Для этого используется загрузчик **Shim** подписанный подписью от **Microsoft**. И этот загрузчик запускает **GRUB 2**, который подписан цифровой подписью от **Canonical**.

Взаимодействуя с **GRUB 2** при старте системы вы можете выбрать, какую из возможных конфигураций загрузить или изменить:



Когда отображается меню, вы можете нажать:

- “**Enter**” – чтобы немедленно загрузить выбранную строку меню;
- любую другую клавишу — чтобы остановить тайм-аут;
- “**e**” – чтобы отредактировать запись;
- “**c**” – чтобы войти в командную строку GRUB2.

Если ваш тайм-аут установлен на 0, то **GRUB 2** немедленно загружает систему, не показывая подобное меню.

12. Типы дистрибутивов Linux

Ранее мы упомянули, что сегодня существует очень много дистрибутивов Linux под самые разные нужды. Перед пользователем возникает проблема выбора оптимальной системы среди такого разнообразия. И здесь в первую очередь нужно учитывать два главных критерия:

Популярность дистрибутива

Чем востребованнее система среди пользователей, тем больше ее комьюнити. А большое сообщество означает доступность и полноценность руководства по данному дистрибутиву. Решать возникающие проблемы можно с помощью форумов, которые для популярных систем весьма посещаемы и действительно полезны.

Кроме того, востребованность дистрибутива стимулирует создание программного обеспечения именно под этот дистрибутив. Для новичка это оптимальный вариант, ведь проще использовать готовые решения, чем самостоятельно компилировать нужные программы из

исходных кодов, как это часто требуется в экзотических и малоизвестных системах.

Команда мейнтейнеров

Желательно, чтобы дистрибутив развивала большая группа разработчиков. Например, в Ubuntu этим занимается крупная компания Canonical, а Fedora и SuSE также поддерживаются весьма многолюдными сообществами.

Следует также помнить, что для каждого, даже популярного дистрибутива, можно найти менее популярный аналог, не уступающий тому в функционале.

Лучшие дистрибутивы операционной системы Linux

Семейство Linux образует полноценную альтернативу другим операционным системам. Здесь рассмотрим наиболее популярные и объективно лучшие дистрибутивы общего назначения.

Отмеченные ниже современные операционные системы Linux универсальны и имеют низкий порог вхождения.

Первое качество означает возможность адаптации ОС под любые задачи. Перед создателями дистрибутивов, как правило, не стоит цель сделать акцент на том или ином функционале. Пользователям предлагается просто многофункциональная платформа наподобие Windows или MacOS.

Под низким порогом вхождения подразумевается простота установки операционной системы Linux и дальнейшего освоения этой системы. В этом смысле показательной является Ubuntu, установить и освоить которую не сложнее, чем Windows. Дружелюбность к новичкам можно назвать главным достоинством таких систем.

Manjaro

На первое место, тем не менее, мы поместили не Ubuntu, а Manjaro. Данная ОС более «заточена» под пользователей, переходящих с Windows и только приступивших к знакомству с Linux.

Дистрибутив основан на Arch Linux — системе, ориентированной на опытных пользователей. Однако для повседневного использования новичками Manjaro вполне годится, не теряя при этом в качестве сборки.

Создатели этой системы изначально задумывали проект для бесплатного и свободного использования людьми, переходящими с других систем. Поэтому перед разработчиками стояла главная задача сделать этот переход максимально легким. С учетом данного требования пользователям предлагается на выбор несколько графических оболочек. Своим внешним

видом они напоминают рабочие столы в Windows и MacOS и подойдут для разных по производительности компьютеров.

Среди большого набора предустановленного ПО имеется офисный пакет, комплект мессенджеров, браузер, программы для запуска Windows-приложений и даже игр, разработанных под Windows. Поэтому при желании Manjaro могут использовать и заядлые геймеры.

Ubuntu

Одна из известнейших операционных систем Linux Ubuntu прославилась тем, что явилась первой системой для простых пользователей, ищущих свободную и бесплатную альтернативу Windows.

Дистрибутив сразу стал примечателен дружественным интерфейсом, бесплатным программным обеспечением и мощной основой в виде пакетной базы Debian. Это позволило Ubuntu выбиться в лидеры среди Linux-систем.

Данная система также отличается многогранностью, поскольку выпускается сразу в нескольких графических вариантах:

- Классическая Ubuntu изначально строилась на базе окружения Gnome, затем использовала собственную оболочку Unity, а впоследствии вернулась на модифицированную версию Gnome 3.
- Kubuntu предлагает пользователю тот же набор ПО, но строится уже на основе оболочки KDE.
- Xubuntu позиционируется как облегченная версия Ubuntu, поскольку здесь используется минималистичное окружение XFCE.
- Lubuntu сходна по потреблению ресурсов с Xubuntu, однако разрабатывается на основе пакетов под KDE.
- Ubuntu LTS визуально не отличается от обычной Ubuntu. Главное отличие состоит в увеличенном цикле поддержки Canonical.
- Ubuntu Server представляет собой серверную ОС.
- Ubuntu Studio — это любопытный вариант для занятия творчеством, включающий в себя в базовом комплекте все необходимые инструменты.

Итак, выбор только лишь в рамках данного дистрибутива достаточно большой, учитывающий самые разные потребности и аппаратные возможности. Во всех вариантах установка и настройка операционной системы Linux не должны здесь доставить проблем новичкам. И этим качеством Ubuntu похожа на Manjaro.

Linux Mint

На базе Ubuntu в свою очередь строятся и другие дистрибутивы, один из которых — Linux Mint. По замыслу создателей данная система использует

альтернативный подход к интерфейсу. А с точки зрения пакетной базы и технических особенностей Mint напоминает Ubuntu.

Первые версии Linux Mint выглядели легче Ubuntu как визуально, так и с точки зрения потребления аппаратных ресурсов. Позднее создатели дистрибутива разработали собственную оболочку Cinnamon, которая сделала систему заметно тяжелее, но и вместе с тем функциональнее. Разумеется, такие характерные качества, как безопасность операционной системы Linux, надежность и стабильность в работе, здесь сохранены.

Возникает вопрос: чем выгодно отличается Mint от Ubuntu? Наиболее важное отличие состоит в разных интерфейсах. Если в Ubuntu предлагаются классические десктопы Gnome 3, KDE и XFCE, то разработчики Linux Mint разнообразили выбор, предложив оболочки Mate (создается на базе Gnome 2) и собственную Cinnamon. Ранее также присутствовала версия с KDE, но позднее от нее решили отказаться.

Elementary OS

Это еще одно ответвление от Ubuntu. Создатели позиционировали свое детище как облегченный вариант Ubuntu для слабых компьютеров. В итоге система все равно «потяжелела», не потеряв при этом в функциональности.

По внешнему виду понятно, что за основу интерфейса взят MacOS. Результатом стала действительно красивая и удобная в использовании система. Разработчики даже позаимствовали стандартные «эпловские» жесты для трекпада, внедрив их сюда.

Кроме того, очевидным сходством с MacOS является наличие магазина приложений, похожего на AppStore. И этим же Elementary OS отличается от других современных дистрибутивов Linux. К работе над системой и нативными программами усиленно привлекались энтузиасты. Специально для этого была создана платформа, где каждый желающий разработчик мог бы размещать и продавать свои программы. Данная мера позволила Elementary OS обзавестись собственными полезными приложениями, которых зачастую нет в других дистрибутивах.

К интересным и уникальным для этой системы утилитам можно отнести редактор заметок, менеджер паролей и другие инструменты. Но ощутимым недостатком такого подхода является платность всех этих продуктов.

Jing OS

От других операционных систем семейства Linux этот дистрибутив отличается оформлением под iPad. Jing OS предназначена в первую очередь для устройств с сенсорным экраном. Визуально система действительно похожа на iOS. Кроме того, она прекрасно работает с жестами.

Здесь используется созданная полностью с нуля графическая оболочка, оптимизированная под мобильные гаджеты. В первой версии дистрибутива поддерживались Surface Pro 6, Huawei Matebook 14 и некоторые другие устройства с сенсорным экраном.

В комплекте предустановленного ПО имеется большой набор приложений, разработанных под сенсорное управление пальцами. Сюда относятся календарь, файловый менеджер, калькулятор, различные утилиты для повседневных задач.

Несмотря на все особенности, это всё тот же Линукс — с ядром, базовыми компонентами, поддержкой стандартных Linux-приложений, возможностью управления привычным способом (клавиатурой и мышью). Поэтому Jing OS можно считать полноценной универсальной системой для новейших сенсорных устройств, но с поддержкой устаревших моделей.

Linux Lite

Этот дистрибутив, как и многие другие, задумывался для постепенной миграции пользователя с Windows на Linux. Отличие заключается в том, что данная система рассчитана прежде всего на слабые в аппаратной части компьютеры.

Подобно другим дистрибутивам общего назначения Linux Lite создавался для обычных пользователей, не обладающих навыками разработки. Поэтому в комплект ПО здесь также включены офисный пакет, веб-браузер, почтовый клиент, проигрыватели медиафайлов и т. д.

Создатели дистрибутива подчеркивают: низкие системные требования Linux Lite положительно влияют на производительность игр. Эта система придется по вкусу всем геймерам, в том числе использующим виртуальные машины.

В качестве основы здесь используется Ubuntu LTS, что автоматически обеспечивает системе длительный срок поддержки, стабильную работу программ, а также доступ к огромному репозиторию Ubuntu.

13. Управление пакетами

Самые современные Unix-подобные операционные системы предлагают централизованные механизмы для поиска и установки пакетов. Программное обеспечение, как правило, распространяется в виде пакетов, которые хранятся в репозиториях. Пакеты содержат основные компоненты операционной системы, а также общие библиотеки, приложения, услуги и документацию.

Система управления пакетами, помимо установки программного обеспечения, предоставляет множество других функций и инструментов для

модернизации уже установленных пакетов. Репозитории пакетов гарантируют, что код прошел необходимую проверку для использования в системе, и что установленные версии программного обеспечения были одобрены разработчиками.

При настойке сервера или окружения разработки лучше искать пакеты в официальных репозиториях. Пакеты в стабильной версии дистрибутива могут оказаться устаревшими, особенно если программное обеспечение быстро обновляется.

Управление пакетами является жизненно важным навыком для системных администраторов и разработчиков.

Данный модуль охватывает основы управления пакетами: поиск, установку и обновление пакетов для различных дистрибутивов.

Системы управления пакетами: краткий обзор

Большинство систем управления пакетами строятся на наборах файлов пакетов. Файл пакета – это, как правило, архив, который содержит скомпилированные бинарные файлы, скрипты установки и другие ресурсы, составляющие программу. Также пакеты содержат ценные метаданные, в том числе их зависимости (список пакетов, необходимых для запуска программы).

Операционная система	Формат
Debian	.deb
Ubuntu	.deb
CentOS	.rpm
Fedora	.rpm
FreeBSD	порты, .txz

CentOS, Fedora и другие системы Red Hat используют файлы RPM. В CentOS для взаимодействия с пакетами и репозиториями используется менеджер yum. В последних версиях Fedora yum был заменён модернизированным менеджером dnf.В системе Debian и основанных на ней системах (Ubuntu, Linux Mint, Raspbian) используется формат .deb. Пакетный менеджер APT (Advanced Packaging Tool) предоставляет команды, используемые для наиболее распространенных операций: поиска репозиториев, управления обновлениями, установки набора пакетов и их зависимостей. Команды APT работают как фронтэнд утилиты нижнего уровня dpkg, которая обрабатывает установку индивидуальных пакетов .deb на локальную систему; при необходимости эту утилиту можно вызывать явно.

Системой бинарных файлов FreeBSD управляет команда pkg. Кроме того, FreeBSD предоставляет коллекцию портов, локальную структуру каталогов и инструментов, которые позволяют извлекать, компилировать и устанавливать пакеты из исходного кода с помощью make-файлов. Обычно удобнее пользоваться менеджером pkg, но иногда предварительно скомпилированные пакеты недоступны.

Обновление списка пакетов

Большинство систем содержит локальную базу данных пакетов, доступных в удаленных репозиториях. Перед установкой или обновлением пакета рекомендуется обновить эту базу данных. Частичным исключением из этого шаблона являются yum и DNF, которые проверяют наличие обновлений пакетов перед выполнением некоторых операций.

Операционная система	Команда
Debian / Ubuntu	sudo apt-get update
CentOS	yum check-update
Fedora	dnf check-update
FreeBSD Packages	sudo pkg update
FreeBSD Ports	sudo portsnap fetch update

Обновление установленных пакетов

Поддержка установленного программного обеспечения в актуальном состоянии без помощи системы управления пакетами – довольно трудоёмкое задание. Вам придется самостоятельно отслеживать обновления пакетов и извещения безопасности для сотен разных пакетов. Конечно, пакетный менеджер не решит всех проблем, которые могут возникнуть при обновлении программного обеспечения, но он поможет поддерживать большинство компонентов системы в актуальном состоянии при помощи всего нескольких команд.

В системе FreeBSD обновление установленных портов может потребовать ручной настройки. Прежде чем обновлять portmaster, рекомендуется ознакомиться с /usr/ports/UPDATING.

Система	Команда
Debian / Ubuntu	sudo apt-get upgrade
	sudo apt-get dist-upgrade
CentOS	sudo yum update
Fedora	sudo dnf upgrade
FreeBSD Packages	sudo pkg upgrade
FreeBSD Ports	less /usr/ports/UPDATING
	cd /usr/ports/ports-mgmt/portmaster && sudo make install && sudo portmaster -a

Поиск пакетов

Большинство дистрибутивов предоставляет графический или управляемый в режиме меню интерфейс для наборов пакетов. Он позволяет просматривать категории и искать новое программное обеспечение. Однако чаще всего самым быстрым и самым эффективным способом поиска пакетов является поиск с помощью инструментов командной строки.

Система	Команда
Debian / Ubuntu	apt-cache search search_string

CentOS	yum search search_string
	yum search all search_string
Fedora	dnf search search_string
	dnf search all search_string
FreeBSD (пакеты)	pkg search search_string
	pkg search -f search_string
	pkg search -D search_string
FreeBSD (порты)	cd /usr/ports && make search name=package
	cd /usr/ports && make search key=search_string

Информация о пакетах

Принимая решение, как пакет установить, обратитесь за помощью к подробному описанию пакетов. Этот удобочитаемый текст часто содержит полезные метаданные: номера версий, список зависимостей и т. п.

Система	Команда
Debian / Ubuntu	apt-cache show package
	dpkg -s package
CentOS	yum info package
	yum deplist package
Fedora	dnf info package
	dnf repoquery –requires package
FreeBSD (пакеты)	pkg info package
FreeBSD (порты)	cd /usr/ports/category/port && cat pkg-descr

Установка пакетов из репозиториев

Зная имя необходимого пакета, его можно очень быстро установить при помощи одной команды. Также в одной команде можно перечислить все пакеты, которые нужно установить.

Система	Команда
Debian / Ubuntu	sudo apt-get install package
	sudo apt-get install package1 package2 …
	sudo apt-get install -y package
CentOS	sudo yum install package
	sudo yum install package1 package2 …
	sudo yum install -y package
Fedora	sudo dnf install package
	sudo dnf install package1 package2 …
	sudo dnf install -y package
FreeBSD (пакеты)	sudo pkg install package
	sudo pkg install package1 package2 …
FreeBSD (порты)	cd /usr/ports/category/port && sudo make install

Установка пакета из локальной файловой системы

Иногда разработчики или поставщики предлагают пакетные файлы для скачивания, даже если приложение официально не упаковано для данной операционной системы.

Как правило, такие пакеты можно загрузить при помощи веб-браузера или утилиты curl в командной строке. После загрузки пакета его можно установить с помощью одной команды.

В Debian-подобных системах индивидуальными файлами пакетов управляет dpkg. Если пакет имеет зависимости, которых не установлены в системе, gdebi может загрузить их из официального репозитория.

В системах CentOS и Fedora для установки отдельных файлов и обработки необходимых зависимостей используется yum и dnf.

Система	Команда
Debian / Ubuntu	sudo dpkg -i package.deb
	sudo apt-get install -y gdebi&& sudo gdebi package.deb
CentOS	sudo yum install package.rpm
Fedora	sudo dnf install package.rpm
FreeBSD (пакеты)	sudo pkg add package.txz
	sudo pkg add -f package.txz

Удаление пакетов

Менеджер пакетов знает, какие именно файлы предоставляет каждый конкретный пакет, потому с его помощью можно легко удалить все ненужные программы.

Система	Команда
Debian / Ubuntu	sudo apt-get remove package
	sudo apt-get autoremove
CentOS	sudo yum remove package
Fedora	sudo dnf erase package
FreeBSD (пакеты)	sudo pkg delete package
	sudo pkg autoremove
FreeBSD (порты)	sudo pkg delete package
	cd /usr/ports/path_to_port && make deinstall

Получение справки

Кроме веб-документации системы Unix предоставляют удобный справочник, в котором можно найти информацию о большинстве команд оболочки. Чтобы открыть мануал, введите:

man page

В справочнике можно перемещаться при помощи кнопок со стрелками. Нажмите /, чтобы искать по тексту страницы, и q, чтобы выйти.

Система	Команда	Примечания
Debian / Ubuntu	man apt-get	Обновление локальной базы данных пакетов и работа с пакетами.
	man apt-cache	Запрос локальной базы данных пакетов.
	man dpkg	Управление индивидуальными файлами пакетов и запрос установленных пакетов.
CentOS	man yum	
Fedora	man dnf	
FreeBSD пакеты	man pkg	Управление предварительно скомпилированными файлами пакетов.
FreeBSD порты	man ports	Управление коллекцией портов.

В операционной системе Linux и других системах семейства Unix после завершения загрузки ядра начинается инициализация Linux системы, сервисов и других компонентов. За это отвечает процесс инициализации, он запускается ядром сразу после завершения загрузки, имеет PID 1, и будет выполняться пока будет работать система.

Процесс инициализации запускает все другие процессы, которые должны быть запущены, это родительский процесс для всего, что выполняется в системе. Другие процессы могут тоже создавать дочерние процессы, но если родительский процесс завершается, для его дочерних процессов родительским становится процесс инициализации.

14. Инициализация системы. Systemd, init-v

За время развития операционных систем были созданы различные системы инициализации Linux. В разных дистрибутивах использовались разные системы. Мы рассмотрим лучшие системы инициализации, которые вы можете сейчас использовать. Начнем с более старых систем с меньшим функционалом, чтобы понять с чего все начиналось, затем подойдем к более новым, созданным в последнее время.

SYSTEM V INIT

System V или SysV — это довольно старая, но до сих пор ещё популярная система инициализации Linux и Unix подобных операционных систем. Она была основой для создания многих других систем инициализации, а также первой коммерческой системой инициализации разработанной для Unix в AT&T. Она была разработана еще в 1983 году.

Почти все дистрибутивы Linux изначально использовали SysV. Исключением была только Gentoo, в которой использовалась собственная система инициализации и Slackware, с инициализацией в стиле BSD.

Основные возможности SysV:

- Написание файлов запуска служб на bash;
- Последовательный запуск служб;
- Сортировка порядка запуска с помощью номеров в именах файлов;
- Команды для запуска, остановки и проверки состояния служб.

Никакой параллельной загрузки, системы зависимостей, запуска по требованию и автоматического запуска здесь не было и в помине.

С момента ее разработки прошло много лет и из-за некоторых недостатков были разработаны другие системы для ее замены, они хоть и имели новые функции и были более эффективны, но они были по-прежнему совместимы с SysV.

OPENRC

OpenRC — это система инициализации Linux и Unix подобных операционных систем совместимая с Sys V Init и поддерживающая систему зависимостей во время запуска. Она приносит некоторые улучшения в SysV, и как и другие системы инициализации Linux, совместима с ней, но вы должны иметь в виду, что OpenRC не заменяет полностью файл /sbin/init. Эта система инициализации используется в Gentoo и дистрибутивах BSD.

Кроме стандартных возможностей SysV, здесь поддерживается также:

- Поддержка зависимостей служб;
- Поддержка параллельного запуска служб;
- Поддерживает настройку в одном отдельном файле;
- Работает как демон;

По сравнению с SysV тут появилось много новых возможностей, но все еще не все те, что нужны для оптимальной работы системы.

SYSTEMD

Systemd — это новая система инициализации Linux. Она была введена по умолчанию в Fedora 15, а сейчас используется почти во всех популярных Linux дистрибутивах. Это не только инициализирующий процесс, поддерживающий огромное количество возможностей, но и набор инструментов для управления службами и этими возможностями из системы. Основная цель - иметь полный контроль над всеми процессами во время их запуска и на протяжении всего выполнения.

Systemd очень сильно отличается от всех существующих систем инициализации, тем как она работает с сервисами, и даже

конфигурационными файлами сервисов. Совместимости со скриптами SysV нет, их нужно преобразовать в linux systemd unit файлы.

Вот ее основные особенности:

- Понятный, простой и эффективный дизайн;
- Параллельная загрузка служб на основе зависимостей;
- Поддерживается завершение дополнительных процессов;
- Поддерживается собственный журнал с помощью journald;
- Поддерживается планирование заданий с помощью таймеров Systemd;
- Поддерживается управление сетью с помощью networkd;
- Для управления DNS используется systemd-resolved;
- Хранение журналов в бинарных файлах;
- Сохранение состояния сервисов linux systemd для возможного восстановления;
- Улучшенная интеграция с Gnome;
- Запуск сервисов по требованию;

RUNINIT

Runinit — это кроссплатформенная система инициализации, которая может работать в GNU Linux, Solaris, BSD и MacOS. Это отличная альтернатива для SysV с поддержкой мониторинга состояния служб.

Здесь есть некоторые интересные особенности, которых нет в других системах инициализации:

- Полный контроль сервисов, каждый сервис привязывается к своему каталогу;
- Надежное средство журналирования и ротации логов;
- Быстрая система загрузки и выключения;
- Портативность;
- Легкое создание файлов конфигурации служб;
- Небольшое количество кода системы инициализации.

UPSTART

Upstart — это система инициализации на основе событий, разработанная в Canonical и призванная заменять SysV. Она может запускать системные службы, выполнять над ними различные задачи, инспектировать их во время выполнения, а также выполнять нужные действия в ответ на события в системе.

Это гибридная система инициализации, она использует как SysV скрипты запуска, так и файлы служб Systemd. Вот ее самые заметные особенности:

- Изначально разработанная для Ubuntu, но может использоваться и в других дистрибутивах;

- Запуск и остановка служб на основе событий;
- Генерация событий во время запуска и остановки служб;
- События могут быть отправлены обычными процессами;
- Связь с процессом инициализации через DBus;
- Пользователи могут запускать и останавливать свои процессы;
- Перезапуск служб, которые неожиданно завершились;
- Параллельная загрузка сервисов;
- Автоматический перезапуск служб;

Большинство ее возможностей работают благодаря интеграции с системой инициализации Systemd. В последнее время всё меньше используются скрипты SysV init и всё больше применяются юнит файлы Systemd. Рано или поздно Systemd вытеснит и полностью заменит Upstart в Ubuntu.

Выводы

Система инициализации запускает и управляет всеми другими процессами в системе Linux. SysV до недавнего времени была основной системой инициализации в большинстве дистрибутивов Linux, но из-за некоторых своих недостатков для нее было разработано несколько замен, в том числе Systemd.

15. Управление пользователями

Пользователи окружают нас всюду, в современных информационных системах это одно из ключевых понятий, вокруг которого строится вся система разграничения прав доступа и безопасности. Поэтому умение управлять пользователями и группами — это один из основных навыков необходимых любому администратору. Базовые механизмы управления пользователями и доступом в Linux достаточно просты и некоторым могут показаться грубыми, но нельзя осваивать более сложные механизмы, не обладая базовыми навыками, поэтому давайте начинать изучение этой непростой темы с самых ее азов.

Чтобы понять современную систему пользователей в Linux сделаем небольшой экскурс в историю. Как известно, Линус Торвальдс создавал ОС по образу и подобию UNIX и поэтому в основе Linux лежат многие простые и изящные решения этой системы. Одна из замечательных абстракций UNIX — это все есть файл, т. е. файлами также являются устройства, процессы, сетевые сокеты и т.д. и т.п. Это позволяет делать при помощи достаточно простых инструментов довольно сложные вещи, например, чтобы сделать образ диска достаточно просто скопировать файл блочного устройства в файл образа, а для

работы с последовательным портом достаточно прочитать или записать что-либо в файл устройства ввода-вывода.

У каждого файла обязательно есть владелец и группа, которой он принадлежит. На этой схеме строится классическая система прав в Linux и UNIX: владелец, группа и остальные.

Таким образом пользователи и группы определяют доступ не только к файлам в привычном нам понимании, но и ко всему спектру объектов операционной системы: процессам, устройствам, сокетам. Просто запомните, что все есть файл, а каждый файл имеет владельца и группу.

Условно всех пользователей можно разделить на специальных, служебных и обычных, но повторимся еще раз — деление это условное, все различие между ними заключается в предоставляемых им правах и некоторых иных возможностях. Чтобы получить список пользователей системы откроем файл /etc/passwd.

Давайте рассмотрим его структуру подробнее. Каждая строка соответствует одному пользователю и имеет следующий формат:

```
имя пользователя:пароль:UID:GID:комментарий:домашняя директория:командная оболочка
```

С именем пользователя все понятно, а вот второй параметр может вызвать некоторое недоумение. Когда-то давно пароли хранились в открытом виде, о чем намекает само название файла, но затем от этой практики отказались и современные системы не хранят пароли в каком-либо виде вообще, хранится только сформированный по специальному алгоритму хеш, такие пароли в Linux называются затененными (shadow). Для такого пароля во втором поле всегда ставится символ x.

UID (User identifier) и GID (Group identifier) — числовые идентификаторы пользователя и группы. Важно понимать, что система различает пользователей именно по UID, а не по наименованиям и многие системы имеют возможность создать несколько пользователей с одинаковым UID, с точки зрения системы это будет один и тот же пользователь.

Этим часто пользуются злоумышленники, один из типовых сценариев проникновения предусматривает создание пользователя с неприметным именем, но идентификаторами root, что позволяет действовать в системе от имени суперпользователя не привлекая ненужного внимания.

В реальных сценариях данную возможность использовать не следует, так как наличие таких пользователей может вызвать конфликты в системе. Например, Gnome 3 в Debian 10 просто отказался загружаться после того, как мы создали такого пользователя.

В поле комментарий можно написать все что угодно, определенного формата в современных системах нет, но исторически данное поле называется GECOS и подразумевает перечисление через запятую следующих опций:

```
Полное имя пользователя, адрес офиса или домашний адрес, рабочий телефон, домашний телефон
```

Следующие два поля указывают на домашнюю директорию пользователя и его командную оболочку. Разным пользователям можно назначить разные командные оболочки, скажем, ваш коллега предпочитает zsh, то вы можете без проблем назначить ему любимую оболочку.

Рассмотрение командных оболочек Linux выходит за рамки данной статьи, поэтому мы оставим эту тему, но расскажем о двух специализированных «оболочках», которые указывают, когда пользователю запрещен интерактивный вход в систему. Это обычно применяется для служб и пользователей от имени которых исполняются некоторые скрипты. Даже если такая учетная запись будет скомпрометирована войти в консоль с ней не удастся.

Обычно для этой цели используется:

/usr/sbin/nologin

или просто

/sbin/nologin

В современных системах директория /sbin является символической ссылкой на /usr/sbin. Реже используется «оболочка»:

/bin/false

Разница между ними заключается в том, что /bin/false просто запрещает вход в систему, а /sbin/nologin выдает сообщение:

```
This account is currently not available.
```

Как мы уже говорили выше, всех пользователей можно условно разделить на три группы. К специальным пользователям относятся root (суперпользователь) и nobody (с группами root и nogroup). Это практически полные противоположности, root имеет наименьшие UID и GID — 0 и является обладателем неограниченных прав, он имеет доступ к любому объекту системы и может выполнять любые настройки. В отличие от Администратора Windows, который максимально огражден от деструктивных действий, root может с легкостью уничтожить систему.

Мы думаем, что следующая команда известна каждому, а некоторым — печально известна:

rm -rf /

Предупреждение! Данная команда уничтожает корневую файловую систему, что приводит к ее полной неработоспособности и потере всех данных. Приведена сугубо в качестве примера.

Современные дистрибутивы по умолчанию блокируют выполнение явно деструктивных команд, но не запрещают их, все это выглядит как предупреждение: «так делать опасно, но если ты настаиваешь…»

С учетом вышесказанного есть ряд сложившихся правил работы с этим пользователем. Во-первых, не следует выполнять из-под учетной записи суперпользователя повседневную работу. Для разовых действий используйте sudo, для административных работ допустимо временно повышать привилегии с последующим обязательным выходом. Во-вторых, доступ к данной учетной записи должен быть тщательно ограничен, потому как утеря контроля над root равносильно полной потере контроля над системой.

Хорошая практика — сделать учетную запись root недоступной для интерактивного входа (как в Ubuntu) используя для повышения прав исключительно sudo, особенно это касается серверов с доступом к SSH из сети интернет.

Пользователь nobody (никто) — имеет наибольший идентификатор — 65534 и не может являться владельцем ни одного файла в системе, не состоит ни в одной привилегированной группе и не имеет никаких полномочий кроме стандартных. Используется для запуска от его имени процессов с низким уровнем доверия, чтобы ограничить их доступ к системе в случае возможной компрометации. Фактически к nobody будут всегда применяться права для «остальных» и при стандартных наборах привилегий 644 на файлы и 755 на директории он будет иметь возможность исключительно чтения и просмотра содержимого каталогов. Для чувствительных конфигурационных файлов, ключей и сертификатов используются более ограниченные наборы прав 640 или 600, к таким файлам nobody доступа не имеет.

Следующая условная группа — системные пользователи, которые используются для запуска служб и доступа к устройствам, например, www-data для веб-сервера или systemd-network для управления сетью через systemd. Первоначально для них выделялись идентификаторы от 1 до 100, но в связи с большим количеством служб в современных системах этот диапазон расширен до 499 в RHEL и производных от него, и до 999 в системах основанных на Debian.

Существует соглашение, что этот диапазон идентификаторов используется только системными службами и в нем не должно быть обычных

пользователей. Однако никто не мешает назначить службе UID выше 1000, а пользователю менее 999, но никаких последствий это иметь не будет и никак не скажется на привилегиях. Это разделение чисто условное и предназначено для повышения удобства администрирования. Встретив в незнакомой системе пользователя с UID до 999, вы будете с большой долей вероятности предполагать, что это служба.

Кроме того, для многих стандартных пользователей идентификаторы зарезервированы, так во всех Debian и основанных на нем дистрибутивах www-data имеет UID и GID — 33, а proxy — 13. Это удобно, скажем вместо:

chown -R www-data:www-data /var/www

быстрее набрать:

chown -R 33:33 /var/www

Но это тоже условность, скажем в отсутствии установленного веб-сервера мы можем назначить «зарезервированный» UID другому пользователю без каких-либо последствий, веб-сервер при установке возьмет ближайший свободный UID, но такие действия безусловно являются дурным тоном.

Многое стороннее ПО, например сервер 1С и сборки PostgresPro занимают ближайшие свободные идентификаторы с верхнего конца диапазона: 999, 998 и т. д.

Из всего изложенного выше вы должны понимать, что система определяет пользователей по идентификаторам, а не по именам, а присвоение идентификаторов регулируется определенными правилами, но никакие из них, кроме двух специальных (0 для root и 65534 для nobody), не дают каких-либо дополнительных прав или привилегий.

Для более гибкого управления пользователями предназначены группы, они позволяют объединять пользователей по любому произвольному принципу и назначать им дополнительные права. Если мы хотим разрешить пользователю повышение прав, то мы включаем его в группу sudo, при этом мы всегда можем легко узнать, кто именно обладает такой возможностью, достаточно посмотреть участников группы.

Посмотреть список групп можно в файле /etc/group

Он имеет следующий формат:

```
наименование группы:пароль:GID:пользователи группы
```

Большая часть полей аналогична записям в файле /etc/passwd и мы не будем подробно останавливаться на них. Разберем последний параметр — пользователи группы. При создании пользователя в Linux, если не указано

иное, ему автоматически создается основная группа с повторяющим имя названием. Такой пользователь в последнем поле не указывается. Это хорошо видно на скриншоте выше. Но мы можем создать и отдельную группу, которая первоначально не содержит пользователей, и она будет иметь точно такой же вид записи с пустым последним полем.

После того как мы добавим пользователей в группу они будут перечислены в последнем поле через запятую.

Давайте внимательно посмотрим на скриншот и проанализируем членство созданного при установке пользователя andrey в дополнительных группах. Практически все они связанны с доступом к оборудованию или дают возможность использовать некоторые системные механизмы, скажем группа plugdev предоставляет возможность монтировать съемные устройства.

В современных системах, при работе в графических средах доступ к оборудованию часто предоставляется механизмами рабочей среды, которые не используют разделение прав на основе членства пользователя в соответствующих группах. Поэтому, если вы используете стандартные механизмы дистрибутива, то для полноценного использования оборудования вам нет необходимости включать пользователя в дополнительные группы. Однако это может потребоваться при работе в консоли или при использовании нестандартных или специализированных устройств в графической среде.

Теперь о паролях, как мы уже говорили, современные системы не хранят пароли в открытом виде, а используют для этого хеши — односторонние криптографические функции, при вводе пароля система вычисляет его хеш и сравнивает с уже сохраненным, если хеши совпадают — то пароль введен верно. Такие пароли в Linux называются «затененными» и хранятся в отдельных файлах. Для пользователей это файл /etc/shadow.

Синтаксис этого файла предусматривает девять полей, содержащих следующую информацию:

Имя пользователя

Хеш пароля, если поле содержит ! или *, это означает, что учётная запись заблокирована и этот пользователь не сможет войти в систему.

Дата последней смены пароля — Количество дней, прошедших с 1 января 1970 г.

Число дней, которое должно пройти до смены пароля — Минимальный срок (в днях), который должен пройти, прежде чем пользователь сможет сменить пароль.

Число дней, после которого необходимо сменить пароль — Максимальный срок (в днях), по истечении которого необходимо сменить пароль.

Число дней до предупреждения о необходимости смены пароля — Число дней до истечения срока действия пароля, в течение которых пользователь получает предупреждение об окончании его срока действия.

Число дней до отключения учётной записи — Число дней после окончания срока действия пароля до отключения учётной записи.

Дата отключения учетной записи — Количество дней, прошедших с 1 января 1970 г.

Зарезервированное поле

Наибольший интерес представляет второе поле, содержащее хеш пароля, оно имеет структуру:

```
$ID$SALT$ENCRYPTED
```

Где ID — применяемый тип шифрования, могут использоваться следующие значения:

$1 — MD5, самый слабый хеш, в настоящее время не используется

$2 — Blowfish, использовался в BSD-системах

$5 — SHA-256

$6 — SHA-512

В современных системах используется наиболее стойкий хеш SHA-512.

Следующая часть строки — SALT — это соль (модификатор), генерируется случайно для каждого пользователя и используется для увеличения стойкости пароля. Хеш — это односторонняя криптографическая функция, которая позволяет быстро вычислить результат, но сделать обратное преобразование хеша невозможно. Единственный возможный вариант атаки — это подбор. Существуют уже вычисленные значения хеша для наиболее часто используемых сочетаний и словарных слов, т. н. радужные таблицы, с их помощью атака на хеш производится поиском по уже готовой базе.

Соль позволяет исключить атаки при помощи заранее вычисленных значений и при ее случайном значении позволяет скрыть наличие у пользователей одинаковых паролей. На скриншоте выше пользователям ivan и maria были установлены одинаковые пароли, но за счет различной соли они имеют различный хеш.

Ну и наконец ENCRYPTED — это, собственно, хеш. Кроме того, данное поле может содержать спецсимволы: *, ! и !!. Все они обозначают, что учетная запись заблокирована для входа с паролем, но не запрещают иные варианты

входа (по SSH-ключу). Символ * обычно используется для системных учетных записей, которым вход в систему запрещен, ! ставится для записей пользователей без пароля или заблокированных администратором (тогда ! ставится перед содержимым поля). Иногда используется символ !!, означающий что данной учетной записи никогда не присваивался пароль (также для такой записи может использоваться просто !).

Для групп используется аналогичный по назначению файл /etc/gshadow

Его структура более проста, а поля имеют следующее назначение:

Имя группы

Хеш пароля, если пароль установлен, не члены группы могут войти в нее, выполнив команду newgrp и указав пароль. Если это поле содержит * или !, ни один пользователь не сможет войти в неё указав пароль.

Администраторы группы — Перечисленные здесь (через запятую) члены группы могут добавлять или удалять членов группы с помощью команды gpasswd.

Члены группы — Здесь перечисляются (через запятую) обычные члены группы, не администраторы.

Поле пароля также может содержать спецсимволы и также как для пользователей символ * используется преимущественно для системных групп, а ! для групп пользователей. По умолчанию пароли групп отсутствуют и вход в них заблокирован, т. е. никто, кроме членов группы, не может войти в группу, а членам пароль не нужен.

Файлы shadow и gshadow содержат важную учетную информацию и должны быть хорошо защищены. Владельцами этих файлов является root:shadow и права доступа для них установлены как 640. Это означает, что вносить изменения в них может только владелец — root, а читать только члены специальной группы shadow. Это дополнительный уровень безопасности, позволяющий исключить доступ к этим файлам добавлением пользователя в группу root (что даст доступ на чтение к другим конфигурационным файлам).

Так как данные о пользователях являются критичными для системы, то указанные выше файлы крайне не рекомендуется изменять вручную, а для управления пользователями и группами следует использовать специальные утилиты, о которых мы поговорим в следующей части. При любом изменении информации в них штатными инструментами система создает резервную копию предыдущей версии файла с добавлением в конце имени символа ~:

passwd

passwd~

Поэтому даже если что-то пойдет не так, всегда есть возможность (при условии физического доступа к системе) восстановить состояние этих файлов на момент перед внесением изменений в конфигурацию. В качестве примера приведем следующую ситуацию: вы случайно удалили единственного пользователя с административными правами из группы sudo, root при этом заблокирован, после завершения сеанса в системе не останется ни одного привилегированного пользователя.

Создание пользователей

Для создания пользователей используется команда useradd, но не будем спешить с ее применением, сначала ознакомимся с используемыми по умолчанию параметрами, для этого выполните:

useradd –D

Здесь мы можем увидеть место размещения домашних каталогов, оболочку по умолчанию и каталог с шаблоном домашней директории /etc/skel. Эта директория будет полностью скопирована в директорию нового пользователя и содержит базовые настройки окружения пользователя. Вы можете воспользоваться этим, если вам нужно формировать определенную структуру домашних каталогов или применять какие-либо специфичные настройки. При этом не рекомендуется вносить изменения в оригинальный /etc/skel, рекомендуется сделать его копию (можно и не одну), в которую внести все необходимые изменения.

Также обратите внимание на параметр:

GROUP=100

Но как нам известно в Debian пользовательские группы начинаются с 1000, а в RHEL с 500, поэтому в современных системах этот параметр игнорируется. Все параметры, кроме SKEL, могут быть изменены, но практический смысл это имеет только для HOME и SHELL.

Скажем вы настраиваете многопользовательский веб-сервер и хотите создавать домашние директории в каталоге веб-сервера, в этом случае можно выполнить:

useradd -D -b /var/www/home

А для изменения командной оболочки:

useradd -D -s /bin/bash

Чтобы добавить нового пользователя введите:

useradd -m ivan

Ключ -m предписывает создать домашний каталог пользователя. Это самый простой вариант использования, но при использовании

дополнительных ключей мы можем сразу задать или переопределить многие параметры пользователя. Получить их список можно командой:

useradd -h

Приведем некоторые из них:

-b — задает домашний каталог пользователя

-c — комментарий к учетной записи

-g — задает основную группу пользователя

-G — задает дополнительные группы

-m — создать домашний каталог пользователя

-N — не создавать группу с именем пользователя

-k — путь к каталогу шаблона домашней директории (по умолчанию /etc/skel)

-s — командная оболочка

Например:

useradd -m -b /var/www/ivanov -g webuser -G www-data -k /etc/myskel -s /sbin/nologin ivanov

Этой командой мы создадим пользователя ivanov, которому назначим домашнюю директорию в /var/www/ivanov, для которой будет использован шаблон из /etc/myskel, включим его в основную группу webuser и дополнительную www-data. Также запретим ему интерактивный вход в систему, назначив оболочкой /sbin/nologin.

Созданная учетная запись будет заблокирована до тех пор, пока мы не установим для нее пароль, это можно сделать следующей командой:

passwd ivan

которая установит пароль к учетной записи ivan. Для блокировки пароля используйте:

passwd -l ivan

Но учтите, что данная блокировка распространяется только на вход по паролю, если у пользователя настроены иные способы входа, скажем по SSH-ключу, то доступ в систему с их помощью сохранится. Поэтому данный способ следует использовать в тех случаях, когда есть подозрение на компрометацию пароля, но в полной блокировке учетной записи нет необходимости. Для разблокировки потребуется заново установить пароль.

Изменение пользователей

Для того, чтобы изменить параметры уже существующей учетной записи используется команда usermod, для ознакомления с ее возможностями получим список ключей командой

usermod -h

Приведем некоторые из них:

-c — изменить комментарий

-d — новый домашний каталог

-g — новая основная группа

-G — новые дополнительные группы

-a — добавить пользователя в дополнительные группы, не удаляя из других групп, используется совместно с ключом -G

-l — новое имя учетной записи

-L — блокировать учетную запись

-m — переместить содержимое домашнего каталога, используется только вместе с —d

-U — разблокировать учетную запись

Например:

usermod -c "Ivanov Ivan" -aG sudo ivan

Данная команда создаст новый комментарий к учетной записи и добавит пользователя ivan в дополнительную группу sudo.

Удаление пользователей

Для удаления учетной записи пользователя используется команда userdel, например:

userdel vasya

которая удалит учетную запись vasya.

Команда имеет дополнительные ключи, которые также можно просмотреть, запустив ее с ключом -h, практическую пользу представляют два из них:

-r — удалить домашний каталог пользователя

-f — удалять файлы, даже если они не принадлежат пользователю

На практике оба ключа обычно используются совместно, но будьте осмотрительны и перед удалением убедитесь, что удаляемые данные не нужны другим пользователям. Чтобы удалить пользователя вместе со всеми данными выполните:

userdel -rf vasya

При этом будет удален не только рабочий каталог, но и сообщения электронной почты пользователя, если таковые были в системе.

Управление группами пользователей

Для управления группами используется аналогичный набор команд: groupadd, groupmod и groupdel. Их ключи также можно посмотреть, запустив их с параметром -h.

Для создания группы используйте:

groupadd office

Из дополнительных ключей имеют практический смысл:

-g — использовать указанный GUID

-f — завершить команду без ошибки если группа с таким именем уже существует и отменить действие -g, если указанный GUID уже используется.

При помощи команды usermod мы можем изменить GUID и наименование группы, скажем:

groupmod -n sales office

Данная команда переименует группу office в группу sales. Используемые ключи:

-g — изменить GUID группы

-n — изменить наименование группы

Ну и наконец groupdel, который используется для удаления групп:

groupdel office

Из параметров можно использовать:

-f — удалить группу, даже если она является основной для пользователя

При удалении групп всегда нужно проявлять осмотрительность и не следует удалять группы, в которых состоят пользователи. Если же такая необходимость существует, то следует предварительно вывести из группы пользователей и изменить группу для всех объектов, которыми владела удаляемая группа.

Управление членством пользователей в группах

Управлять членством пользователей в группах можно различным образом. Выше мы уже приводили примеры с использованием usermod, для изменения основной группы пользователя используйте:

usermod -g office ivan

Данная команда изменит основную группу пользователя ivan на office. Для добавления в дополнительные группы следует выполнить команду:

usermod -aG office, sales ivan

Которая добавит пользователя ivan в группы office и sales.

Также существует и другая команда gpasswd, которая позволяет не только добавлять, но и удалять членство пользователей в группах. Для добавления пользователя в группу используйте:

gpasswd -a ivan office

Данная команда добавить пользователя ivan в группу office. Для удаления членства выполните:

gpasswd -d ivan office

Для того, чтобы проверить членство пользователя в группах используйте команду:

groups ivan

где ivan — имя пользователя.

Чтобы быстро удалить пользователя из всех дополнительных групп используйте:

usermod -G "" ivan

Как видим, управление пользователями и группами в Linux не такое уж и сложное дело, особенно когда вы понимаете, что значит тот или иной параметр и для чего вы это делаете.

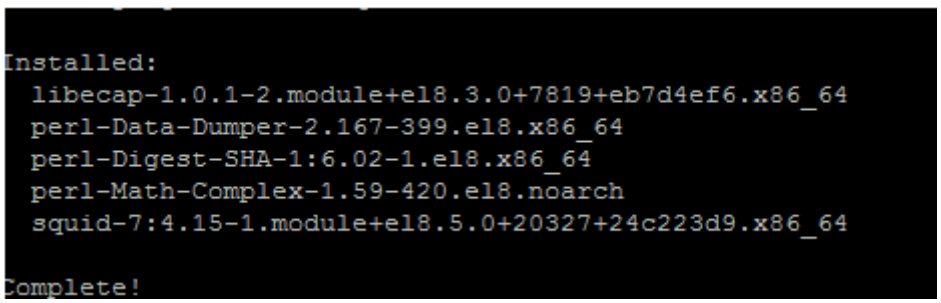
16. Кеширующий прокси-сервер (squid)

Squid это, наверное, самый известный и популярный open source кэширующий прокси сервер. Прокси сервер Squid поддерживает большое количество кэширующих протоколов (ICP, HTCP, CARP, WCCP), позволяет проксировать трафик протоколов HTTP, HTTPS, FTP и т.д. Squid легко масштабируется, его можно использовать для фильтрации контента и управления скоростью доступа к сайтам для разных категорий пользователей, он поддерживает аутентификацию пользователей (в том числе пользователей Active Directory через LDAP). В этой статье мы рассмотрим типовую установку прокси сервера Squid на Linux.

Установка прокси сервера Squid в Linux

Установите пакет Squid на хост с помощью вашего пакетного менеджера. В зависимости от версии Linux нужно выполните одну из следующих команд:

- Oracle, Rocky, CentOS или Red Hat Enterprise Linux: dnf install -y squid
- SUSE Linux Enterprise Server: zypper install squid
- Ubuntu или Debian: apt-get install squid



Если вы хотите использовать аутентификацию пользователей в Squid, нужно дополнительно установить пакет:

- В RHEL/CentOS/Fedora: dnf -y install httpd-tools

- В Debian/Ubuntu: sudo apt install apache2-utils

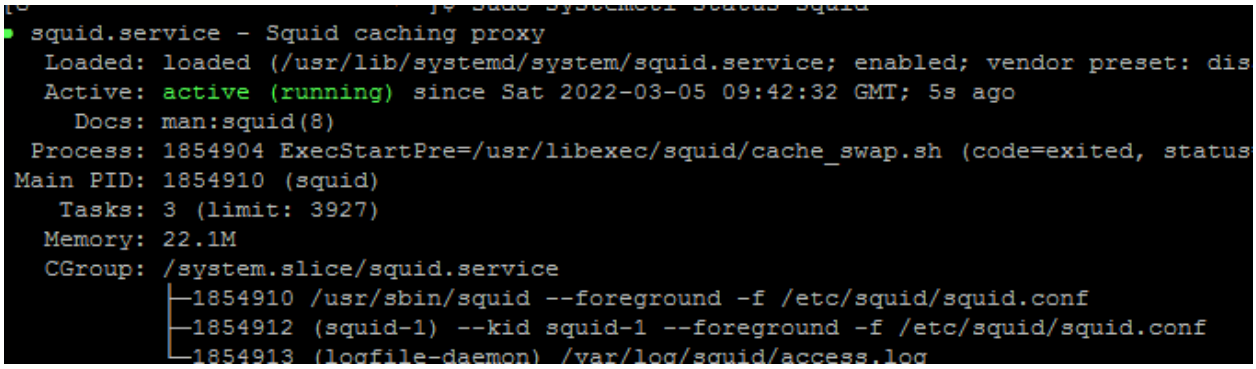
Добавьте сервис Squid в автозагрузку Linux через systemd, запустите службу и проверьте статус:

systemctl enable squid

systemctl start squid

systemctl status squid

systemctl status squid – running



Чтобы ввести все опции, с которыми собран squid:

squid –v

```
Squid Cache: Version 4.15
Service Name: squid
This binary uses OpenSSL 1.1.1k FIPS 25 Mar 2021. For legal restrictions on distribution see https://www.ope
nssl.org/source/license.html
configure options: '--build=x86_64-redhat-linux-gnu' '--host=x86_64-redhat-linux-gnu' '--program-prefix=' '-
-prefix=/usr' '--exec-prefix=/usr' '--bindir=/usr/bin' '--sbindir=/usr/sbin' '--sysconfdir=/etc' '--datadir
=/usr/share' '--includedir=/usr/include' '--libdir=/usr/lib64' '--libexecdir=/usr/libexec' '--localstatedir
=/var' '--sharedstatedir=/var/lib' '--mandir=/usr/share/man' '--infodir=/usr/share/info' '--libexecdir=/usr/
lib64/squid' '--datadir=/usr/share/squid' '--sysconfdir=/etc/squid' '--with-logdir=/var/log/squid' '--with-p
idfile=/var/run/squid.pid' '--disable-dependency-tracking' '--enable-eui' '--enable-follow-x-forwarded-for'
'--enable-auth' '--enable-auth-basic=DB,fake,getpwnam,LDAP,NCSA,PAM,POP3,RADIUS,SASL,SMB,SMB_LM' '--enable-a
uth-ntlm=SMB_LM,fake' '--enable-auth-digest=file,LDAP' '--enable-auth-negotiate=kerberos' '--enable-external
-acl-helpers=LDAP_group,time_quota,session,unix_group,wbinfo_group,kerberos_ldap_group' '--enable-storeid-re
write-helpers=file' '--enable-cache-digests' '--enable-cachemgr-hostname=localhost' '--enable-delay-pools'
'--enable-epoll' '--enable-icap-client' '--enable-ident-lookups' '--enable-linux-netfilter' '--enable-remova
l-policies=heap,lru' '--enable-snmp' '--enable-ssl' '--enable-ssl-crtd' '--enable-storeio=aufs,diskd,ufs,roc
k' '--enable-diskio' '--enable-wccpv2' '--enable-esi' '--enable-ecap' '--with-aio' '--with-default-user=squi
d' '--with-dl' '--with-openssl' '--with-pthreads' '--disable-arch-native' '--disable-security-cert-validator
s' '--with-swapdir=/var/spool/squid' 'build_alias=x86_64-redhat-linux-gnu' 'host_alias=x86_64-redhat-linux-g
nu' 'CFLAGS=-O2 -g -pipe -Wall -Werror=format-security -Wp,-D_FORTIFY_SOURCE=2 -Wp,-D_GLIBCXX_ASSERTIONS -fe
xceptions -fstack-protector-strong -grecord-gcc-switches -specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 -spe
cs=/usr/lib/rpm/redhat/redhat-annobin-cc1 -m64 -mtune=generic -fasynchronous-unwind-tables -fstack-clash-pro
tection -fcf-protection' 'LDFLAGS=-Wl,-z,relro -Wl,-z,now -specs=/usr/lib/rpm/redhat/redhat-hardened-ld' 'CX
XFLAGS=-O2 -g -pipe -Wall -Werror=format-security -Wp,-D_FORTIFY_SOURCE=2 -Wp,-D_GLIBCXX_ASSERTIONS -fexcept
ions -fstack-protector-strong -grecord-gcc-switches -specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 -specs=/u
sr/lib/rpm/redhat/redhat-annobin-cc1 -m64 -mtune=generic -fasynchronous-unwind-tables -fstack-clash-protecti
on -fcf-protection' 'PKG_CONFIG_PATH=:/usr/lib64/pkgconfig:/usr/share/pkgconfig'
```

Настройка кэширующего прокси сервера Squid

По умолчанию squid использует конфигурационный файл /etc/squid/squid.conf.

Прежде чем редактировать этот файл, создайте его копию и защитите ее от записи:

```
sudo cp /etc/squid/squid.conf /etc/squid/squid.conf.original
sudo chmod a-w /etc/squid/squid.conf.original
```

В дальнейшем мы сможете вернуться к оригинальному конфигурационному файлу или использовать его в качестве справочника директив.

Чтобы отредактировать конфигурационный файл squid, выполните:

sudo mcedit /etc/squid/squid.conf

В начале конфигурационного файла Squid задается список контроля доступа, определяющий IP адреса (подсети) клиентов, с которых разрешено подключаться к прокси-серверу.

Например, вы хотите разрешить доступ только из вашей локальной сети. Добавьте директиву:

acl localnet src 192.168.10.0/24

Остальные строки acl localnet можно закомментировать.

Чуть ниже в конфигурации указан директива, разрешающая доступ для этих сетей:

http_access allow localnet

По умолчанию Squid принимает подключения пользователей на порту 3128. Вы можете изменить этот порт в директиве http_port. Мы повесим Squid на порт 4555:

http_port 4555

Если вы хотите использовать прокси сервер для кэширования, нужно настроить директорию для кэша:

cache_dir ufs /var/spool/squid 5120 32 256

5120 – размер кэша в МБ

32 – количество каталогов первого уровня для кэша, 256 – каталоги второго уровня

Чтобы создать на диске структуру папок под кэш squid, выполните команду:

squid -z

Для аутентификации пользователей на squid можно использовать простейшую basic аутентификацию. Сначала создайте файл для хранения пользователей и паролей:

sudo touch /etc/squid/passwd && sudo chown squid /etc/squid/passwd

Чтобы добавить пользователя в этот файл, выполните команду:

sudo htpasswd -c /etc/squid/passwd username1

При добавлении следующих пользователей указывать параметр -c не нужно.

```
[(                          squid]$ sudo htpasswd /etc/squid/passwd prxuser1
New password:
Re-type new password:
Adding password for user prxuser1
```

добавление пользователя в squid

AeroAdmin – доступная программа для удаленного доступа к компьютеру

Затем добавьте следующие строки в начало конфигурационного файла squid.conf:

```
#использовать basic auth и указать путь к файлу с паролями
auth_param basic program /usr/lib64/squid/basic_ncsa_auth /etc/squid/passwd
# количество одновременных подключений
auth_param basic children 5
auth_param basic realm Squid Basic Authentication
# длительность сессии без повторного ввода логина и пароля
auth_param basic credentialsttl 8 hours
auth_param basic casesensitive off
acl auth_users proxy_auth REQUIRED
http_access allow auth_users
```

В разных дистрибутивах Linux путь к файлу basic_ncsa_auth может отличаться:

/usr/lib64/squid/basic_ncsa_auth (Oracle Linux, RHEL, CentOS)

/usr/lib/squid/basic_ncsa_auth (Debian, Ubuntu, Kali Linux)

squid.conf конфигурационный файл squid

```
#
# Recommended minimum configuration:
#

auth_param basic program /usr/lib64/squid/basic_ncsa_auth /etc/squid/passwd
auth_param basic realm proxy
acl authenticated proxy_auth REQUIRED
```

Можно реализовать аутентификацию в AD с помощью basic auth:

```
auth_param basic program /usr/lib64/squid/basic_ldap_auth -R -b "dc=winitpro,dc=ru" -D "squid_srv@win
itpro.ru " -w "<user password>" -f "sAMAccountName=%s" <AD domain controller IP address>
```

Недостаток Basic аутентификации в тои, что пароль передается в виде текста, зашифрованного с помощью Base64 (легко расшифровывается, см. пример в статье об отправке писем через telnet с аутентификацией). Поэтому для интеграции с Active Directory лучше использовать Kerberos аутентификацию в Squid. Если AD нет, можно настроить HTTP Digest аутентификацию в Squid. Но это уже за рамками этой статьи.

В конце конфигурационного файла с помощью директив http_access задаются разрешающие и запрещающие правила. Например, чтобы разрешить доступ только аутентифицированным пользователям:

http_access allow localhost

http_access allow authenticated

http_access deny all

В Squid можно заблокировать определенный сайты. Создайте файл со списком запрещенных сайтов:

sudo vi /etc/squid/blocked_sites

Добавьте в файл список сайтов, доступ к которым нужно запретить:

facebook.com

twitter.com

instagram.com

И добавьте такие строки в конфигурационный файл squid:

acl blocked_sites dstdomain "/etc/squid/blocked_sites"

http_access deny blocked_sites

После внесения изменений в конфигурационный файл squid нужно проверить его на синтаксические ошибки:

sudo squid -k parse

Если ошибок нет, перечитайте конфигурацию squid для применения новых настроек (не нужно перезапускать демона):

sudo squid -k reconfigure

Разрешите в фаейрволе подключение к порту прокси сервера (в нашем примере это TCP 4555). Если у вас используется firewalld, добавьте разрешающее правило так:

firewall-cmd --zone=public --add-port=4555/tcp

firewall-cmd --reload

Если ваш Linux сервер расположен у внешнего провайдера (AWS, Azure, Oracle Cloud, и т. д.), не забудьте добавить разрешающее правило для порта 4555 в соответствующую Security Group.

Проверьте, что Squid слушает порт, указанный в директиве http_port:

netstat -tulnp

С помощью curl можно проверить доступность прокси сервера squid:

curl -x http://xx.xx.xx.xx:4555 -L http://google.com

Если появляется ошибка authentication required/ Access Denied , значит нужно передать имя пользователя и пароль для подключения к squid:

curl -x http://xx.xx.xx.xx:4555 --proxy-user proxyuser1:TGFTM0xDVjNm -I http://google.com



Для анализа логов squid в реальном времени используйте команду:

sudo tail -f /var/log/squid/access.log

Если браузер выводит ошибку The proxy server is refusing connections, а в логе access.log есть строка TCP_DENIED/403 4041 CONNECT, проверьте ваши ACL.

При успешной аутентификации пользователя в логе squid появится сообщение TCP_TUNNEL/200 39 CONNECT winitpro.ru:443 proxyusername HIER_DIRECT/xx.xx.xx.xx или TCP_MISS/200 .

Не забудьте задать настройки прокси сервера в браузерах на компьютерах пользователей. Задайте IP адрес (имя) хоста squid и порт в настройках браузера. На компьютеры Windows можно распространить настройки прокси через GPO.

17. Производительность системы

В этой главе мы подробно обсудим производительность системы в Unix.

Мы познакомим вас с несколькими бесплатными инструментами, которые доступны для мониторинга и управления производительностью в системах Unix. Эти инструменты также содержат рекомендации по диагностике и устранению проблем с производительностью в среде Unix.

Unix имеет следующие основные типы ресурсов, которые необходимо контролировать и настраивать−

- **Процессор**
- **Память**
- **Дисковое пространство**
- **Линии связи**
- **Время ввода-вывода**
- **Сетевое время**
- **Прикладные программы**

Компоненты производительности

В следующей таблице перечислены пять основных компонентов, которые занимают системное время −

Sr.No .	Компонент и описание
1	**Состояние пользователя CPU** Фактическое количество времени, которое процессор тратит на запуск программы пользователя в пользовательском состоянии. Он включает время, затраченное на выполнение библиотечных вызовов, но не включает время, затраченное на ядро от его имени
2	**Состояние системы CPU** Это количество времени, которое процессор проводит в состоянии системы от имени этой программы. Все **процедуры ввода**-вывода требуют служб ядра. Программист может повлиять на это значение, блокируя передачу ввода-вывода
3	**Время ввода-вывода и сетевое время** Это время, затрачиваемое на перемещение данных и обслуживание запросов ввода-вывода
4	**Производительность виртуальной памяти** Это включает переключение контекста и замену
5	**Прикладная программа** Время, затраченное на запуск других программ - когда система не обслуживает это приложение, потому что другое приложение в настоящее время имеет процессор

Инструменты производительности

Unix предоставляет следующие важные инструменты для измерения и точной настройки производительности системы Unix−

Sr.No .	Команда и описание
1	**nice/renice** Запускает программу с измененным приоритетом планирования
2	**netstat** Печатает сетевые подключения, таблицы маршрутизации, статистику интерфейса, маскарадные соединения и членство в многоадресной рассылке
3	**время** Помогает время простой команды или дать использование ресурсов
4	**время безотказной работы**

	Это средняя загрузка системы
5	**ps** Сообщает моментальный снимок текущих процессов
6	**vmstat** Отчеты о статистике виртуальной памяти
7	**gprof** Отображение данных профиля графика вызовов
8	**проф** Облегчает профилирование процесса
9	**Наверх** Отображает системные задачи

Вы можете использовать справку Manpage, чтобы проверить полный синтаксис для каждой команды, упомянутой здесь.



МОДУЛЬ 3. Системы и сети передачи данных

1. Модель OSI/ISO. Обзор сетевых протоколов
2. L2-сеть
3. L3-сеть
4. L4-сеть
5. Сеть передачи данных (SAN)
6. NAT
7. Виртуальные частные сети (VPN): OpenVPN, L2TP/Ipsec, SSTP, PPTP
8. Firewall
9. Высокоуровневые сетевые протоколы
10. Траблшутинг
11. DHCP, PXE
12. DNS
13. HTTP/HTTPS
14. IPv4, IPv6
15. Базовое программирование на Bash. Коды возврата, функции.
16. Regexp и их использование для синтаксического анализа. Полезные утилиты.
17. Разбор скриптов и написание своих скриптов. Linter. Shell check.

1. Модель OSI/ISO. Обзор сетевых протоколов

Эталонная модель OSI

При огромном числе компаний, разрабатывающих и производящих оборудование и программы для организации сетей вопрос стандартизации играет важнейшую роль. Возможность работы разнородного оборудования в одной сети обеспечивает стандарт Ethernet, разработанный IEEE. Использование этого стандарта всеми производителями адаптеров и концентраторов позволяет им выпускать оборудование, способное взаимодействовать с программами и аппаратурой других фирм при работе в одной сети.

Для обеспечения совместимости программ Международная Организация по Стандартизации (ISO - International Standards Organization) разработала модель сетевой архитектуры, получившую известность как OSI-модель. Модель Взаимодействия Открытых Систем (Open Systems Interconnect) описывает структуру сетевых уровней. Не все разработчики программ в точности следуют этой модели, однако она дает основы понимания способов взаимодействия сетевых компонент.

Модель OSI

В начале 80-х годов ISO опубликовала модель, предназначенную для оказания помощи разработчикам при объединении различных сетей. Получившая широкую известность модель OSI содержит семь дискретных уровней, каждый из которых обеспечивает выполнение определенной части сетевых функций при обмене данными между компьютерами сети.

Преимущества использования модели OSI:

- Деление сложных сетевых операций на легко управляемые уровни.
- Изменения на одном уровне не действуют на другие уровни модели, что позволяет разработчикам приложений специализироваться на определенных и ограниченных задачах.
- Определение стандартных интерфейсов для простой интеграции оборудования различных компаний.

Эталонная модель OSI – это описательная схема сети; ее стандарты гарантируют высокую совместимость и способность к взаимодействию различных типов сетевых технологий. Кроме того, она иллюстрирует процесс перемещения информации по сетям. Это концептуальная структура, определяющая сетевые функции, реализуемые на каждом ее уровне. Модель OSI описывает, каким образом информация проделывает путь через сетевую

среду (например, провода) от одной прикладной программы (например, программы обработки таблиц) к другой прикладной программе, находящейся в другом подключенном к сети компьютере.

Эталонная модель OSI делит задачу перемещения информации между компьютерами через сетевую среду на семь менее крупных и, следовательно, более легко разрешимых подзадач.

Сетевой протокол — это набор правил, позволяющий осуществлять соединение и обмен данными между двумя и более включёнными в сеть компьютерами. Фактически разные протоколы зачастую описывают лишь разные стороны одного типа связи; взятые вместе, они образуют так называемый стек протоколов. Названия "протокол" и "стек протоколов" также указывают и на программное обеспечение, которым реализуется протокол.

Уровни протоколов

В соответствии с моделью OSI протоколы делятся на 7 уровней по своему назначению - от физического (формирование и распознавание электрических или других сигналов) до прикладного (API для передачи информации приложениями)

Поскольку нижние уровни (с 1 по 3) модели OSI управляют физической доставкой сообщений посети, их часто называют уровнями среды передачи данных (media layers). Верхние уровни (с 4 по 7) модели OSI обеспечивают точную доставку данных между компьютерами в сети, поэтому их часто называют уровнями хост-машины (host layers).

Прикладной уровень, Application layer - Верхний (7-й) уровень модели, обеспечивает взаимодействие сети и пользователя. Уровень разрешает приложениям пользователя доступ к сетевым службам, таким как обработчик запросов к базам данных, доступ к файлам, пересылке электронной почты. Также отвечает за передачу служебной информации, предоставляет приложениям информацию об ошибках и формирует запросы к уровню представления. Пример: HTTP, POP3, SMTP.

Уровень представления, Presentation layer - 6-й уровень отвечает за преобразование протоколов и кодирование/декодирование данных. Запросы приложений, полученные с уровня приложений, он преобразует в формат для передачи по сети, а полученные из сети данные преобразует в формат, понятный приложениям. На уровне представления может осуществляться сжатие/распаковка или кодирование/декодирование данных, а также перенаправление запросов другому сетевому ресурсу, если они не могут быть обработаны локально.

Сеансовый уровень, Session layer - 5-й уровень модели отвечает за поддержание сеанса связи, что позволяет приложениям взаимодействовать между собой длительное время. Сеансовый уровень управляет созданием/завершением сеанса, обменом информацией, синхронизацией задач, определением права на передачу данных и поддержанием сеанса в периоды неактивности приложений. Синхронизация передачи обеспечивается помещением в поток данных контрольных точек, начиная с которых возобновляется процесс при нарушении взаимодействия.

Транспортный уровень, Transport layer - 4-й уровень модели, предназначен для доставки данных без ошибок, потерь и дублирования в той последовательности, как они были переданы. При этом неважно, какие данные передаются, откуда и куда, то есть он предоставляет сам механизм передачи. Блоки данных он разделяет на фрагменты, размер которых зависит от протокола, короткие объединяет в один, а длинные разбивает. Протоколы этого уровня предназначены для взаимодействия типа точка-точка. Пример: TCP, UDP.

Сетевой уровень, Network layer - 3-й уровень сетевой модели OSI, предназначен для определения пути передачи данных. Отвечает за трансляцию логических адресов и имён в физические, определение кратчайших маршрутов, коммутацию и маршрутизацию, отслеживание неполадок и заторов в сети. На этом уровне работает такое сетевое устройство, как маршрутизатор.

Канальный уровень, Data Link layer - этот уровень предназначен для обеспечения взаимодействия сетей на физическом уровне и контроля за ошибками, которые могут возникнуть. Данные, полученные с физического уровня, он упаковывает во фреймы, проверяет на целостность, если нужно исправляет ошибки и отправляет на сетевой уровень. Канальный уровень может взаимодействовать с одним или несколькими физическими уровнями, контролируя и управляя этим взаимодействием. Спецификация IEEE 802 разделяет этот уровень на 2 подуровня - MAC (Media Access Control) регулирует доступ к разделяемой физической среде, LLC (Logical Link Control) обеспечивает обслуживание сетевого уровня. На этом уровне работают коммутаторы, мосты. В программировании этот уровень представляет драйвер сетевой платы, в операционных системах имеется программный интерфейс взаимодействия канального и сетевого уровней между собой, это не новый уровень, а просто реализация модели для конкретной ОС. Примеры таких интерфейсов: ODI, NDIS.

Физический уровень, Physical layer - самый нижний уровень модели, предназначен непосредственно для передачи потока данных. Осуществляет передачу электрических или оптических сигналов в кабель или в радиоэфир и соответственно их приём и преобразование в биты данных в соответствии с методами кодирования цифровых сигналов. Другими словами, осуществляет интерфейс между сетевым носителем и сетевым устройством. На этом уровне работают концентраторы (хабы), повторители (ретрансляторы) сигнала и медиаконверторы. Функции физического уровня реализуются на всех устройствах, подключенных к сети. Со стороны компьютера функции физического уровня выполняются сетевым адаптером или последовательным портом.

В основном используются протокол TCP/IP

Определение:

Transmission Control Protocol/Internet Protocol, TCP/IP (Протокол управления передачей/Протокол Интернета)

Большинство операционных систем сетевых серверов и рабочих станций поддерживает TCP/IP, в том числе серверы NetWare, все системы Windows, UNIX, последние версии Mac OS, системы OpenMVS и z/OS компании IBM, а также OpenVMS компании DEC. Кроме того, производители сетевого оборудования создают собственное системное программное обеспечение для TCP/IP, включая средства повышения производительности устройств. Стек TCP/IP изначально применялся на UNIX-системах, а затем быстро распространился на многие другие типы сетей.

Протоколы локальных сетей

- IPX/SPX;
- NetBEUI;
- AppleTalk;
- TCP/IP;
- SNA;
- DLC;
- DNA;

Свойства протоколов локальной сети

В основном протоколы локальных сетей имеют такие же свойства, как и Другие коммуникационные протоколы, однако некоторые из них были разработаны давно, при создании первых сетей, которые работали медленно, были ненадежными и более подверженными электромагнитным и радиопомехам. Поэтому для современных коммуникаций некоторые протоколы не вполне пригодны. К недостаткам таких протоколов относится

слабая защита от ошибок или избыточный сетевой трафик. Кроме того, определенные протоколы были созданы для небольших локальных сетей и задолго до появления современных корпоративных сетей с развитыми средствами маршрутизации.

Протоколы локальных сетей должны иметь следующие основные характеристики:

- обеспечивать надежность сетевых каналов;
- обладать высоким быстродействием;
- обрабатывать исходные и целевые адреса узлов;
- соответствовать сетевым стандартам, в особенности - стандарту IEEE 802.

В основном все протоколы, рассматриваемые в этой главе, соответствуют перечисленным требованиям, однако, как вы узнаете позднее, у одних протоколов возможностей больше, чем у других.

В таблице перечислены протоколы локальных сетей и операционные системы, с которыми эти протоколы могут работать. Далее в главе указаны протоколы и системы (в частности, операционные системы серверов и хост компьютеров) будут описаны подробнее.

Таблица Протоколы локальных сетей и сетевые операционные системы

ПРОТОКОЛ	СООТВЕТСТВУЮЩАЯ ОПЕРАЦИОННАЯ СИСТЕМА
IPX/SPX	Novell NetWare
NETBEUI	Первые версии операционных систем Microsoft Windows
APPLETALK	Apple Macintosh
TCP/IP	UNIX, Novel NetWare, современные версии операционных систем Microsoft Windows, операционные системы мэйнфреймов IBM
SNA	Операционные системы мэйнфреймов и миникомпьютеров IBM
DLC	Клиентские системы, взаимодействующие с мэйнфреймами IBM, настроенными на работу с протоколом SNA

Понятие протокола Интернет

Очевидно, что рано или поздно компьютеры, расположенные в разных точках земного шара, по мере увеличения своего количества должны были обрести некие средства общения. Такими средствами стали компьютерные сети. Сети бывают локальными и глобальными. Локальная сеть — это сеть,

объединяющая компьютеры, географически расположенные на небольшом расстоянии друг от друга - например, в одном здании. Глобальные сети служат для соединения сетей и компьютеров, которых разделяют большие расстояния - в сотни и тысячи километров. Интернет относится к классу глобальных сетей.

Простое подключение одного компьютера к другому - шаг, необходимый для создания сети, но не достаточный. Чтобы начать передавать информацию, нужно убедиться, что компьютеры "понимают" друг друга. Как же компьютеры "общаются" по сети? Чтобы обеспечить эту возможность, были разработаны специальные средства, получившие название "протоколы". Протокол — это совокупность правил, в соответствии с которыми происходит передача информации через сеть. Понятие протокола применимо не только к компьютерной индустрии. Даже те, кто никогда не имел дела с Интернетом, скорее всего работали в повседневной жизни с какими-либо устройствами, функционирование которых основано на использовании протоколов. Так, обычная телефонная сеть общего пользования тоже имеет свой протокол, который позволяет аппаратам, например, устанавливать факт снятия трубки на другом конце линии или распознавать сигнал о разъединении и даже номер звонящего.

Исходя из этой естественной необходимости, миру компьютеров потребовался единый язык (то есть протокол), который был бы понятен каждому из них.

Краткое описание протоколов Интернет

TCP/IP

Над созданием протоколов, необходимых для существования глобальной сети, трудились лучшие умы человечества. Одним из них был Винтон Серф (Vinton G. Cerf). Сейчас этого человека называют "отцом Интернета". В 1997 году Президент США Билл Клинтон наградил Винтона Серфа и его коллегу Роберта Кана (Robert E. Kahn) Национальной медалью за заслуги в области технологии, отметив их вклад в становление и развитие Интернета. Ныне Винтон Серф занимает пост старшего вице-президента по Интернет-архитектуре в корпорации MCI WorldCom Inc.

В 1972 году группа разработчиков под руководством Винтона Серфа разработала протокол TCP/IP - Transmission Control Protocol/Internet Protocol (Протокол управления передачей/Протокол Интернета).

Эксперимент по разработке этого протокола проводился по заказу Министерства обороны США. Данный проект получил название ARPANet

(Advanced Research Projects Agency Network - Сеть агентства важных исследовательских проектов). Очевидно, что в обстановке войны, когда необходимость в обмене информацией встает как никогда остро, возникает проблема непредсказуемости состояния пути, по которому будет передана та или иная информация - любой из узлов передачи в любой момент может быть выведен из строя противником. Поэтому главной задачей при разработке сетевого протокола являлась его "неприхотливость" - он должен был работать с любым сетевым окружением и, кроме того, обладать гибкостью в выборе маршрута при доставке информации.

Позже TCP/IP перерос свое изначальное предназначение и стал основой стремительно развивавшейся глобальной сети, ныне известной как Интернет, а также небольших сетей, использующих технологии Интернета - интранет. Стандарты TCP/IP являются открытыми и непрерывно совершенствуются.

На самом деле TCP/IP является не одним протоколом, а целым набором протоколов, работающих совместно. Он состоит из двух уровней. Протокол верхнего уровня, TCP, отвечает за правильность преобразования сообщений в пакеты информации, из которых на приемной стороне собирается исходное послание. Протокол нижнего уровня, IP, отвечает за правильность доставки сообщений по указанному адресу. Иногда пакеты одного сообщения могут доставляться разными путями.

Схема функционирования протокола TCP/IP:

HTTP

Протокол HTTP (Hypertext Transfer Protocol - Протокол передачи гипертекста) является протоколом более высокого уровня по отношению к протоколу TCP/IP - протоколом уровня приложения. HTTP был разработан для эффективной передачи по Интернету Web-страниц. Именно благодаря HTTP мы имеем возможность созерцать страницы Сети во всем великолепии. Протокол HTTP является основой системы World Wide Web.

Вы отдаете команды HTTP, используя интерфейс браузера, который является HTTP-клиентом. При щелчке мышью на ссылке браузер запрашивает у Web-сервера данные того ресурса, на который указывает ссылка - например, очередной Web-страницы.

Чтобы текст, составляющий содержимое Web-страниц, отображался на них определенным образом - в соответствии с замыслом создателя страницы - он размечается с помощью особых текстовых меток - тегов языка разметки гипертекста (HyperText Markup Language, HTML).

Адреса ресурсов Интернета, к которым вы обращаетесь по протоколу HTTP, выглядит примерно следующим образом: http://www.tut.by

FTP

Протокол FTP (File Transfer Protocol - Протокол передачи файлов) специально разработан для передачи файлов по Интернету. Позже мы поговорим о нем подробно. Сейчас скажем лишь о том, что адрес FTP-ресурса в Интернете выглядит следующим образом: ftp://ftp.netscape.com

TELNET

С помощью этого протокола вы можете подключиться к удаленному компьютеру как пользователь (если наделены соответствующими правами, то есть знаете имя пользователя и пароль) и производить действия над его файлами и приложениями точно так же, как если бы работали на своем компьютере.

Telnet является протоколом эмуляции терминала. Работа с ним ведется из командной строки. Если вам нужно воспользоваться услугами этого протокола, не стоит рыскать по дебрям Интернета в поисках подходящей программы. Telnet-клиент поставляется, например, в комплекте Windows 98.

Чтобы дать команду клиенту Telnet соединиться с удаленным компьютером, подключитесь к Интернету, выберите в меню Пуск (Start) команду Выполнить (Run) и наберите в строке ввода, например, следующее: telnet lib.ru

(Вместо lib.ru вы, разумеется, можете ввести другой адрес.) После этого запустится программа Telnet, и начнется сеанс связи.

WAIS

WAIS расшифровывается как Wide-Area Information Servers. Этот протокол был разработан для поиска информации в базах данных. Информационная система WAIS представляет собой систему распределенных баз данных, где отдельные базы данных хранятся на разных серверах. Сведения об их содержании и расположении хранятся в специальной базе данных - каталоге серверов. Просмотр информационных ресурсов осуществляется с помощью программы-клиента WAIS.

Поиск информации ведется, по ключевым словам, которые задает пользователь. Эти слова вводятся для определенной базы данных, и система находит все соответствующие им фрагменты текста на всех серверах, где располагаются данные этой базы. Результат представляется в виде списка ссылок на документы с указанием того, насколько часто встречается в данном документе искомое слово и все искомые слова в совокупности.

Даже в наши дни, когда систему WAIS можно считать морально устаревшей, специалисты во многих областях при проведении научных исследований тем не менее обращаются к ней в поисках специфической информации, которую не могут найти традиционными средствами.

Адрес ресурса WAIS в Интернете выглядит примерно так: wais://site.edu

Gorpher

Протокол Gopher - протокол уровня приложения, разработанный в 1991 году. До повсеместного распространения гипертекстовой системы World Wide Web Gopher использовался для извлечения информации (в основном текстовой) из иерархической файловой структуры. Gopher был провозвестником WWW, позволявшим с помощью меню передвигаться от одной страницы к другой, постепенно сужая круг отображаемой информации. Программы-клиенты Gopher имели текстовый интерфейс. Однако пункты меню Gopher могли указывать и не только на текстовые файлы, но также, например, на telnet-соединения или базы данных WAIS.

Gopher переводится как "суслик", что отражает славное университетское прошлое разработчиков этой системы. Студенческие спортивные команды Университета Миннесоты носили название Golden Gophers ("Золотые суслики").

ейчас ресурсы Gopher можно просматривать с помощью обычного Web-браузера, так как современные браузеры поддерживают этот протокол.

Адреса информационных ресурсов Gopher имеют примерно следующий вид: gopher://gopher.tc.umn.edu

WAP

WAP (Wireless Application Protocol) был разработан в 1997 году группой компаний Ericsson, Motorola, Nokia и Phone.com (бывшей Unwired Planet) для того, чтобы предоставить доступ к службам Интернета пользователям беспроводных устройств - таких, как мобильные телефоны, пейджеры, электронные органайзеры и др., использующих различные стандарты связи.

К примеру, если ваш мобильный телефон поддерживает протокол WAP, то, набрав на его клавиатуре адрес нужной Web-страницы, вы можете увидеть ее (в упрощенном виде) прямо на дисплее телефона. В настоящее время подавляющее большинство производителей устройств уже перешли к выпуску моделей с поддержкой WAP, который также продолжает совершенствоваться.

2. L2-сеть

Все семь уровней модели OSI можно условно разделить на две группы:

Media layers (уровни среды),

Host layers (уровни хоста).

Уровни группы Media Layers (L1, L2, L3) занимаются передачей информации (по кабелю или беспроводной сети), используются сетевыми устройствами, такими как коммутаторы, маршрутизаторы и т. п.

2 соответствует уровню связи (самый низкий уровень) в модели сети TCP/IP. Уровень 2 – это сетевой уровень, используемый для передачи данных между смежными узлами сети в сети wide area или между узлами одной и той же локальной сети.

Кадр – это блок данных протокола, наименьший блок битов в сети уровня 2. Кадры передаются и получаются от устройств в одной локальной сети (LAN). Unilke bits, кадры имеют определенные структуры и могут быть использованы для обнаружения ошибок, плоскость управления действий и так далее. Не все фреймы переносят данные пользователя. Сеть использует несколько кадров для управления самим соединением данных.

На уровне 2 одноадресная рассылка относится к отправке фреймов от одного узла к другому, в то время как многоадресная рассылка означает передачу трафика от одного узла на несколько узлов, а трансляция – передача фреймов всем узлам в сети. Домен широковещательной передачи – это логическое разделение сети, в которой все узлы этой сети могут быть разделены широковещательной передачей на уровне 2.

Сегменты LAN могут быть соединены на уровне кадров с помощью мостов. Формирование спонтанных каналов создает отдельные домены широковещательной передачи в локальной сети, создавая сети VLANs, которые являются независимыми логическими сетями, которые группируются вместе связанных устройств в отдельные сегменты сети. Группировка устройств в VLAN не зависит от местонахождения устройств в локальной сети. Без использования настройки и VLANs все устройства в сети Ethernet находятся в едином домене широковещательной передачи, и все устройства обнаруживают все пакеты в локальной сети.

Для увеличения длины общей сети, состоящей из различных сегментов кабеля, используются ретрансляторы. Ретранслятор является устройством 1-го уровня и работает только на физическом уровне. Сигналы, которые переносят информацию в пределах сети, могут пройти фиксированное расстояние до того момента, когда затухание создаст угрозу целостности данных.

Ретранслятор получает сигнал, и прежде, чем он становится слишком слабым или искаженным, восстанавливает первоначальный вид бита. Затем он передает регенерированный сигнал. Ретранслятор может увеличить физическую длину сети.

Уровень 2 содержит две подслоя:

Подслой логического управления каналом (LLC), ответственный за управление каналами связи и обработку трафика кадров.

Подслой управления доступом к средствам связи (MAC), который управляет доступом протокола к физической среде сети. Используя MAC-адреса, присвоенные всем портам коммутатора, несколько устройств на одном физическом соединении могут уникально идентифицировать друг друга.

Порты или интерфейсы коммутатора работают либо в режиме доступа, либо в режиме доступа с тегами, либо trunk-режим:

Порты режима доступа подключаются к сетевому устройству, например к настольному компьютеру, IP-телефону, принтеру, файловом серверу или камере безопасности. Сам порт принадлежит одной VLAN. Кадры, передаваемые через интерфейс доступа, являются обычными кадрами Ethernet. По умолчанию все порты коммутатора находятся в режиме доступа.

Порты в режиме доступа с тегами подключаются к сетевому устройству, например к настольному компьютеру, IP-телефону, принтеру, файловом серверу или камере безопасности. Сам порт принадлежит одной VLAN. Кадры, передаваемые через интерфейс доступа, являются обычными кадрами Ethernet. По умолчанию все порты коммутатора находятся в режиме доступа. Режим доступа с тегами облако, в частности, в сценариях, включая виртуальные компьютеры и виртуальные компьютеры. Поскольку на одном физическом сервере можно включить несколько виртуальных компьютеров, пакеты, генерируемые одним сервером, могут содержать агрегацию пакетов VLAN, полученных от различных виртуальных компьютеров на этом сервере. В этой ситуации помеченный режим доступа отражает пакеты, возвращаясь на физический сервер на тот же 9-й порт, когда адрес назначения пакета был завуалирован на этом 9-м порту. Пакеты также отражаются обратно на физический сервер 9-го порта, когда назначение еще не известно. Поэтому третий режим интерфейса (tagged access) имеет некоторые характеристики режима доступа и некоторые характеристики trunk-режим:

Порты магистрали обрабатывают трафик нескольких VLANs, мультиплексинг трафика для всех этих VLANs, подключенных к одному

физическому соединению. Магистральные интерфейсы обычно используются для соединения коммутаторов с другими устройствами или коммутаторами.

При настройке своей VLAN кадры, не переносимые тегами VLAN, отправляются по магистральным интерфейсам. Если имеется ситуация, когда пакеты передаются от устройства к коммутатору в режиме доступа, а затем необходимо отправить эти пакеты с коммутатора через магистральный порт, используйте режим native VLAN. Настройте единственную VLAN на порте коммутатора (который находится в режиме доступа) как родную VLAN. Затем магистральные порты коммутатора будут обрабатывать эти кадры иначе, чем другие помеченные пакеты. Например, если порт магистрали имеет три сети VLAN, 10, 20 и 30, которые назначены ему с VLAN 10 как родная VLAN, то кадры VLAN 10, которые покидают магистральные порты на другом конце, не имеют задатки 802.1Q (метки). Существует еще один параметр native VLAN. Коммутатор может добавить и удалить теги для непомеченных пакетов. Для этого сначала нужно настроить одну VLAN в качестве родной VLAN на порте, подключенном к устройству на границе сети. Затем назначьте метку VLAN ID единственной native VLAN на порту, подключенном к устройству. В конце добавьте VLAN ID к магистральному порту. Теперь, когда коммутатор получает непомеченный пакет, он добавляет указанный ID и отправляет, и получает тегируемые пакеты на магистральном порту, настроенном для примите эту VLAN.

В том числе и подуровни, второй уровень серия QFX поддерживает следующие функции:

Преодоление.

VLAN 802.1Q — также известная как маркировка VLAN. Этот протокол позволяет множественным сетям с мостами прозрачно использовать один и тот же физический сетевой физический соединение, добавляя теги VLAN к кадру Ethernet.

Расширение VLAN с уровня 2 на несколько коммутаторов с использованием протокола STP предотвращает распространение петель в сети.

Изучение MAC-адресов, включая обучение по VLAN MAC и подавление обучения на уровне 2. Этот процесс получает MAC-адреса всех узлов сети

Агрегирование физических уровней в группах процессов интерфейсов Ethernet для формирования интерфейса одного уровня соединения, также известного как группа агрегации физических соединений (LAG) или связка LAG.

3. L3-сеть

(Layer 3 или L3) фактически являются маршрутизаторами, которые реализуют механизмы маршрутизации (логическая адресация и выбор пути доставки данных (маршрута) с использованием протоколов маршрутизации (RIP v.1 и v.2, OSPF, BGP, проприетарные протоколы маршрутизации и др.) не в программном обеспечении устройства, а с помощью специализированных аппаратных средств (микросхем).

Маршрутизатор является наиболее распространенным сетевым устройством, относящимся к Уровню 3. Данные коммутаторы выполняет функции маршрутизации (логическую адресацию и выбор пути доставки) пакетов на IP-адрес получателя (Интернет-протокол). Коммутаторы уровня 3 проверяют IP-адреса источника и получателя каждого пакета данных в своей таблице IP-маршрутизации и определяют лучший адрес для последующей пересылки пакета (маршрутизатору или коммутатору). Если IP-адрес назначения не найден в таблице, пакет не будет отправлен до тех пор, пока не будет определен конечный муршрутизатор. По этой причине процесс маршрутизации осуществляется с определенной временной задержкой.

Коммутаторы уровня 3 (или многоуровневого коммутатора) имеют часть функций коммутаторов уровня 2 и маршрутизаторов. По сути, это три разных устройства, предназначенных для разных приложений, которые в значительной степени зависят от доступных функций. Однако, все три устройства также имеют часть общих функций.

Коммутатор уровня 2 VS Коммутатор уровня 3: в чём разница?

Основное различие между коммутаторами уровня 2 и уровня 3 — это функция маршрутизации. Коммутатор уровня 2 работает только с MAC-адресами, игнорируя IP-адреса и элементы более высоких уровней. Коммутатор уровня 3 выполняет все функции коммутатора уровня 2. Кроме того, он может осуществлять статическую и динамическую маршрутизацию. Это значит, что коммутатор уровня 3 имеет как таблицу MAC-адресов, так и таблицу маршрутизации IP-адресов, а также соединяет несколько устройств локальной вычислительной сети VLAN и обеспечивает маршрутизацию пакетов между различными VLAN. Коммутатор, который осуществляет только статическую маршрутизацию, обычно называется Layer 2+ или Layer 3 Lite. Помимо пакетов маршрутизации коммутаторы уровня 3 также включают в себя некоторые функции, требующие наличие информации о данных IP-адресов в коммутаторе, таких как маркирование трафика VLAN на основе IP-адреса вместо ручной настройки порта. Более того, коммутаторы уровня 3

имеют большую потребляемую мощность и повышенные требования безопасности.

4. L4-сеть

Уровни группы Host Layers (L4, L5, L6, L7) используются непосредственно на устройствах, будь то стационарные компьютеры или портативные мобильные устройства.

Четвертый уровень — это посредник между Host Layers и Media Layers, относящийся скорее к первым, чем к последним, его главной задачей является транспортировка пакетов. Естественно, при транспортировке возможны потери, но некоторые типы данных более чувствительны к потерям, чем другие. Например, если в тексте потеряются гласные, то будет сложно понять смысл, а если из видеопотока пропадет пара кадров, то это практически никак не скажется на конечном пользователе. Поэтому, при передаче данных, наиболее чувствительных к потерям на транспортном уровне используется протокол TCP, контролирующий целостность доставленной информации.

Для мультимедийных файлов небольшие потери не так важны, гораздо критичнее будет задержка. Для передачи таких данных, наиболее чувствительных к задержкам, используется протокол UDP, позволяющий организовать связь без установки соединения.

При передаче по протоколу TCP, данные делятся на сегменты. Сегмент — это часть пакета. Когда приходит пакет данных, который превышает пропускную способность сети, пакет делится на сегменты допустимого размера. Сегментация пакетов также требуется в ненадежных сетях, когда существует большая вероятность того, что большой пакет будет потерян или отправлен не тому адресату. При передаче данных по протоколу UDP, пакеты данных делятся уже на датаграммы. Датаграмма (datagram) — это тоже часть пакета, но ее нельзя путать с сегментом.

Главное отличие датаграмм в автономности. Каждая датаграмма содержит все необходимые заголовки, чтобы дойти до конечного адресата, поэтому они не зависят от сети, могут доставляться разными маршрутами и в разном порядке. Датаграмма и сегмент — это два PDU транспортного уровня модели OSI. При потере датаграмм или сегментов получаются «битые» куски данных, которые не получится корректно обработать.

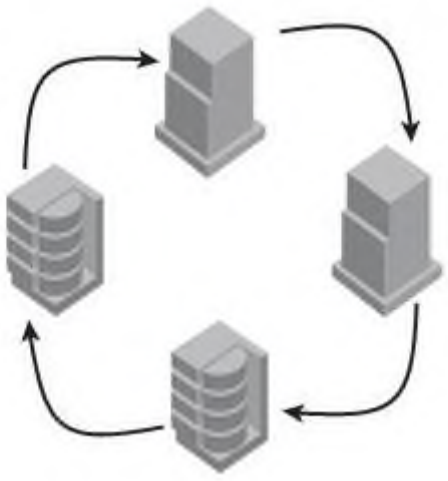
Первые четыре уровня — специализация сетевых инженеров, но с последними тремя они не так часто сталкиваются, потому что пятым, шестым и седьмым занимаются разработчики.

5. Сеть передачи данных (SAN)

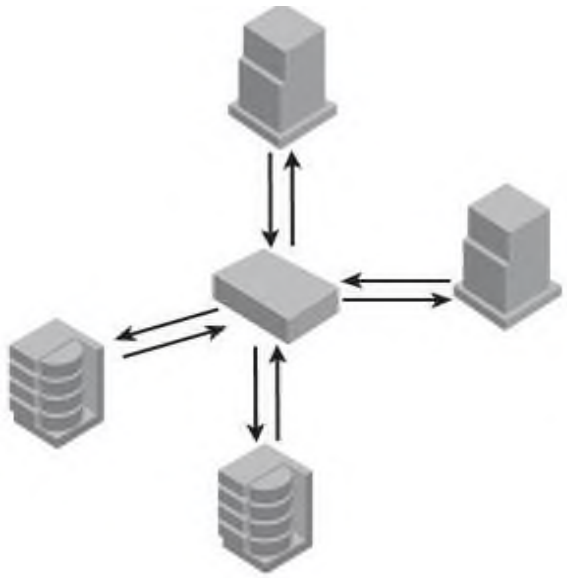
Архитектура SAN

Топология сети

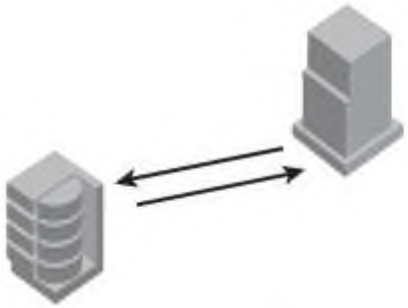
SAN является высокоскоростной сетью передачи данных, предназначенной для подключения серверов к устройствам хранения данных. Разнообразные топологии SAN (точка-точка, петля с арбитражной логикой (Arbitrated Loop) и коммутация) замещают традиционные шинные соединения «сервер — устройства хранения» и предоставляют по сравнению с ними большую гибкость, производительность и надежность. В основе концепции SAN лежит возможность соединения любого из серверов с любым устройством хранения данных, работающим по протоколу Fibre Channel. Принцип взаимодействия узлов в SAN с топологиями точка-точка или коммутацией показан на рисунках. В SAN с топологией Arbitrated Loop передача данных осуществляется последовательно от узла к узлу. Для того, чтобы начать передачу данных передающее устройство инициализирует арбитраж за право использования среды передачи данных (отсюда и название топологии – Arbitrated Loop).



Петля с арбитражной логикой



Коммутация



Точка-точка

Транспортную основу SAN составляет протокол Fibre Channel, использующий как медные, так и волоконно-оптические соединения устройств.

Компоненты SAN

Компоненты SAN подразделяются на следующие:

- Host Bus Adaptors (HBA);
- Ресурсы хранения данных;
- Устройства, реализующие инфраструктуру SAN;
- Программное обеспечение.

Host Bus Adaptors

HBA устанавливаются в серверы и осуществляют их взаимодействие с SAN по протоколу Fibre Channel. Стек протоколов Fibre Channel реализован внутри HBA. Наиболее известными производителями HBA являются компании Emulex, JNI, Qlogic и Agilent.

Ресурсы хранения данных

К ресурсам хранения данных относятся дисковые массивы, ленточные накопители и библиотеки с интерфейсом Fibre Channel. Многие свои возможности ресурсы хранения реализуют только будучи включенными в SAN. Так дисковые массивы высшего класса могут осуществлять репликацию данных между массивами по сетям Fibre Channel, а ленточные библиотеки могут реализовывать перенос данных на ленту прямо с дисковых массивов с интерфейсом Fibre Channel, минуя сеть и серверы (Serverless backup). Наибольшую популярность на рынке приобрели дисковые массивы компаний EMC, Hitachi, IBM, Compaq (семейство Storage Works, доставшееся Compaq от Digital), а из производителей ленточных библиотек следует упомянуть StorageTek, Quantum/ATL, IBM.

Устройства, реализующие инфраструктуру SAN

Устройствами, реализующими инфраструктуру SAN, являются коммутаторы Fibre Channel (Fibre Channel switches, FC switches), концентраторы (Fibre Channel Hub) и маршрутизаторы (Fibre Channel-SCSI routers). Концентраторы используются для объединения устройств, работающих в режиме Fibre Channel Arbitrated Loop (FC_AL). Применение концентраторов позволяет подключать и отключать устройства в петле без остановки системы, поскольку концентратор автоматически замыкает петлю в случае отключения устройства и автоматически размыкает петлю, если к нему было подключено новое устройство. Каждое изменение петли сопровождается сложным процессом её инициализации. Процесс инициализации многоступенчатый, и до его окончания обмен данными в петле невозможен.

Все современные SAN построены на коммутаторах, позволяющих реализовать полноценное сетевое соединение. Коммутаторы могут не только соединять устройства Fibre Channel, но и разграничивать доступ между устройствами, для чего на коммутаторах создаются так называемые зоны. Устройства, помещенные в разные зоны, не могут обмениваться информацией друг с другом. Количество портов в SAN можно увеличивать, соединяя коммутаторы друг с другом. Группа связанных коммутаторов носит название Fibre Channel Fabric или просто Fabric. Связи между коммутаторами называют Interswitch Links или сокращенно ISL.

Программное обеспечение

Программное обеспечение позволяет реализовать резервирование путей доступа серверов к дисковым массивам и динамическое распределение нагрузки между путями. Для большинства дисковых массивов существует простой способ определить, что порты, доступные через разные контроллеры, относятся к одному диску. Специализированное программное обеспечение поддерживает таблицу путей доступа к устройствам и обеспечивает отключение путей в случае аварии, динамическое подключение новых путей и распределение нагрузки между ними. Как правило, изготовители дисковых массивов предлагают специализированное программное обеспечение такого типа для своих массивов. Компания VERITAS Software производит программное обеспечение VERITAS Volume Manager, предназначенное для организации логических дисковых томов из физических дисков и обеспечивающее резервирование путей доступа к дискам, а также распределение нагрузки между ними для большинства известных дисковых массивов.

Используемые протоколы

В сетях хранения данных используются низкоуровневые протоколы:

- Fibre Channel Protocol (FCP), транспорт SCSI через Fibre Channel. Наиболее часто используемый на данный момент протокол. Существует в вариантах 1 Gbit/s, 2 Gbit/s, 4 Gbit/s, 8 Gbit/s и 10 Gbit/s.
- iSCSI, транспорт SCSI через TCP/IP.
- FCoE, транспортировка FCP/SCSI поверх "чистого" Ethernet.
- FCIP и iFCP, инкапсуляция и передача FCP/SCSI в пакетах IP.
- HyperSCSI, транспорт SCSI через Ethernet.
- FICON транспорт через Fibre Channel (используется только мейнфреймами).
- ATA over Ethernet, транспорт ATA через Ethernet.
- SCSI и/или TCP/IP транспорт через InfiniBand (IB).

Преимущества

- Высокая надёжность доступа к данным, находящимся на внешних системах хранения. Независимость топологии SAN от используемых СХД и серверов.
- Централизованное хранение данных (надёжность, безопасность).
- Удобное централизованное управление коммутацией и данными.
- Перенос интенсивного трафика ввода-вывода в отдельную сеть – разгрузка LAN.

- Высокое быстродействие и низкая латентность.
- Масштабируемость и гибкость логической структуры SAN
- Географические размеры SAN, в отличие от классических DAS, практически не ограничены.
- Возможность оперативно распределять ресурсы между серверами.
- Возможность строить отказоустойчивые кластерные решения без дополнительных затрат на базе имеющейся SAN.
- Простая схема резервного копирования – все данные находятся в одном месте.
- Наличие дополнительных возможностей и сервисов (снапшоты, удаленная репликация).
- Высокая степень безопасности SAN.

Совместное использование систем хранения как правило упрощает администрирование и добавляет изрядную гибкость, поскольку кабели и дисковые массивы не нужно физически транспортировать и перекоммутировать от одного сервера к другому.

Другим приемуществом является возможность загружать сервера прямо из сети хранения. При такой конфигурации можно быстро и легко заменить сбойный сервер, переконфигурировав SAN таким образом, что сервер-замена, будет загружаться с LUN'а сбойного сервера. Эта процедура может занять, например, полчаса. Идея относительно новая, но уже используется в новейших датацентрах.

Также сети хранения помогают более эффективно восстанавливать работоспособность после сбоя. В SAN может входить удаленный участок со вторичным устройством хранения. В таком случае можно использовать репликацию - реализованную на уровне контроллеров массивов, либо при помощи специальных аппаратных устройств. Поскольку каналы WAN на основе протокола IP встречаются часто, были разработаны протоколы Fibre Channel over IP (FCIP) и iSCSI с целью расширить единую SAN средствами сетей на основе протокола IP. Спрос на такие решения значительно возрос после событий 11 сентября 2001 года в США.

Недостатки

Все минусы сводятся только к высокой стоимости подобного рода решений. Российский рынок СХД в целом отстает от рынка западных развитых стран, особенно – в широком использовании сетей хранения данных. В частности, определенное влияние продолжают оказывать дефицит и высокая стоимость скоростных каналов связи.

6. NAT

Что такое НАТ — NAT

NAT — это специальный механизм, реализованный в сетях TCP/IP, который позволяет изменять айпи адреса пересылаемых пакетов, т. е. тех внутренних IP, которые присылаются на сетевой шлюз — в глобальные для дальнейшей отправки во внешний интернет. Также, такие пакеты называют транзитными.

Реализуется через маршрутизатор на программном уровне или настраивается самим провайдером. Полностью расшифровывается, как Network Address Translation и переводится на русский, как Преобразование Сетевых Адресов.

Простыми словами, работает это так. При создании сетей обычно используются частные IP вида: 10/8, 172.16/12, 192.168/16. Обмен пакетами между этими айпи внутри такой сети происходит напрямую. Но, чтобы выйти в глобальную паутину, такой IP должен быть преобразован в глобальный/публичный — именно этим и занимается NAT.



Именно НАТ позволяет вам из частной сети общаться с внешними ip адресами, за ее пределами. А между прочем, даже ваш роутер и подсоединенные к нему ПК, ноутбук, телевизор, смартфон и т. д. — это уже локальная частная сеть.

Интересно! Вот таким вот простыми способом, была решена проблема нехватки уникальных айпи IPv4 для каждого устройства. Т. е. в своей сети — свой IP, а во всемирной паутине — публичный, другой IP.

Типы NAT

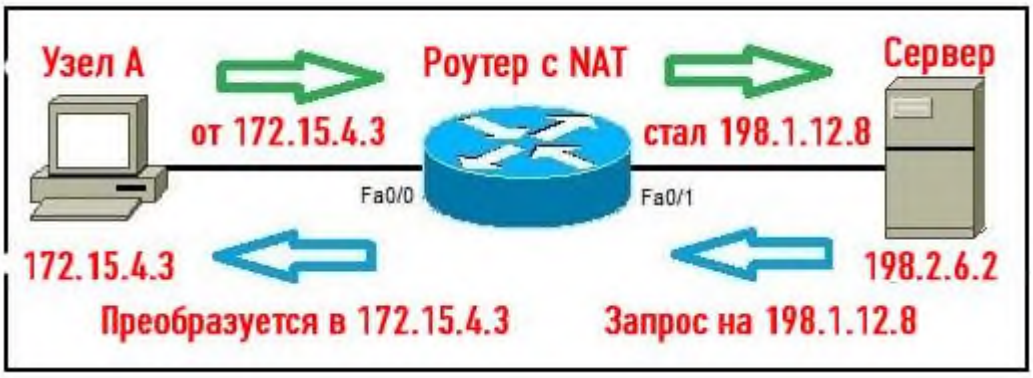
Статический, Static NAT

Берется один IP-адрес из внутренней сети и преобразовывается во внешний/публичный и все запросы, которые будут на него посылаться извне, будут пересылаться именно на этот внутренний айпи. Т. е. для всемирной паутины он будет выглядеть полностью, как белый/публичный IP.

Используется чаще всего в корпоративных сетях, когда необходимо, чтобы какой-либо IP всегда был доступен из глобальной паутины.

Работает это так:

- Есть во внутренней сети айпи 172.15.4.3, он обращается ко внешнему серверу на 198.2.6.2, отправляет пакет данных вначале естественно на шлюз.
- Шлюз 172.15.4.1, на котором настроен NAT преобразует его в белый/внешний 198.1.12.8.
- По глобальном интернете пакеты передаются по этому IP-адресу и обратно приходят к нему же.
- На маршрутизаторе с НАТ есть указание, все пакеты данных, которые приходят на 198.1.12.8 перенаправлять на 172.15.4.3.



Вот и все. Если у вас несколько роутеров, то такую процедуру можно проделать со всеми ними.

Динамический, Dynamic NAT

Если вы имеете определенное количество белых/внешних IP, например, ваш поставщик интернета предоставил вам сеть 199.49.101.0/28 — с 16 айпи. Первый и конечный адреса всегда заняты. Еще два мы возьмем для оборудования на функционирование маршрутизации. И остается их 12, которые можно применять для NAT и пускать посредством них свои устройства.

Тут все практически также, как и со статичным НАТ, также 1 внутренний айпи привязан к 1 внешнему, но с той лишь разницей, что адреса теперь будут

выдаваться динамически каждому нуждающемуся пользователю во внутренней сети, а не одному определенному узлу.

Все бы хорошо, но, если, например, адресов 12, а пользователей 300? Те, кто успел, те и смогут их использовать.

NAT Overload

Этот вид еще имеет другие названия такие как: Many-to-One NAT, Port Address Translation (PAT) и IP Masquerading.

Как вы знаете, чаще всего провайдер выдает вам лишь один IP адрес, который может быть статическим или динамическим. Но дома могут быть множества устройств с разными программами, которым необходим доступ во всемирную паутину и как определить, какое из них отправило запрос, какое приложение? Проблема эта решается использованием портов, по этой ссылке очень подробно описана эта тема.

Важно! Порт пишется сразу после айпи через двоеточие, например: 176.57.209.9:443. И такая связка называется — сокет.



Как это будет выглядеть, когда NAT подменяет один IP на другой? Так, например, с двух устройств с внутренними адресами 173.52.6.7 и 173.52.4.5, которые должны заменяться на: 198.50.100.9 и 198.50.100.11 будет выполнен запрос на какой-либо сервер в интернете — это будет смотреться так:

Устройство 1 — отправитель: IP — 173.52.6.7, порт: 21784; получатель: IP — 192.16.5.3, порт: 80

Устройство 2 — отправитель: IP — 173.52.4.5, порт: 32714; получатель: IP — 192.16.5.3, порт: 80

Что происходит дальше? NAT берет IP-пакет первого узла, вынимает его TCP сегмент, при этом узнавая порт. У этого айпи внешний адрес 198.50.100.9 — на него он меняется при выходе в глобальный интернет. Далее происходит выбор порта, к примеру, 11837 и все данные уровня приложений упаковываются в новый TCP сегмент. Порт адресата естественно остается —

80, а вот того, кто отправляет заменяется с 21784 на 11837. TCP сегмент вставляется в новый IP-пакет со сменой самого адреса на 198.50.100.9. Такая же процедура происходит и со вторым хостом только отбирается следующий незанятый PORT.

Теперь данные, отправляемые в интернет, будут выглядеть так:

Устройство 1 — отправитель: IP — 198.50.100.9, порт: 11837; получатель: IP — 192.16.5.3, порт: 80

Устройство 2 — отправитель: IP — 198.50.100.11, порт: 27289; получатель: IP — 192.16.5.3, порт: 80

В NAT-таблицу записываются значения отправителя и получателя:

Локальный сокет отправителя 1: 173.52.6.7:21784; сокет получателя: 192.16.5.3:80

Глобальный сокет отправителя 1: 198.50.100.9:11837; сокет получателя: 192.16.5.3:80

Локальный сокет отправителя 2: 173.52.4.5:32714; сокет получателя: 192.16.5.3:80

Локальный сокет отправителя 2: 198.50.100.11:27289; сокет получателя: 192.16.5.3:80

Когда приходит ответ от сервера происходит сопоставление данных с таблицей и пакеты данных доставляются именно тому, кто совершил запрос.

Преимущества NAT

Под одним внешним IPv4 адресом может сидеть в глобальной паутине множество пользователей одновременно.

Скрывает ваш настоящий внутренний IP в частной сети и показывает лишь внешний. Так, все устройства из вне видят только ваш общедоступный айпи.

В определенной степени выполняет функции файрвола, не позволяя соединениям извне достигать основного компьютера. Т. е. если на устройство с НАТ извне приходит пакет, который не ожидался — то он категорически не будет допущен.

Роутер с поддержкой НАТ может быть настроен сразу с несколькими внешними/белыми IP, это еще называется — пул НАТ.

Недостатки NAT

Невысокая скорость передачи данных для протоколов реального времени, например, для VoIP. Т. к. когда НАТ переделывает заголовки у каждого IPv4 адреса — происходят задержки.

Проблемы с идентификацией, т. к. под одним IP может находиться сразу несколько человек.

Для пиринговых сетей нужна дополнительная настройка маршрутизации, т. к. приложению еще нужно и принимать соединения.

Если много пользователей с одним айпи зайдут на какой-либо сервис — это может вызывать подозрение на Do S-атаку.

7. Виртуальные частные сети (VPN): OpenVPN, L2TP/Ipsec, SSTP, PPTP

Что такое VPN

Виртуальная частная сеть защищает личные или корпоративные данные при их передаче через интернет или другие сети. Для этого VPN использует виртуальный туннель, а передаваемые данные зашифровывает.

Сам VPN действует как сетевой интерфейс для клиента и прозрачен для операционной системы, приложений и пользователей, получающих к нему доступ.



Для чего используется VPN

Прежде всего, виртуальная частная сеть обеспечивает конфиденциальную передачу данных. VPN защищает информацию во время ее передачи через общедоступную сеть. Если злоумышленники каким-то образом перехватят данные, то не смогут их расшифровать и использовать в своих целях.

VPN также используется в компаниях, где сотрудники работают удаленно — на дому, в публичных местах и даже в транспорте. Домашние сети

и личные устройства не так хорошо защищены, как офисные сети и рабочие места. Вдобавок работники используют публичные сети, которые пересылают данные незашифрованными. Использование VPN предоставляет безопасное подключение к внутренним системам организации. Все конфиденциальные файлы или интеллектуальная собственность защищены при передаче через интернет.

VPN обязателен для любой организации, чьи сотрудники выезжают за границу по работе. В целях безопасности многие компании ограничивают доступ к своей внутренней сети списком определенных IP-адресов. Вдобавок в ряде стран некоторые необходимые для работы веб-сайты блокируются. VPN решает эту проблему.

Еще одна распространенная проблема безопасности — DNS-утечки. Из-за неправильной настройки запросы могут отправляться в незашифрованном виде в обход VPN. Благодаря этому злоумышленник способен получить информацию об IP-адресе устройств или сети, посещенных веб-сайтах и их IP-адресах. Эта информация поможет мошеннику создать фишинговую кампанию, чтобы украсть данные для входа в систему.

Недостатки VPN

Основной недостаток один — скорость. VPN-соединение — это еще одна остановка на пути передаваемых данных. Вдобавок шифрование требует дополнительного времени. В результате каждое VPN-соединение немного замедляет скорость работы.

В целом скорость VPN-соединения зависит от скорости соединения обеих конечных точек. Например, пользователь, получающий доступ к корпоративной сети через VPN, ограничен скоростью:

подключения своей рабочей машины к интернету,

подключения интернета к серверу VPN,

подключения сервера VPN к доступным ресурсам.

Низкая скорость VPN-сервиса может привести к значительному падению пропускной способности.

Как устроена виртуальная частная сеть

Есть несколько типов VPN, каждый из них может использовать разные протоколы и способы шифрования. Выбор зависит от целей использования VPN.

Типы организации VPN-соединения

Основных типов организации виртуальной частной сети три: Remote Access VPN (шлюз защищенного удаленного доступа), Site to Site VPN (соединение точка-точка или роутер-роутер) и Client to Provider.

Remote Access VPN

Этот тип VPN позволяет отдельным пользователям устанавливать безопасные соединения с удаленной компьютерной сетью. Они получают доступ к защищенным ресурсам в этой сети через интернет, как если бы подключались напрямую к серверам сети. Такой способ подходит организациям, где работают сотни удаленных сотрудников.

Другое название этого типа VPN — виртуальная частная сеть с коммутируемым доступом (VPDN). Слово «коммутируемый» напоминает, что в своей самой ранней форме VPN с удаленным доступом требовал подключения к серверу с использованием аналоговой телефонии.

Как работает

Для этого типа нужны два компонента. Первый — сервер доступа к сети NAS. Он может быть выделенным сервером или одним из нескольких программных приложений, работающих на общем сервере. NAS требует, чтобы пользователь предоставил свои учетные данные для входа в VPN. Аутентификацию NAS проводит самостоятельно или использует для этого отдельный сервер.

Второй компонент виртуальных частных сетей удаленного доступа — клиентское ПО. Оно устанавливается на компьютеры пользователей. Большинство операционных систем сегодня имеют встроенное ПО для подключения к VPN. Однако, некоторые VPN могут потребовать от пользователей установки специального приложения. Клиентское ПО устанавливает туннелированное соединение с NAS-сервером. Оно также управляет шифрованием, необходимым для обеспечения безопасности соединения.

Крупные корпорации или предприятия с квалифицированным ИТ-персоналом обычно развертывают и обслуживают свои собственные виртуальные частные сети Remote Access VPN.

Site to site VPN

Такой тип VPN подходит крупным корпорациям с филиалами по всей стране или миру.

Как работает

Существует два типа Site to Site VPN — интранет и экстранет.

Интранет

Если у компании есть один или несколько географически удаленных офисов, которые надо объединить в единую частную сеть, то создается интрасеть VPN. Она подключает каждую отдельную локальную сетку к общей сети организации.

Экстранет

Он создается для работы нескольких компаний, которые хотят совместно использовать часть ресурсов, а другую часть оставить приватной. На основе экстрасети каждая организация подключается к VPN и выбирает ресурсы, которые хочет сделать доступным для другой компании. Такая сеть VPN позволяет организациям работать вместе в безопасной общей сетевой среде, предотвращая доступ к их отдельным интрасетям.

Client-to-provider

Это способ подключения для пользователей, использующих небезопасную общедоступную сеть Wi-Fi — например, в кафе, аэропорту или отеле.

Также этот способ подключения предназначен для тех, кому важно сохранить конфиденциальность своих данных. Благодаря ему пользователь шифрует трафик от своего интернет-провайдера.

Как работает

Для предотвращения перехвата трафика пользователь устанавливает VPN-соединение с провайдером VPN, а тот уже перенаправляет трафик в интернет. Легко перехватываемый локальный беспроводной трафик шифруется на всем пути к провайдеру, который затем безопасно подключается к Интернету. Это снижает вероятность атаки посредника (MITM) или сниффера.

VPN-клиенты

Для подключения к защищенной сети нужен VPN-клиент — программное обеспечение, работающее на выделенном устройстве. Оно действует как туннельный интерфейс для нескольких подключений и избавляет каждый компьютер от необходимости запускать собственное клиентское ПО VPN. Выбор способа подключения зависит от целей использования.

Автономный

На удаленной конечной точке устанавливается специализированное ПО. При запуске оно создает зашифрованное соединение с виртуальной частной сетью. Чтобы установить VPN-соединение, конечная точка должна запустить VPN-клиент и подключиться к другой конечной точке. Такой тип подключения

часто встречается в общедоступных VPN-сервисах. Обычно пользователь самостоятельно загружает VPN-клиент для подключения.

Встроенный в операционную систему

Операционные системы Windows, iOS, MacOS, Android и Linux, позволяют подключаться к удаленному VPN-серверу при условии, что удаленная конечная точка поддерживает тот же протокол и конфигурацию VPN.

VPN-сервер действует как шлюз и роутер на границе локальной сети, либо на границе интернета. Он отвечает за развертывание пакетов и их переупаковку для передачи в локальной сети или в интернете.

Этот вариант подключения обычно используется в корпоративной среде. В крупных компаниях есть штатные ИТ-специалисты, которые способны устанавливать, настраивать и поддерживать инсталляцию клиентов и серверы VPN.

Router VPN

Решения для частных пользователей, у которых роутер поддерживает функцию настройки VPN. Такой способ позволяет подключить к VPN несколько устройств сразу. Минус такого подхода — сложность настройки, требующая хороших технических знаний.

VPN-расширение браузера

Несмотря на название, такие расширения не являются VPN-сервисами. Они не поддерживают защиту трафика — ключевую функцию VPN. Она обеспечивается оборудованием, а не браузером. Браузерное расширение только может при помощи прокси-сервера подменить IP-адрес пользователя. Этого достаточно, чтобы заходить на сайты, заблокированные Роскомнадзором. А вот скрыться от слежки не получится. Например, руководитель легко узнает на каких сайтах сидит сотрудник.

VPN-протоколы

Протоколы служат для создания защищенного соединения между VPN-клиентом и VPN-сервером и отличаются разным уровнем безопасности и производительности. Одни лучше работают на мобильных устройствах, другие предназначены для крупных корпоративных сетей.

OpenVPN

Протокол с открытым исходным кодом и высоким уровнем безопасности, благодаря чему, стал одним из самых популярных. Аутентификация пользователей возможна несколькими способами: при помощи предустановленного ключа, сертификата или логина и пароля. У OpenVPN нет коробочного решения — каждый провайдер VPN разрабатывает

свое клиентское ПО для работы с ним. Поэтому протокол работает на всех операционных системах и совместим с сервисами облачных провайдеров.

PPTP

PPTP — протокол туннелирования точка-точка. Он все еще используется, хотя и устарел. Обычно он работает в бесплатных VPN-сервисах. Протокол медленно восстанавливается после обрыва соединения, а уровень его безопасности ниже, чем у прочих.

L2TP/IPSec

Комбинация двух протоколов — L2TP это усовершенствованная версия PPTP, а IPSec отвечает за аутентификацию и шифрование. У L2TP/IPSec высокий уровень безопасности, но скорость передачи данных ниже, чем у остальных.

IKEv2/IPSec

Это усовершенствованная версия протокола L2TP. IKEv2 хорошо подходит для мобильных устройств — он устойчив к частой смене сетей.

SSTP

Это протокол безопасного туннелирования сокетов. Протокол входит в состав ОС Windows. Шифрование обеспечивает SSL-протокол, а аутентификацию — сразу три протокола: — SSL, PPP и SSTP.

WireGuard

Новый и пока малораспространенный протокол. У него простой код, поэтому в нем легче обнаружить и исправить уязвимости. Разработчики предлагают его в качестве замены протоколам IPsec и OpenVPN.

Шифрование данных в VPN

Оно обеспечивает конфиденциальность данных, передаваемых через частную виртуальную сеть.

Зашифрованную информацию невозможно прочесть без ключа, который известен только VPN-серверу и компьютеру.

Есть два способа шифрования трафика.

Симметричное шифрование

Все пользователи, или точнее, компьютеры, используют один и тот же ключ, предназначенный для шифрования и для дешифрования сообщения.

Шифрование с открытым ключом

Каждый компьютер имеет пару ключей: закрытый и открытый. Закрытым ключом он шифрует отправляемые данные, а открытым расшифровывает то, что получает с других ПК.

Вместе с шифрованием в VPN применяется протокол безопасности IPSec, который обеспечивает дополнительную защиту.

IPSec

Широко используемый протокол для защиты трафика в IP-сетях. IPSec может шифровать данные между различными устройствами:

роутера с роутером,

роутера с межсетевым экраном,

компьютера с роутером,

компьютера с сервером.

IPSec состоит из двух подпротоколов, которые предоставляют инструкции, необходимые VPN для защиты своих пакетов: ESP и AH. Описывать их тут не будем.

Сетевые устройства используют IPSec-протокол в туннельном режиме — создают виртуальный туннель между двумя сетями.

Компьютеры на каждом конце туннеля шифруют отправляемые данные и расшифровывают их после получения.

VPN и виртуальное частное облако (VPC)

VPN создает защищенное соединение локальной сети с сервисами облачного провайдера. Как правило, провайдеры предлагают своим клиентам публичное или частное облако — Virtual Private Service (VPC). Если в публичном облаке все клиенты совместно используют выделяемые ресурсы, то в частном каждый получает отдельные изолированные мощности. Проще говоря, Virtual Private Service — это облако в облаке.

8. Firewall

Файрволл, или брандмауэр, — термин для обозначения технических и программных средств, которые обрабатывают входящий и исходящий сетевой трафик. Проходящие данные проверяются на соответствие набору заданных правил и могут быть заблокированы от дальнейшей передачи.

Оба названия были заимствованы из иностранных языков и означают одно и то же: «противопожарный щит/стена» — в переводе с английского («firewall») и немецкого («brandmauer»), соответственно.

Как и проистекает из изначальной формулировки, фаервол предназначен для защиты внутренней информационной среды или ее отдельных частей от некоторых внешних потоков, и наоборот, предохраняет от прохождения отдельных пакетов вовне — например, в интернет. Фаерволы позволяют отфильтровывать подозрительный и вредоносный трафик, в том числе пресекать попытки взлома и компрометации данных.

При правильной настройке сетевой щит позволяет пользователям сети иметь доступ ко всем нужным ресурсам и отбрасывает нежелательные

соединения от хакеров, вирусов и других вредоносных программ, которые пытаются пробиться в защищенную среду.

Межсетевой экран нового поколения был определен аналитиками Gartner как технология сетевой безопасности для крупных предприятий, включающая полный набор средств для проверки и предотвращения проникновений, проверки на уровне приложений и точного управления на основе политик.

Если организация изучает возможность использования межсетевого экрана нового поколения, то самое главное — определить, обеспечит ли такой экран возможность безопасного внедрения приложений во благо организации. На первом этапе вам потребуется получить ответы на следующие вопросы:

Позволит ли межсетевой экран нового поколения повысить прозрачность и понимание трафика приложений в сети?

Можно ли сделать политику управления трафиком более гибкой, добавив дополнительные варианты действий, кроме разрешения и запрета?

Будет ли ваша сеть защищена от угроз и кибератак, как известных, так и неизвестных?

Сможете ли вы систематически идентифицировать неизвестный трафик и управлять им?

Можете ли вы внедрять необходимые политики безопасности без ущерба производительности?

Будут ли сокращены трудозатраты вашей команды по управлению межсетевым экраном?

Позволит ли это упросить управление рисками и сделать данный процесс более эффективным?

Позволят ли внедряемые политики повысить рентабельность работы предприятия?

В случае положительного ответа на вышеприведенные вопросы можно сделать следующий шаг и обосновать переход со старых межсетевых экранов на межсетевые экраны нового поколения. После выбора поставщика или узкого круга поставщиков, выполненного с помощью заявки, последует этап оценки физических функций межсетевого экрана, выполняемой с применением трафика различных типов и комбинаций, а также объектов и политик, которые точно передают особенности бизнес-процессов организации.

Основной задачей сетевого экрана является защита компьютерных сетей или отдельных узлов от несанкционированного доступа. Также сетевые

экраны часто называют фильтрами, так как их основная задача — не пропускать (фильтровать) пакеты, не подходящие под критерии, определённые в конфигурации.

Защита корпоративных сетей базируется на межсетевых экранах, которые теперь должны не только фильтровать потоки информации по портам, но и контролировать данные, передаваемые по наиболее популярным из них. По оценкам экспертов SophosLabs компании Sophos, до 80% нападений совершаются с использованием веб-браузера по протоколам HTTP или HTTPS, однако простым фильтрованием этих протоколов проблему не решить. Таким образом, к новому поколению межсетевых экранов, совмещенных с системами обнаружения вторжений, появляются новые требования.

Межсетевые экраны нового поколения рекомендуется внедрять для решения следующих задач:

- контроля отдельных веб-приложений;
- обнаружения вторжений по наиболее популярным протоколам, таким как HTTP, SMTP и POP3;
- создания VPN-соединений для удаленного подключения мобильных пользователей;
- оптимизации сетевого взаимодействия.

Следует отметить, что наличие систем обнаружения вторжений требуется также при обработке персональных данных, защите банковских и платежных систем, а также и других сложных информационных инфраструктур. Совмещение же их с межсетевыми экранами очень удобно и выгодно для пользователей.

Другие названия

Брандма́уэр (нем. Brandmauer) — заимствованный из немецкого языка термин, являющийся аналогом английского firewall в его оригинальном значении (стена, которая разделяет смежные здания, предохраняя от распространения пожара). Интересно, что в области компьютерных технологий в немецком языке употребляется слово «firewall».

Файрволл, файрво́л, файерво́л, фаерво́л — образовано транслитерацией английского термина firewall, эквивалентного термину межсетевой экран, в настоящее время не является официальным заимствованным словом в русском языке [источник не указан 169 дней].

История межсетевых экранов

Файрволлом может быть как программное средство, так и комплекс ПО и оборудования. И поначалу они были чисто железными, как и давшие им название противопожарные сооружения.

В контексте компьютерных технологий термин стал применяться в 1980-х годах. Интернет тогда был в самом начале своего применения в глобальных масштабах.

Есть мнение, что, прежде чем название файрволла пришло в реальную жизнь, оно прозвучало в фильме «Военные игры» 1983 г., где главный герой — хакер, проникший в сеть Пентагона. Возможно, это повлияло на заимствование и использование именно такого именования оборудования.

Первыми файрволлами можно назвать маршрутизаторы, которые защищали сети в конце 1980-х. Все передаваемые данные проходили сквозь них, поэтому логично было добавить им возможность фильтрации пакетов.

Функции сетевых экранов

Современная корпоративная сеть – не замкнутое информационное пространство. Зачастую это распределенная сеть, связанная с внешним ЦОДом, использующая облака и периферию, состоящая из множества сегментов. Современный корпоративный межсетевой экран должен обладать соответствующими функциями для ее защиты.

Разновидности сетевых экранов

Сетевые экраны подразделяются на различные типы в зависимости от следующих характеристик:

обеспечивает ли экран соединение между одним узлом и сетью или между двумя или более различными сетями;

происходит ли контроль потока данных на сетевом уровне или более высоких уровнях модели OSI;

отслеживаются ли состояния активных соединений или нет.

В зависимости от охвата контролируемых потоков данных сетевые экраны делятся на:

- традиционный сетевой (или межсетевой) экран — программа (или неотъемлемая часть операционной системы) на шлюзе (сервере, передающем трафик между сетями) или аппаратное решение, контролирующие входящие и исходящие потоки данных между подключенными сетями.
- персональный сетевой экран — программа, установленная на пользовательском компьютере и предназначенная для защиты от несанкционированного доступа только этого компьютера.

Вырожденный случай — использование традиционного сетевого экрана сервером, для ограничения доступа к собственным ресурсам.

В зависимости от уровня, на котором происходит контроль доступа, существует разделение на сетевые экраны, работающие на:

- сетевом уровне, когда фильтрация происходит на основе адресов отправителя и получателя пакетов, номеров портов транспортного уровня модели OSI и статических правил, заданных администратором;
- сеансовом уровне (также известные как stateful) — отслеживающие сеансы между приложениями, не пропускающие пакеты нарушающих спецификации TCP/IP, часто используемых в злонамеренных операциях — сканировании ресурсов, взломах через неправильные реализации TCP/IP, обрыв/замедление соединений, инъекция данных.
- уровне приложений, фильтрация на основании анализа данных приложения, передаваемых внутри пакета. Такие типы экранов позволяют блокировать передачу нежелательной и потенциально опасной информации, на основании политик и настроек.

Некоторые решения, относимые к сетевым экранам уровня приложения, представляют собой прокси-серверы с некоторыми возможностями сетевого экрана, реализуя прозрачные прокси-серверы, со специализацией по протоколам. Возможности прокси-сервера и многопротокольная специализация делают фильтрацию значительно более гибкой, чем на классических сетевых экранах, но такие приложения имеют все недостатки прокси-серверов (например, анонимизация трафика).

В зависимости от отслеживания активных соединений сетевые экраны бывают:

- stateless (простая фильтрация), которые не отслеживают текущие соединения (например, TCP), а фильтруют поток данных исключительно на основе статических правил;
- stateful, stateful packet inspection (SPI) (фильтрация с учётом контекста), с отслеживанием текущих соединений и пропуском только таких пакетов, которые удовлетворяют логике и алгоритмам работы соответствующих протоколов и приложений. Такие типы сетевых экранов позволяют эффективнее бороться с различными видами DoS-атак и уязвимостями некоторых сетевых протоколов. Кроме того, они обеспечивают функционирование таких протоколов, как H.323, SIP, FTP и т. п., которые используют сложные схемы передачи данных между адресатами, плохо поддающиеся описанию статическими правилами, и, зачастую, несовместимых со стандартными, stateless сетевыми экранами.

Хронология событий

2020: 53% ИБ-экспертов считают межсетевые экраны бесполезными

29 октября 2020 года стало известно, что специалисты по информационной безопасности (ИБ) стали отказываться от использования межсетевых экранов. Большинство мотивируют это тем, что в современном мире они больше не могут обеспечить требуемый уровень защиты, сообщили CNews 29 октября 2020 года.

Потерю ИБ-специалистами доверия к брандмауэрам подтвердило исследование компании Ponemon Institute, с 2002 г. работающей в сфере информационной безопасности, проведенное совместно с компанией Guardicore из той же отрасли. Ее сотрудники опросили 603 ИБ-специалиста в американских компаниях и выяснили, что подавляющее большинство респондентов негативно отзываются о межсетевых экранах, в настоящее время используемых в их компаниях. 53% из них активно ищут другие варианты защиты сетей и устройств в них, попутно частично или полностью отказываясь от файрволлов из-за их неэффективности, высокой стоимости и высокой сложности.

Согласно результатам опроса, 60% его участников считают, что устаревших межсетевых экранов нет необходимых возможностей для предотвращения атак на критически важные бизнес-приложения. Столько же респондентов посчитали, что устаревшие межсетевые экраны демонстрируют свою неэффективность для создания сетей с нулевым доверием (zero trust).

76% специалистов, участвовавших в опросе Ponemon Institure, пожаловались на то, что при использовании устаревших файрволлов им требуется слишком много времени для защиты новых приложений или изменения конфигурации в существующих программах.

62% опрошенных специалистов считают, что политики контроля доступа в межсетевых экранах недостаточно детализированы, что ограничивает их способность защищать наиболее ценную информацию. 48% респондентов подчеркнули также, что внедрение межсетевых экранов занимает слишком много времени, что увеличивает их итоговую стоимость и время окупаемости.

Согласно отчету, в то время как 49% респондентов в какой-то степени внедрили модель безопасности Zero Trust, 63% считают, что устаревшие межсетевые экраны их организаций не могут обеспечить нулевое доверие в корпоративной сети. 61% респондентов отметили, что межсетевые экраны их организаций не могут предотвратить взлом дата-серверов компании, в то время как 64% считают, что устаревшие межсетевые экраны неэффективны против многих современных видов атак, включая атаки при помощи

программ-вымогателей. Большинство опрошенных специалистов заявили, что файрволлы бесполезны, когда вопрос касается и облачной безопасности. 61% уверены в их неэффективности при защите данных в облаке, а 63% считают, что нет смысла пытаться использовать их для обеспечения безопасности критически важных облачных приложений.

9. Высокоуровневые сетевые протоколы

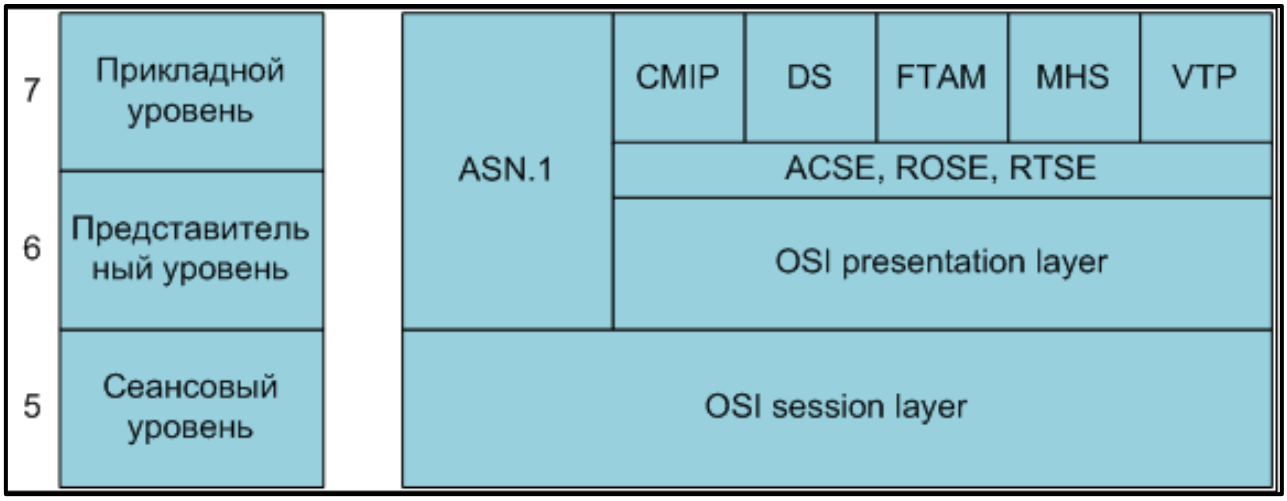
Высокоуровневые протоколы постоянно разрабатываются и совершенствуются. В этом нет ничего плохого, даже наоборот: всегда существует возможность придумать новый, более эффективный способ передачи данных.

Как правило, высокоуровневые протоколы реализуются в виде драйверов к сетевому оборудованию.

Существует множество различных протоколов. Одни из них узконаправленные, другие применяются широко. Протоколы разрабатываются несколькими фирмами, поэтому не удивительно, что каждая из них продвигает собственный набор. Эти наборы по умолчанию несовместимы, однако существуют протоколы-мосты, благодаря которым в одной операционной системе возможно использование нескольких несовместимых протоколов.

Не все протоколы могут применяться в одинаковых условиях. Иногда использование одного протокола выгодно для небольшой группы компьютеров, но совершенно не рационально для большого количества машин с несколькими маршрутизаторами и подключением к Интернету.

Основные протоколы высших уровней OSI представлены на рис. ниже



Сеансовый уровень

Протоколы сеансового уровня OSI преобразуют в сеансы потоки данных, поставляемых четырьмя низшими уровнями, путем реализации различных управляющих механизмов. В число этих механизмов входит ведение учета, управление диалогом (т. е. определение, кто и когда может говорить) и согласование параметров сеанса.

Управление диалогом сеанса реализуется путем использования маркера (token), обладание которым обеспечивает право на связь. Маркер можно запрашивать, и конечным системам ES могут быть присвоены приоритеты, обеспечивающие неравноправное пользование маркером.

Представительный уровень

Представительный уровень OSI, как правило, является просто проходным протоколом для информации из соседних уровней. Хотя многие считают, что Abstract Syntax Notation 1 (ASN.1) (Абстрактное представление синтаксиса) является протоколом представительного уровня OSI, ASN.1 используется для выражения форматов данных в независимом от машины формате. Это позволяет осуществлять связь между прикладными задачами различных компьютерных систем способом, прозрачным для этих прикладных задач.

Прикладной уровень

Прикладной уровень OSI включает действующие протоколы прикладного уровня, а также элементы услуг прикладного уровня (application service elements - ASE). ASE обеспечивают легкую связь протоколов прикладного уровня с низшими уровнями. Тремя наиболее важными ASE являются Элемент услуг управления ассоциацией (Association Control Service Element - ACSE), Элемент услуг получения доступа к операциям отдаленного устройства (Remote Operations Service Element - ROSE) и Элемент услуг надежной передачи (Reliable Transfer Service Element - RTSE). При подготовке к связи между двумя протоколами прикладного уровня ACSE объединяет их имена друг с другом. ROSE реализует родовой (generic) механизм "запрос/ответ", который разрешает доступ к операциям отдаленного устройства способом, похожим на вызовы процедуры обращений к отделенной сети (remote procedure calls - RPC). RTSE способствует надежной доставке, делая конструктивные элементы сеансового уровня легкими для использования.

Прикладные протоколы

За долгие годы использования в сетях различных стран и организаций стек TCP/IP накопил большое количество протоколов и сервисов прикладного

уровня. К ним относятся такие широко используемые протоколы, как протокол копирования файлов FTP, протокол эмуляции терминала telnet, почтовый протокол SMTP, используемый в электронной почте сети Internet, гипертекстовые сервисы доступа к удаленной информации, такие как WWW и многие другие.

Протокол пересылки файлов FTP

До появления службы WWW сетевая файловая служба на основе протокола FTP (File Transfer Protocol — протокол передачи файлов), описанная в спецификации RFC 959, долгое время была самой популярной службой доступа к удаленным данным в Интернете и

корпоративных IP-сетях. FTP-серверы и FTP-клиенты имеются практически в каждой ОС, кроме того, для доступа ко все еще популярным FTP-архивам используются FTP-клиенты, встроенные в браузеры.

Протокол FTP позволяет целиком переместить файл с удаленного компьютера на локальный, и наоборот. FTP также поддерживает

несколько команд просмотра удаленного каталога и перемещения по каталогам удаленной файловой системы. Поэтому FTP особенно

удобно использовать для доступа к тем файлам, данные которых нет смысла просматривать удаленно, а гораздо эффективней целиком переместить на клиентский компьютер (например, файлы исполняемых модулей приложений).

Для того, чтобы обеспечить надежную передачу, FTP использует в качестве транспорта протокол с установлением соединений - TCP. Кроме пересылки файлов протокол FTP предлагает и другие услуги. Так, пользователю предоставляется возможность интерактивной работы с удаленной машиной, например, он может распечатать содержимое ее каталогов. Наконец, FTP выполняет аутентификацию пользователей. Прежде, чем получить доступ к файлу, в соответствии с протоколом пользователи должны сообщить свое имя и пароль. Для доступа к публичным каталогам FTP-архивов Internet парольная аутентификация не требуется, и ее обходят за счет использования для такого доступа предопределенного имени пользователя Anonymous.

В стеке TCP/IP протокол FTP предлагает наиболее широкий набор услуг для работы с файлами, однако он является и самым сложным для программирования. Приложения, которым не требуются все возможности FTP, могут использовать другой, более экономичный протокол - простейший протокол пересылки файлов TFTP (Trivial File Transfer Protocol). Этот протокол реализует только передачу файлов, причем в качестве транспорта

используется более простой, чем TCP, протокол без установления соединения - UDP.

Протокол Telnet

Он обеспечивает передачу потока байтов между процессами, а также между процессом и терминалом. Наиболее часто этот протокол используется для эмуляции терминала удаленного компьютера. При использовании сервиса telnet пользователь фактически управляет удаленным компьютером так же, как и локальный пользователь, поэтому такой вид доступа требует хорошей защиты. Поэтому серверы telnet всегда используют как минимум аутентификацию по паролю, а иногда и более мощные средства защиты, например систему Kerberos. Кроме того, администратору трудно контролировать потребление ресурсов компьютера, находящегося под удаленным управлением.

При нажатии клавиши соответствующий код перехватывается клиентом telnet, помещается в TCP-сообщение и отправляется через сеть узлу, которым пользователь хочет управлять. При поступлении на узел назначения код нажатой клавиши извлекается из TCP-сообщения сервером telnet и передается операционной системе (ОС) узла. ОС рассматривает сеанс telnet как один из сеансов локального пользователя. Если ОС реагирует на нажатие клавиши выводом очередного символа на экран, то для сеанса удаленного пользователя этот символ также упаковывается в TCP-сообщение и по сети отправляется удаленному узлу. Клиент telnet извлекает символ и

отображает его в окне своего терминала, эмулируя терминал удаленного узла.

Протокол SNMP

Simple Network Management Protocol используется для организации сетевого управления. Изначально протокол SNMP был разработан для удаленного контроля и управления маршрутизаторами Internet, которые традиционно часто называют также шлюзами. С ростом популярности протокол SNMP стали применять и для управления любым коммуникационным оборудованием - концентраторами, мостами, сетевыми адаптерами и т.д. и т.п. Проблема управления в протоколе SNMP разделяется на две задачи:

Первая задача связана с передачей информации. Протоколы передачи управляющей информации определяют процедуру взаимодействия SNMP-агента, работающего в управляемом оборудовании, и SNMP-монитора, работающего на компьютере администратора, который часто называют также

консолью управления. Протоколы передачи определяют форматы сообщений, которыми обмениваются агенты и монитор.

Вторая задача связана с контролируемыми переменными, характеризующими состояние управляемого устройства. Стандарты регламентируют, какие данные должны сохраняться и накапливаться в устройствах, имена этих данных и синтаксис этих имен. В стандарте SNMP определена спецификация информационной базы данных управления сетью. Эта спецификация, известная как база данных MIB (Management Information Base), определяет те элементы данных, которые управляемое устройство должно сохранять, и допустимые операции над ними.

Протокол 9P

9P (или протокол файловой системы Plan 9 или Styx) — сетевой протокол, разработанный для распределённой операционной системы Plan 9 для организации соединения компонентов операционной системы Plan 9. Ключевыми объектами системы Plan 9 являются файлы — ими представлены окна, сетевые соединения, процессы, и почти всё, что доступно в операционной системе Plan 9. В отличие от NFS, 9P поддерживает кэширование и обслуживание синтетических файлов (например, /proc для представления процессов).

Исправленная версия 9P для 4 редакции Plan 9, которая была значительно улучшена, получила имя 9P2000. В последней версии операционной системы Inferno также используется 9P2000, который носит название Styx, но технически он всегда являлся вариантом реализации 9P.

Другая версия 9P, 9p2000.u, была переработана для лучшей поддержки окружения Unix. Серверная реализация 9P для Unix, u9fs, включена в дистрибутив Plan 9. Драйвер клиента для Linux является частью проекта v9fs. Протокол 9P и его производные реализации находят применение во встраиваемых системах, как, к примеру, Styx в проекте Brick.

Протокол BitTorrent

BitTórrent (букв. англ. «битовый поток») — пиринговый (P2P) сетевой протокол для кооперативного обмена файлами через Интернет.

Файлы передаются частями, каждый torrent-клиент, получая (скачивая) эти части, в то же время отдаёт (закачивает) их другим клиентам, что снижает нагрузку и зависимость от каждого клиента-источника и обеспечивает избыточность данных.

Протокол был создан Брэмом Коэном, написавшим первый torrent-клиент «BitTorrent» на языке Python 4 апреля 2001 года. Запуск первой версии состоялся 2 июля 2001 года.

Существует множество других программ-клиентов для обмена файлами по протоколу BitTorrent.

Протокол BOOTP

BOOTP (от англ. bootstrap protocol) — сетевой протокол, используемый для автоматического получения клиентом IP-адреса. Это обычно происходит во время загрузки компьютера. BOOTP определён в RFC 951.

BOOTP позволяет бездисковым рабочим станциям получать IP-адрес прежде, чем будет загружена полноценная операционная система. Исторически это использовалось для Unix-подобных бездисковых станций, которые в том числе могли получать информацию о местоположении загрузочного диска посредством этого протокола. А также большими корпорациями для установки предварительно настроенного программного обеспечения (например, операционной системы) на новоприобретённые компьютеры.

Изначально предполагалось использование дискет для установки предварительного сетевого соединения, но позже поддержка протокола появилась в BIOS некоторых сетевых карт и во многих современных материнских платах.

DHCP (Dynamic Host Configuration Protocol) — протокол, основанный на BOOTP, предоставляющий некоторые дополнительные возможности и являющийся более сложным. Многие DHCP-серверы поддерживают и BOOTP.

Инкапсуляция происходит следующим образом: BOOTP-> UDP-> IP-> …

Протокол DNS

DNS (англ. Domain Name System — система доменных имён) — компьютерная распределённая система для получения информации о доменах. Чаще всего используется для получения IP-адреса по имени хоста (компьютера или устройства), получения информации о маршрутизации почты, обслуживающих узлах для протоколов в домене (SRV-запись).

Распределённая база данных DNS поддерживается с помощью иерархии DNS-серверов, взаимодействующих по определённому протоколу.

Основой DNS является представление об иерархической структуре доменного имени и зонах. Каждый сервер, отвечающий за имя, может делегировать ответственность за дальнейшую часть домена другому серверу (с административной точки зрения — другой организации или человеку), что позволяет возложить ответственность за актуальность информации на серверы различных организаций (людей), отвечающих только за «свою» часть доменного имени.

Начиная с 2010 года, в систему DNS внедряются средства проверки целостности передаваемых данных, называемые DNS Security Extensions (DNSSEC). Передаваемые данные не шифруются, но их достоверность проверяется криптографическими способами. Внедряемый стандарт DANE обеспечивает передачу средствами DNS достоверной криптографической информации (сертификатов), используемых для установления безопасных и защищённых соединений транспортного и прикладного уровней.

Протокол HTTP

HTTP (англ. HyperText Transfer Protocol — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных (изначально — в виде гипертекстовых документов). Основой HTTP является технология «клиент-сервер», то есть предполагается существование потребителей (клиентов), которые инициируют соединение и посылают запрос, и поставщиков (серверов), которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом.

HTTP в настоящее время повсеместно используется во Всемирной паутине для получения информации с веб-сайтов. В 2006 году в Северной Америке доля HTTP-трафика превысила долю P2P-сетей и составила 46 %, из которых почти половина — это передача потокового видео и звука.

HTTP используется также в качестве «транспорта» для других протоколов прикладного уровня, таких как SOAP, XML-RPC, WebDAV.

Основным объектом манипуляции в HTTP является ресурс, на который указывает URI (англ. Uniform Resource Identifier) в запросе клиента. Обычно такими ресурсами являются хранящиеся на сервере файлы, но ими могут быть логические объекты или что-то абстрактное. Особенностью протокола HTTP является возможность указать в запросе и ответе способ представления одного и того же ресурса по различным параметрам: формату, кодировке, языку и т. д. (В частности, для этого используется HTTP-заголовок.) Именно благодаря возможности указания способа кодирования сообщения клиент и сервер могут обмениваться двоичными данными, хотя данный протокол является текстовым.

HTTP — протокол прикладного уровня, аналогичными ему являются FTP и SMTP. Обмен сообщениями идёт по обыкновенной схеме «запрос-ответ». Для идентификации ресурсов HTTP использует глобальные URI. В отличие от многих других протоколов, HTTP не сохраняет своего состояния. Это означает отсутствие сохранения промежуточного состояния между парами «запрос-ответ». Компоненты, использующие HTTP, могут самостоятельно

осуществлять сохранение информации о состоянии, связанной с последними запросами и ответами (например, «куки» на стороне клиента, «сессии» на стороне сервера). Браузер, посылающий запросы, может отслеживать задержки ответов. Сервер может хранить IP-адреса и заголовки запросов последних клиентов. Однако сам протокол не осведомлён о предыдущих запросах и ответах, в нём не предусмотрена внутренняя поддержка состояния, к нему не предъявляются такие требования.

Протокол NFS

Network File System (NFS) — протокол сетевого доступа к файловым системам, первоначально разработан Sun Microsystems в 1984 году. Основан на протоколе вызова удалённых процедур (ONC RPC, Open Network Computing Remote Procedure Call, RFC 1057, RFC 1831). Позволяет подключать (монтировать) удалённые файловые системы через сеть, описан в RFC 1094, RFC 1813 , RFC 3530 и RFC 5661 .

NFS абстрагирована от типов файловых систем как сервера, так и клиента, существует множество реализаций NFS-серверов и клиентов для различных операционных систем и аппаратных архитектур. В настоящее время используется наиболее зрелая версия NFS v.4 (RFC 3010 ,RFC 3530), поддерживающая различные средства аутентификации (в частности, Kerberos и LIPKEY с использованием протокола RPCSEC GSS) и списков контроля доступа (как POSIX, так и Windows-типов).

NFS предоставляет клиентам прозрачный доступ к файлам и файловой системе сервера. В отличие от FTP, протокол NFS осуществляет доступ только к тем частям файла, к которым обратился процесс, и основное достоинство его в том, что он делает этот доступ прозрачным. Это означает, что любое приложение клиента, которое может работать с локальным файлом, с таким же успехом может работать и с NFS файлом, без каких-либо модификаций самой программы.

NFS клиенты получают доступ к файлам на NFS сервере путем отправки RPC-запросов на сервер. Это может быть реализовано с использованием обычных пользовательских процессов — а именно, NFS клиент может быть пользовательским процессом, который осуществляет конкретные RPC вызовы на сервер, который так же может быть пользовательским процессом.

Важной частью последней версии стандарта NFS (v4.1) стала спецификация pNFS, нацеленная на обеспечение распараллеленной реализации общего доступа к файлам, увеличивающая скорость передачи данных пропорционально размерам и степени параллелизма системы.

Протокол POP, POP3

POP3 (англ. Post Office Protocol Version 3 — протокол почтового отделения, версия 3) — стандартный Интернет-протокол прикладного уровня, используемый клиентами электронной почты для извлечения электронного сообщения с удаленного сервера по TCP/IP-соединению.

POP и IMAP (Internet Message Access Protocol) — наиболее распространенные Интернет-протоколы для извлечения почты. Практически все современные клиенты и сервера электронной почты поддерживают оба стандарта. Протокол POP был разработан в нескольких версиях, нынешним стандартом является третья версия (POP3). Большинство поставщиков услуг электронной почты (такие как Hotmail, Gmail и Yahoo! Mail) также поддерживают IMAP и POP3. Предыдущие версии протокола (POP, POP2) устарели.

Альтернативным протоколом для сбора сообщений с почтового сервера является IMAP.

Протокол SMPT

SMTP (англ. Simple Mail Transfer Protocol — простой протокол передачи почты) — это широко используемый сетевой протокол, предназначенный для передачи электронной почты в сетях TCP/IP.

SMTP впервые был описан в RFC 821 (1982 год); последнее обновление в RFC 5321 (2008) включает масштабируемое расширение — ESMTP (англ. Extended SMTP). В настоящее время под «протоколом SMTP», как правило, подразумевают и его расширения. Протокол SMTP предназначен для передачи исходящей почты с использованием порта TCP 25.

В то время, как электронные почтовые серверы и другие агенты пересылки сообщений используют SMTP для отправки и получения почтовых сообщений, работающие на пользовательском уровне клиентские почтовые приложения обычно используют SMTP только для отправки сообщений на почтовый сервер для ретрансляции. Для получения сообщений клиентские приложения обычно используют либо POP (англ. Post Office Protocol — протокол почтового отделения), либо IMAP (англ. Internet Message Access Protocol), либо патентованные системы (такие как Microsoft Exchange и Lotus Notes/Domino) для доступа к учетной записи своего почтового ящика на сервере.

Протокол X.400

X.400 — протокол, представляет собой набор рекомендаций по построению системы передачи электронных сообщений, не зависящей от используемых на сервере и клиенте операционных систем и аппаратных

средств. Рекомендации X.400 являются результатом деятельности международного комитета по средствам телекоммуникаций (CCITT во французской транскрипции или ITU в английской), созданного при Организации Объединённых Наций.

Рекомендации X.400 охватывают все аспекты построения среды управления сообщениями: терминологию, компоненты и схемы их взаимодействия, протоколы управления и передачи, форматы сообщений и правила их преобразования. В рекомендациях X.400 наиболее полно отражается накопленный в индустрии компьютеров и телекоммуникаций опыт создания и применения информационных систем. В настоящее время существуют три редакции рекомендаций:

рекомендации 1984 года, известные также как «Красная книга» (Red Book);

рекомендации 1988 года, известные также как «Голубая книга» (Blue Book);

рекомендации 1992 года, известные также как «Белая книга» (White Book).

Более поздние рекомендации описывают дополнительные протоколы и форматы передачи данных, корректируют неточности и/или изменяют трактовку более ранних. Исправления и дополнения к указанным спецификациям выпускаются ежегодно, однако существующие системы в подавляющем большинстве поддерживают рекомендации 1984 и/или 1988 годов. Эти спецификации не являются свободно доступными и распространяются за довольно высокую плату.

Рекомендации X.400 опираются на семиуровневую модель и семейство протоколов OSI международной организации по стандартам (ISO). Согласно этой модели, каждый из уровней использует сервисы только находящегося непосредственно под ним и предоставляет сервисы только находящемуся непосредственно над ним уровню. Это обеспечивает системам, построенным на основе такой модели, высокую степень независимости от среды передачи данных. Поскольку рекомендации X.400 определяют набор спецификаций для самого верхнего уровня (Application), отвечающие этим рекомендациям приложения должны свободно взаимодействовать друг с другом, вне зависимости от применяемых операционных систем, аппаратуры и сетевых протоколов.

Для разделения входящего потока данных между приложениями на каждом из уровней, транспортом (Transport), сеанса (Session) и представлений (Presentation), используется механизм так называемых точек доступа (access

point). Каждая точка доступа имеет уникальный идентификатор, или селектор (selector), который может быть либо символьной строкой, либо последовательностью шестнадцатеричных цифр. Длина селектора транспортного уровня — 32 символа (64 цифры), уровня сеансов — 16 символов (32 цифры) и уровня представлений — 8 символов (16 цифр). Чтобы два приложения в сети могли взаимодействовать, каждое из них должно знать набор селекторов другого.

Протокол X.400 используется в тех случаях, когда требуется высокая надёжность, например, в банковских информационных системах. Из-за высокой сложности стандартов практические реализации X.400 весьма дорогостоящи и не получили большого распространения.

Устаревшие версии Microsoft Exchange Server поддерживали X.400 и использовали его в качестве своего проприетарного внутреннего формата. Позднее поддержка X.400 была удалена из продукта.

Протокол X.500

X.500 — серия стандартов ITU-T (1993 г.) для службы распределенного каталога сети. Каталоги X.500 предоставляют централизованную информацию обо всех именованных объектах сети (ресурсах, приложениях и пользователях) (рекомендации МККТТ для каталогов). Изначально стандарт X.500 планировался для использования именований узлов, адресов и почтовых ящиков, предусмотренных стандартом X.400.

Каталоги, как правило, содержат статические и редко изменяемые элементы, так как каталоги изначально оптимизированы для очень быстрого отклика на запросы поиска и чтения данных.

Каталоги полностью структурированы. Каждый элемент данных имеет имя, которое, одновременно определяет положение элемента в иерархии каталога. Каждый атрибут элемента, как правило, может иметь несколько значений и это является нормальным поведением, в отличие от обычных баз данных.

Каталоги являются очень специфическими системами хранения данных. Их удобно использовать для иерархически скомпонованных объектов. Каталоги могут быть реплицированы между несколькими серверами, для удобного доступа и распределения нагрузки. Текстовая информация очень хорошо подходит для каталогов, так как легко поддается поиску, но данные могут быть представлены и в любой другой форме.

Очень удобно использовать каталоги для управления пользовательскими аккаунтами, машинами, схемами доступа, приложениями

и многим другим, поскольку механизмы управления чаще всего только считывают данные из центрального храни

Протокол SPDY

SPDY (читается как «speedy», «спиди») — протокол прикладного уровня для передачи веб-контента. Протокол разработан корпорацией Google. По замыслу разработчиков, данный протокол позиционируется как замена некоторых частей протокола HTTP — таких, как управление соединениями и форматы передачи данных.

Основной задачей SPDY является снижение времени загрузки веб-страниц и их элементов. Это достигается за счет расстановки приоритетов и мультиплексирования передачи нескольких файлов таким образом, чтобы требовалось только одно соединение для каждого клиента.

Документация по проекту уже доступна, было проведено первое лабораторное тестирование. Тесты проходили таким образом: создатели сымитировали сеть и загрузили по SPDY-протоколу 25 крупнейших мировых сайтов. Статистика говорит о том, что в ряде случаев веб-страницы загружались на 55% быстрее, чем при использовании HTTP-протокола. В документации также сказано, что время загрузки страниц стало меньше на 36%.

10. Траблшутинг

Траблшутинг (англ. troubleshooting — устранение неполадок, работа над проблемой) — форма решения проблем, часто применяемая к ремонту неработающих устройств или процессов. Представляет собой систематический, опосредованный определённой логикой поиск источника проблемы с целью её решения. Траблшутинг как поиск и устранение неисправностей необходим для поддержания и развития сложных систем, где проблема может иметь множество различных причин.

Как правило, траблшутингом занимается техническая поддержка.

Есть разные причины, по которым все идет не так в наших сетях: люди делают ошибки в своих настройках, оборудование может выйти из строя, обновления программного обеспечения могут включать ошибки, а изменение структуры трафика может вызвать перегрузку в наших сетях. Для устранения этих ошибок существуют различные подходы, и некоторые из них более эффективны, чем другие.

Устранение неполадок состоит из 3 этапов:



Все это начинается, когда кто-то или что-то сообщает о проблеме. Часто это будет пользователь, который звонит в службу поддержки, потому что что-то работает не так, как ожидалось, но также возможно, что вы обнаружите проблемы из-за мониторинга сети (Вы ведь контролируете свою сеть?). Следующий шаг — это диагностика проблемы, и очень важно найти ее корень. Как только вы обнаружите проблему, вы реализуете (временное) решение.

Диагностика проблемы является одним из самых важных шагов, чтобы устранить неполадки в сети. Для начала нам нужно найти первопричину проблемы. И для этого, необходимо выполнить ряд действий:

Сбор информации: в большинстве случаев отчет о проблеме не дает нам достаточно информации. Пользователи просто нам сообщают, что "сеть не работает" или "Мой компьютер не работает", но это нам ничего не дает. Мы должны собирать информацию, задавая нашим пользователям подробные вопросы, или мы используем сетевые инструменты для сбора информации.

Анализ информации: как только мы собрали всю информацию, мы проанализируем ее, чтобы увидеть, что не так. Мы можем сравнить нашу информацию с ранее собранной информацией или другими устройствами с аналогичными конфигурациями.

Устранение возможных причин: нам нужно подумать о возможных причинах и устранить потенциальные причины проблемы. Это требует досконального знания сети и всех протоколов, которые в ней задействованы.

Гипотеза: после определения возможных причин, вы в итоге получите список этих причин, которые могут вызывать проблему работу сети. Мы выберем самую наиболее вероятную причину возникновения проблемы.

Проверка гипотезы: мы проверим нашу гипотезу, чтобы увидеть, правы мы или нет. Если мы правы, у нас есть победа...если мы ошибаемся, мы проверяем наши другие возможные причины.

Если вы применяете структурированный подход для устранения неполадок, вы можете просто "**следовать интуиции**" и запутаться, потому что вы забыли, что вы уже пробовали или нет. Это упрощает поиск проблемы, если вы работаете вместе с другими сетевыми администраторами, потому что вы можете поделиться шагами, которые вы уже выполнили.



Вот шаги поиска проблемы в блок-схеме. Мы называем это структурированным подходом к устранению неполадок.

Вместо того чтобы выполнять все различные этапы структурированного подхода к устранению неполадок, мы также можем перейти от этапа "сбор информации" непосредственно к шагу "гипотеза" и пропустить этапы "**анализ информации**" и "**устранение возможных причин**". По мере того, как вы наберётесь опыта в устранении неполадок, вы сможете пропустить некоторые шаги.



Траблшутинг сетевых неполадок

Шаги, которые мы пропускаем, выделены синим цветом. Если вас ваши интуиция подведет, то вы потеряете много времени. Если вы правы, то вы сэкономите много времени.

Устранение возможных причин является важным шагом в процессе устранения неполадок, и есть несколько подходов, как вы можете это сделать.

Перечислим эти подходы:

- Сверху вниз;
- Снизу вверх;
- Разделяй и властвуй;
- Отследить путь трафика;
- Поиск отличий;

- Замена компонентов.

Давайте пройдемся по разным подходам один за другим!

Модель OSI

Метод "сверху вниз"

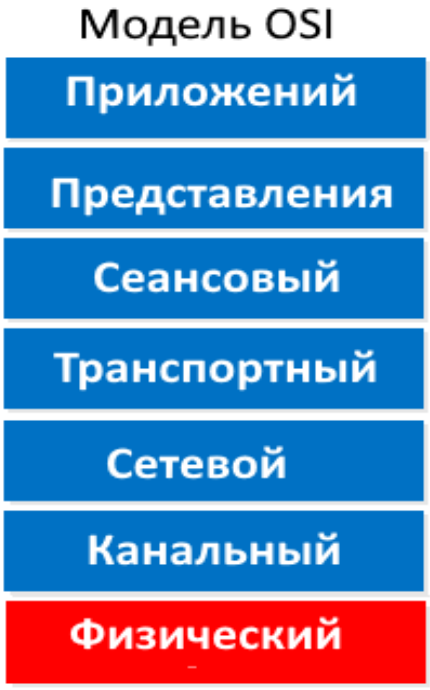
"Сверху вниз" означает, что мы начинаем с верхней части модели OSI (прикладной уровень) и продвигаемся дальше вниз. Идея заключается в том, что мы проверим приложение, чтобы увидеть, работает ли оно, и предположим, что если определенный уровень работает, то все нижеперечисленные уровни также работают. Если вы посылаете эхо-запрос с одного компьютера на другой (ICMP), то можете считать, что уровни 1,2 и 3 работают. Недостатком этого подхода является то, что вам нужен доступ к приложению, в котором устраняете неполадки.



Метод "снизу вверх"

"Снизу вверх" означает, что мы начинаем с нижней части модели OSI и будем продвигаться вверх. Мы начнем с физического уровня, который означает, что мы проверяем наши кабели и разъемы, переходим к канальному уровню, чтобы увидеть, работает ли Ethernet, связующее дерево работает нормально, безопасность портов не вызывает проблем, VLAN настроены правильно, а затем переходим на сетевой уровень. Здесь мы будем проверять наши IP-адреса, списки доступа, протоколы маршрутизации и так далее. Этот подход является очень тщательным, но и отнимает много времени. Если вы новичок в устранении неполадок рекомендуется

использовать этот метод, потому что вы устраните все возможные причины проблем.



Метод "разделяй и властвуй"

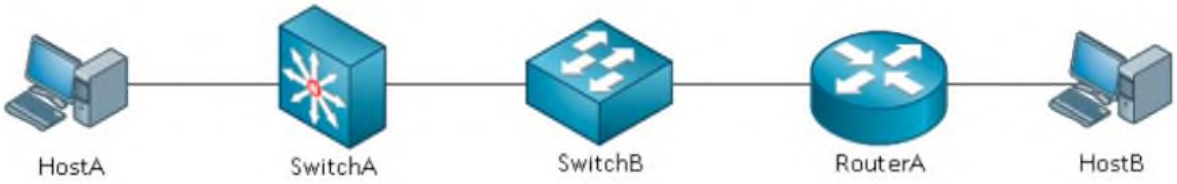
Разделяй и властвуй означает, что мы начинаем с середины OSI-модели. Вы можете использовать эту модель, если не уверены, что нисходящее или восходящее движение более эффективно. Идея заключается в том, что вы попытаетесь отправить эхо-запрос с одного устройства на другое. Если ping работает, вы знаете, что уровень 1–3 работает, и вы можете продвинуться вверх по модели OSI.

Если эхо-запрос терпит неудачу, то вы знаете, что что-то не так, и вы будете причину проблемы в нижней части модели OSI.



Метод "путь трафика"

Изучение путь следования трафика очень полезно. Сначала мы попытаемся отправить эхо-запрос с хоста A на хост B. В случае сбоя мы проверим все устройства на его пути. Сначала мы проверим, правильно ли настроен коммутатор A, и, далее, мы перейдем на коммутатор B, проверим его, а затем перейдем к маршрутизатору A.



Метод "поиск отличий"

Этот подход вы, скорее всего, делали и раньше. Поиск отличий в конфигурации или вывод команд show может быть полезным, но очень легко что-то пропустить. Если у вас есть несколько маршрутизаторов филиала с похожей конфигурацией, и только один не работает, вы можете заметить отличие в конфигурациях. Сетевые администраторы, которые не имеют большого опыта, обычно используют этот подход. Возможно, вам удастся решить проблему, но есть риск, что вы на самом деле не знаете, что делаете.



Метод "замена компонентов"

Последний подход к решению нашей проблемы — это замена компонентов. Допустим, у нас есть сценарий, в котором компьютер не может получить доступ к сети. В приведенном выше примере мы можем заменить компьютер, чтобы устранить любую вероятность того, что компьютер является проблемой. Мы можем заменить кабель, и, если мы подозреваем, что коммутатор не работает или неверно настроен, мы можем заменить его на новый и скопировать старую конфигурацию, чтобы увидеть, есть ли какие-либо проблемы с оборудованием.

11. DHCP, PXE

DHCP — протокол автоматизации назначения IP-адреса клиенту. Он широко используется в современных сетях. Рассмотрим принципы работы, процесс DORA, основные опции и другие аспекты протокола.

Для чего нужен протокол DHCP

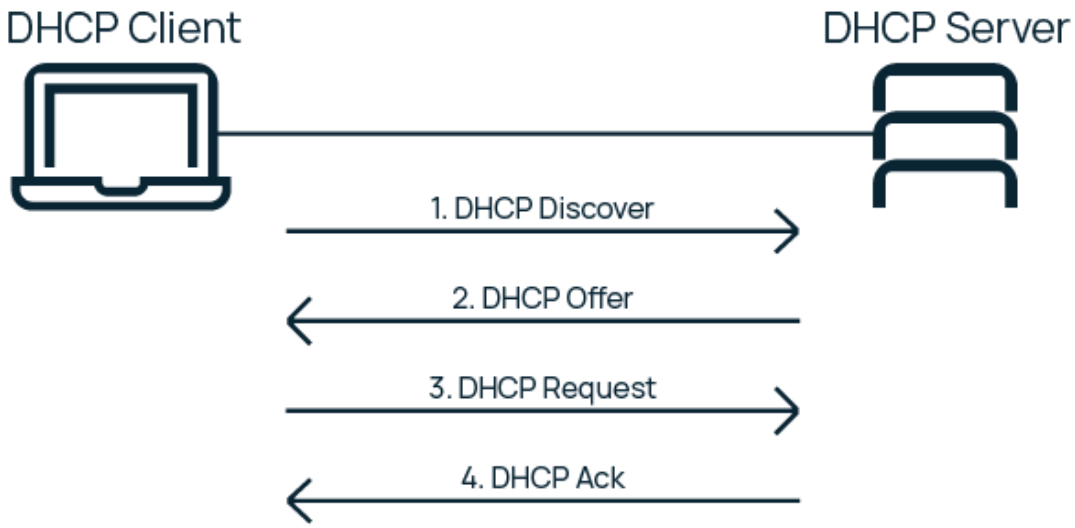
DHCP — протокол прикладного уровня модели TCP/IP, служит для назначения IP-адреса клиенту. Это следует из его названия — Dynamic Host Configuration Protocol. IP-адрес можно назначать вручную каждому клиенту, то есть компьютеру в локальной сети. Но в больших сетях это очень трудозатратно, к тому же, чем больше локальная сеть, тем выше возрастает вероятность ошибки при настройке. Поэтому для автоматизации назначения IP был создан протокол DHCP.

Впервые протокол был описан в 1993 году в документе RFC 1531, но с тех пор в описание вносились правки. На сегодняшний день основным документом, регламентирующим протокол, является RFC 2131. Помимо автоматизации процесса настройки IP, DHCP позволяет упростить диагностику подключения и переход из одной подсети в другую, оставляя уведомления для системного администратора в логах.

Принцип работы DHCP

Из вступления ясно, какие функции предоставляет DHCP, но по какому принципу он работает? Получение адреса проходит в четыре шага. Этот процесс называют DORA по первым буквам каждого шага: Discovery, Offer, Request, Acknowledgement.

Давайте подробнее рассмотрим DORA — принцип работы DHCP.



Протокол DHCP, получение адреса IP — DORA

Discovery, или поиск

Изначально клиент находится в состоянии инициализации (INIT) и не имеет своего IP-адреса. Поэтому он отправляет широковещательное (broadcast) сообщение DHCPDISCOVER на все устройства в локальной сети. В той же локальной сети находится DHCP-сервер. DHCP-сервер — это, например, маршрутизатор или коммутатор, существуют также выделенные DHCP-серверы.

Не всегда одну сеть обслуживает один DHCP-сервер, нередко организации устанавливают сразу несколько. Какие порты использует DHCP? Сервер всегда слушает 67 порт, ожидает широковещательное сообщение от клиента, а после его получения отправляет ответное предложение — DHCPOFFER. Клиент принимает сообщение на 68 порту.

Offer, или предложение

DHCP-сервер отвечает на поиск предложением, он сообщает IP, который может подойти клиенту. IP выделяются из области (SCOPE) доступных адресов, которая задается администратором.

Если имеются адреса, которые не должны быть назначены DHCP-сервером, область можно ограничить, указав только разрешенные адреса. Например, администратор может задать диапазон используемых IP-адресов от 192.0.0.10 до 192.0.0.255.

Бывает и так, что не все доступные адреса должны быть назначены клиентам. Например, администратор может исключить (exclude) диапазон 192.0.0.100–192.0.0.200 из используемой области. Такое ограничение называется исключением.

DHCP выделяет доступные IP-адреса из области только временно (об этом позже), поэтому нет гарантии, что при следующем подключении у данного клиента останется прежний IP. Но есть возможность назначить какому-либо клиенту определенный IP навсегда. К примеру, забронировать 192.0.0.10 за компьютером системного администратора. Такое сохранение IP для отдельных клиентов называют резервацией (reservation).

DHCPOFFER содержит IP из доступной области, который предлагается клиенту отправкой широковещательного (broadcast, «если вы тот, кто запрашивал IP-адрес, то доступен вот такой») или прямого (unicast, «вы запрашивали IP, предлагаю вот такой») сообщения. При этом, поскольку нужный клиент пока не имеет IP, для отправки прямого сообщения он идентифицируется по MAC-адресу.

Request, или запрос

Клиент получает DHCPOFFER, а затем отправляет на сервер сообщение DHCPREQUEST. Этим сообщением он принимает предлагаемый адрес и уведомляет DHCP-сервер об этом. Широковещательное сообщение почти полностью дублирует DHCPDISCOVER, но содержит в себе уникальный IP, выделенный сервером. Таким образом, клиент сообщает всем доступным DHCP-серверам «да, я беру этот адрес», а сервера помечают IP как занятый.

Acknowledgement, или подтверждение

Сервер получает от клиента DHCPREQUEST и окончательно подтверждает передачу IP-адреса клиенту сообщением DHCPACK. Это широковещательное или прямое сообщение утверждает не только владельца IP, но и срок, в течение которого клиент может использовать этот адрес.

Со схемой отправки сообщений разобрались, но, если в сети несколько DHCP-серверов, пославших предложение, какое из них выберет клиент? Хороший вопрос. В состоянии INIT, если клиент получает адрес впервые, он будет принимать только первое предложение IP. Однако, если клиент уже общался ранее с определенным DHCP-сервером, он отдаст предпочтение этому серверу и, наоборот, сервер выберет знакомого клиента.

Срок аренды

Когда DHCP-сервер выделяет IP из области, он оставляет запись о том, что этот адрес зарезервирован за клиентом с указанием срока действия IP. Этот срок действия называется срок аренды (lease time). Срок аренды может составлять от 24 часов до нескольких дней, недель или даже месяцев, он задается в настройках самого сервера.

Предоставление адреса в аренду, а не на постоянной основе необходимо по нескольким причинам. Во-первых, это разумное использование IP-адресов — отключенные или вышедшие из строя клиенты не резервируют за собой адрес. Во-вторых, это гарантия того, что новые клиенты при необходимости смогут получить уникальный адрес.

После получения адреса из области, клиент берет его в аренду на время, называемое T. Клиент переходит в связанное (BOUND) состояние и продолжает нормальную работу, пока не наступит время половины срока аренды — T1.

По наступлении T1 клиент инициализирует процедуру получения нового IP или обновления адреса — состояние RENEWING. Процесс повторного получения происходит по упрощенной схеме: клиент прямым сообщением запрашивает (DHCPREQUEST), а сервер подтверждает (DHCPACK) запрос. Время аренды начинает отсчитываться заново.

Если подтверждение (DHCPACK) от сервера не поступает, клиент снова запрашивает адрес, но только когда истекает половина T1. Если запрос адреса остается без ответа второй раз, клиент отправляет еще одно сообщение, когда истекает половина от T1/2 (25% от полного срока аренды). Следующий запрос будет отправлен после истечения еще половины оставшегося времени, потом еще половины. И так далее, пока не наступит T2, которое равняется 87,5%, или 7/8 от всего времени аренды. После T2 все попытки продлить аренду IP будут широковещательными. Это значит, что, если первый сервер по какой-то причине недоступен, на запрос адреса сможет ответить любой другой, и работа не будет прервана.

Три подхода к распределению адресов

Сервер назначает IP одним из трех основных способов.

Статическое распределение (static allocation). Почти как ввод адреса на каждом компьютере вручную. Отличие в том, что системный администратор задает нужные соответствия IP для MAC-адресов клиентов на самом DHCP-сервере. IP останется за клиентом, даже если тот выйдет из сети, отключится, перейдет в новую сеть и т. п.

Автоматическое распределение (automatic allocation). Сервер закрепляет IP из области за каждым клиентом навсегда. Срок аренды не ограничен.

Динамическое распределение (dynamic allocation). DHCP-сервер назначает адрес из области на определенное время, называемое сроком аренды. Такой подход полезен, если число доступных IP ограничено. IP назначается каждому клиенту при подключении к сети и возвращается в область, как только клиент его освобождает. В таком случае IP может отличаться при каждом подключении, но обычно назначается прежний.

Особые DHCP сообщения

Кроме DORA — четырех сообщений для получения адреса — DHCP использует и другие. Давайте рассмотрим каждое.

DHCPNAK. Нередко в источниках можно встретить написание DHCPNACK, что является неправильным, так как RFC 2131 регламентирует именно NAK. DHCPNAK отправляется сервером вместо окончательного подтверждения. Такой отказ может быть отправлен клиенту, если аренда запрашиваемого IP истекла или клиент перешел в новую подсеть.

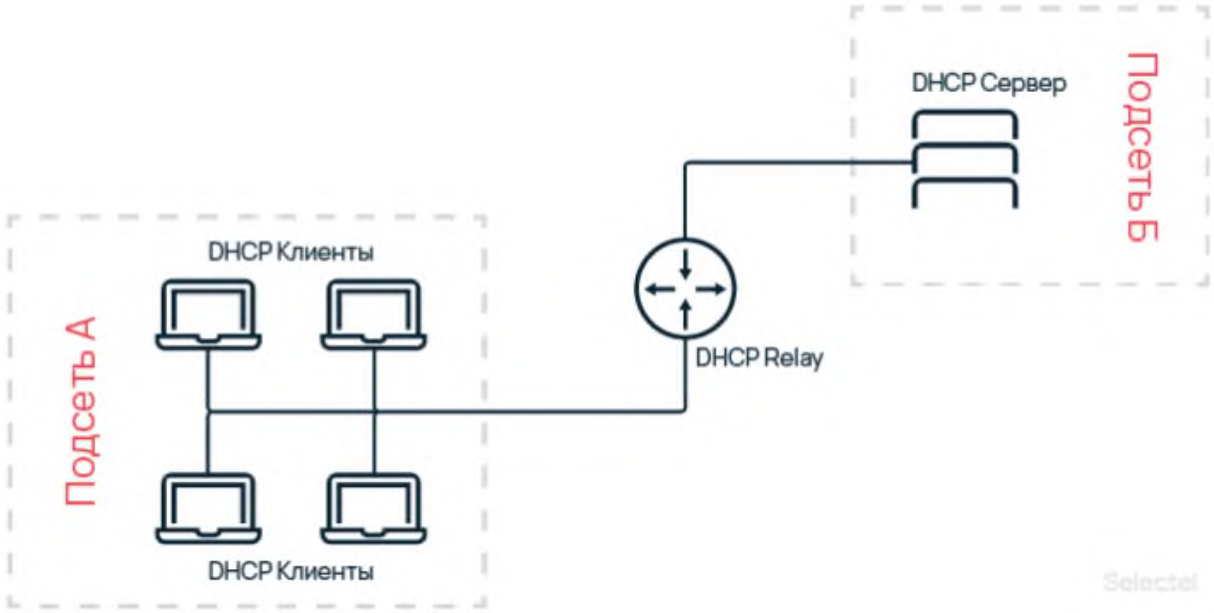
DHCPRELEASE. Клиент отправляет это сообщение, чтобы уведомить сервер об освобождении занимаемого IP. Иными словами, это досрочное окончание аренды.

DHCPINFORM. Этим сообщением клиент запрашивает у сервера локальные настройки. Отправляется, когда клиент уже получил IP, но для правильной работы ему требуется конфигурация сети. Сервер информирует клиента ответным сообщением с указанием всех запрошенных опций.

Опции DHCP

Для работы в сети клиенту требуется не только IP, но и другие параметры DHCP — например, маска подсети, шлюз по умолчанию и адрес сервера. Опции представляют собой пронумерованные пункты, строки данных, которые содержат необходимые клиенту сервера параметры конфигурации. Дадим описания некоторым опциям:

- Option 1 — маска подсети IP;
- Option 3 — основной шлюз;
- Option 6 — адрес сервера DNS (основной и резервный);
- Option 51 определяет, на какой срок IP-адрес предоставляется в аренду клиенту;
- Option 55 — список запрашиваемых опций. Клиент всегда запрашивает опции для правильной конфигурации. Отправляя сообщение с Option 55, клиент выставляет список запрашиваемых числовых кодов опций в порядке предпочтения. DHCP-сервер старается отправить ответ с опциями в том же порядке.



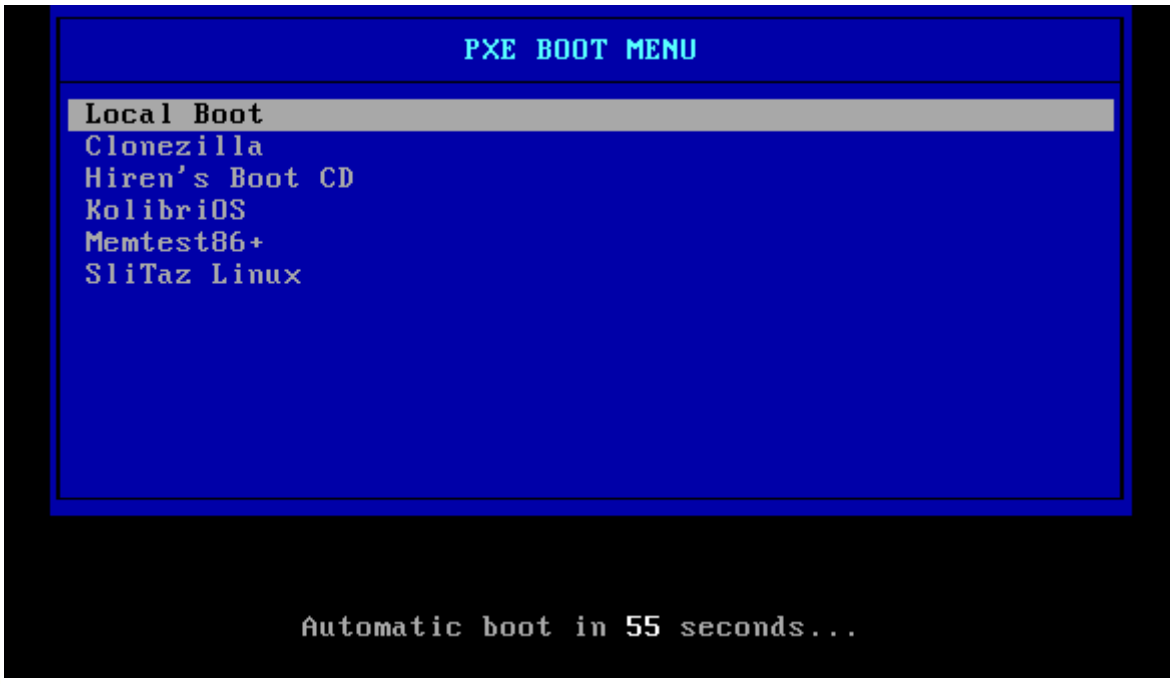
Option 82 — ретрансляция DHCP-сервера

Option 82 — информация об агенте ретрансляции (relay agent information). Благодаря ретранслятору клиент и сервер могут общаться, находясь в разных подсетях. По умолчанию широковещательные сообщения не могут выходить за пределы текущего широковещательного домена

(подсети). Внимательный читатель скажет, что выше мы писали, как клиент отправляет широковещательное сообщение DHCPDISCOVER всем доступным DHCP-серверам. Что, если в сети нет DHCP?

Предположим, широковещательные сообщения не выходят за пределы подсети компании, которая не установила DHCP-сервер. В таком случае сообщение DHCPDISCOVER должно будет пропасть, и ни один компьютер компании не сможет выйти в интернет. Однако в реальности отсутствие DHCP-сервера не мешает выходу в сеть.

Значит ли это, что широковещательные сообщения каким-то образом выходят за пределы подсети? Не совсем. За пределы подсети выходят только широковещательные DHCP-сообщения. Это становится возможным благодаря агенту ретрансляции. Обычно в его роли выступает маршрутизатор или сервер. Ретранслятор получает сообщения от клиента в своей подсети, направляют его на DHCP-сервер, который тем же образом — через ретранслятор — отправляет ответ. Так ретранслятор выступает в качестве посредника между подсетями.



Опции DHCP для загрузки PXE

PXE (англ. Preboot Execution Environment) — средство загрузки компьютеров с помощью сетевой карты без использования жёстких дисков, компакт-дисков и других устройств, применяемых при загрузке операционной системы.

Протокол DHCP позволяет загрузку компьютера без использования носителя данных. Такая загрузка происходит с сетевой карты и называется PXE (Preboot eXecution Environment). Для конфигурации сетевой загрузки LEGACY BIOS PXE используются DHCP-опции 43, 60, 66 и 67.

Option 43 зарезервирована для обмена информацией производителей;

Option 60 — классовый идентификатор; здесь указывается, например, PXE клиент;

Option 66 и 67 необходимы для указания имени сервера PXE и имени файла загрузки соответственно.

12. DNS



Любой сайт в интернете фактически находится на каком-либо устройстве. Отдельный компьютер, подключенный к интернету, имеет индивидуальный номер, который называется IP-адресом. Он представляет собой набор из четырех чисел от 0 до 255. Благодаря этому адресу можно узнать, откуда загружается страница нужного нам ресурса.

IP-адрес устройства можно сравнить с номером мобильного телефона, а DNS — с телефонной книгой. Если говорить конкретнее, DNS — система доменных имен, которая обеспечивает связь между наименованием сайта и его цифровым адресом.

Иными словами, пользователь набирает доменное имя ресурса в адресной строке браузера, а DNS конвертирует его в IP-адрес и передает вашему устройству. После чего компьютер, находящийся по этому адресу, обрабатывает запрос и присылает информацию для открытия необходимой страницы сайта.

Структуру DNS можно сравнить с логическим деревом. Система содержит распределенную базу доменных имен — пространство, которое организовано по принципу иерархии. На верхнем уровне располагается корневой домен, к которому примыкают домены первого уровня. К каждому из них присоединяются домены второго уровня и так далее.

Что такое DNS-сервер

Система доменных имен действует посредством DNS-сервера, который нужен для выполнения двух основных функций:

- хранения данных о соответствии имени домена конкретному IP-адресу,
- кэширования ресурсных записей прочих DNS-серверов.

Если пользователь собирается посетить сайт, находящийся в другой стране, то регулярная передача запросов к первичному серверу занимает много времени и приводит к медленной загрузке страниц. Чтобы избежать подобных неудобств, DNS-сервер, находящийся рядом с вашим устройством, кэширует данные о запрашиваемых ранее IP-адресах и выдает их при следующем обращении.

Источниками хранения ресурсных записей являются исходные DNS-серверы, содержащие начальные связи между доменами и сетевыми адресами узлов.

Как правило, рекомендуют задействовать два сервера: первичный и вторичный. Это гарантирует получение доступа к вашему домену, потому как, если будет недоступен один сервер, ответит другой.

Как работают DNS-серверы

Рассмотрим поэтапно функционирование приложений, предназначенных для ответа на DNS-запросы:

- Браузер получает запрос от пользователя и направляет его DNS-серверу сети, который ищет совпадение доменного имени и сетевого адреса. Если ответ обнаружен, то страница сайта загружается сразу. В противном случае запрос будет отправлен серверу более высокого уровня или корневому.
- Корневой сервер направляет запрос серверу первого уровня, который в свою очередь передает его серверу второго уровня. Это движение продолжается до тех пор, пока не будет найдено совпадение имени и IP-адреса.
- Браузер получает ответ на свой запрос, направляет его к хостингу, и страница открывается.

Также возможна обратная процедура — поиск имени домена в DNS-сервере, соответствующего запрашиваемому IP-адресу. К примеру, это происходит в случае работы с сервером электронной почты.

Где находятся DNS-серверы

Фундаментом для обработки запросов о доменных именах являются корневые серверы, отвечающие за корневую DNS-зону. Ими руководят

разные операторы, которые обеспечивают бесперебойное функционирование серверов. Первые корневые серверы появились в Северной Америке, но с течением времени они стали появляться в других странах мира. На сегодня существует 123 корневых сервера, которые располагаются в разных точках мира (в зависимости от интенсивности пользования всемирной паутиной).

Типы записей DNS-сервера

Одному домену могут подходить несколько сетевых адресов, например, интернет-сайт и почтовый сервер. Более того, каждое доменное имя содержит один или несколько поддоменов.

Все соответствия домена и его IP-адресов хранятся в файле на DNS-сервере, содержимое которого называется DNS-зона. Чтобы внести информацию в систему DNS, необходимо прописать ресурсные записи.

Различают несколько ключевых типов ресурсных записей, информация о которых хранится на DNS-сервере:

- A — адрес веб-ресурса, который соответствует введенному имени домена.
- MX — адрес почтового сервера.
- CNAME — указание привязки аналога к собственному доменному имени. Чаще всего используется для прикрепления поддомена. Например, можно привязать веб-адрес www.site.ru к фактическому сайту для домена site.ru.
- NS — адрес DNS-сервера, отвечающего за содержание прочих ресурсных записей.
- TXT — любая текстовая информация о домене.
- SPF — данные с указанием списка серверов, которым позволено отправлять письма от имени указанного домена.
- SOA — исходная запись зоны, в которой указаны сведения о сервере, содержащем образцовую информацию о доменном имени.

Зачем нужно прописывать DNS-серверы

Представьте, что вы только что зарегистрировали домен. Чтобы остальным серверам стала доступна информация о существовании вашего домена, необходимо прописать ресурсные записи. Первым делом нужно произвести настройку и прописать для домена DNS-серверы.

Серверы такого формата обновляются до 24 часов, в этот период сайт может не работать. Поэтому необходимо подождать сутки после того, как вы их прописали.

Зачастую такие серверы прописывают парами. У каждого домена существует один первичный и до 12 вторичных серверов. Это нужно для обеспечения надежности работы сайта. Если один из них полетит, то домен и ресурс будут функционировать.

Защита DNS-серверов от атак

В наши дни опасность воздействия хакеров на DNS приобрела глобальные масштабы. Ранее уже были ситуации атак на серверы такого формата, которые приводили к многочисленным сбоям в работе всемирной паутины, в особенности известных социальных сетей.

Наиболее опасными считают нападения на корневые серверы, хранящие данные об IP-адресах. Например, в историю вошла произошедшая в октябре 2002 года DDoS-атака на 10 из 13 серверов верхнего уровня.

Протокол DNS получает результаты по запросам с помощью протокола пользовательских датаграмм UDP. UDP использует модель передачи данных без соединений для обеспечения безопасности и целостности информации. Таким образом, большинство атак производятся на этот протокол с помощью подделки IP-адресов.

Существует несколько схем, настройка которых позволит защитить DNS-сервер от атак хакеров:

- **Использование технологии uRPF (Unicast Reverse Path Forwarding).**

Суть состоит в том, чтобы определить возможность принятия пакета с конкретным адресом отправителя на указанном устройстве для передачи данных. Пакет проходит проверку и принимается в том случае, когда сетевой интерфейс, с которого он получен, предназначен для обмена информацией с адресатом данного пакета. В обратной ситуации пакет будет отброшен. Этот способ помогает выявить и частично отобрать фальшивый трафик, но не гарантирует надежную защиту от фальсификации. uRPF полагает, что данные отправляются на определенный адрес через неизменный интерфейс. Ситуация усложняется, если появляется несколько провайдеров.

- **Применение функции IP Source Guard.**

В ее основе лежит технология uRPF и проверка DHCP-пакетов. IP Source Guard отслеживает DHCP-трафик в интернете и выясняет, какие IP-адреса получили сетевые устройства. Это позволяет выявить поддельный трафик на некоторых портах установки. После этого данные собираются и записываются в общую таблицу итогов проверки DHCP-пакетов. В дальнейшем IP Source Guard обращается к этой таблице, чтобы осуществить проверку пакетов,

полученных коммутатором. Если IP-адрес пакета не совпадает с адресом источника, то пакет откладывается.

- Использование утилиты dns-validator.

Эта программа контролирует передачу всех пакетов DNS, соотносит запрос с ответом и в случае расхождения названий отправляет уведомление пользователю.

Вывод

DNS-сервер хранит информацию о соответствии домена IP-адресу, а также содержит сведения об остальных ресурсных записях.

Распределенная система доменных имен была создана в начале 1980-х годов. До настоящего времени она обеспечивает взаимодействие с адресным пространством всемирной паутины. Вместе с тем технологии DNS регулярно совершенствуются. Например, в 2010 году в России был внедрен национальный кириллический домен первого уровня. рф.

Значительное внимание уделяется усилению безопасности системы и снижению чувствительности к перебоям в работе. В наши дни интернет является существенной частью жизни, поэтому стихийные бедствия, перепады напряжения в сети и отключение электроэнергии не должны влиять на его производительность.

13. HTTP/HTTPS

HTTP и HTTPS — это протоколы, которые используются веб-браузерами и серверами для передачи пакетов данных. Образно говоря, это почтовые службы, по которым передают письма из байтов. Разница между протоколами заключается в том, что HTTPS — защищенный инструмент, который обеспечивает сохранность данных при передаче. HTTP, напротив, обладает встроенной уязвимостью, которая в прошлом приводила к сбоям в работе сайтов и к крупным похищениям пользовательских данных.

Для обозначения защищенного соединения (протокол HTTPS) используется пиктограмма в виде закрытого зеленого замка.

Чем опасно использование HTTP?

Для того чтобы ответить на вопрос о целесообразности отказа от HTTP, необходимо понять, каким образом функционирует этот протокол.

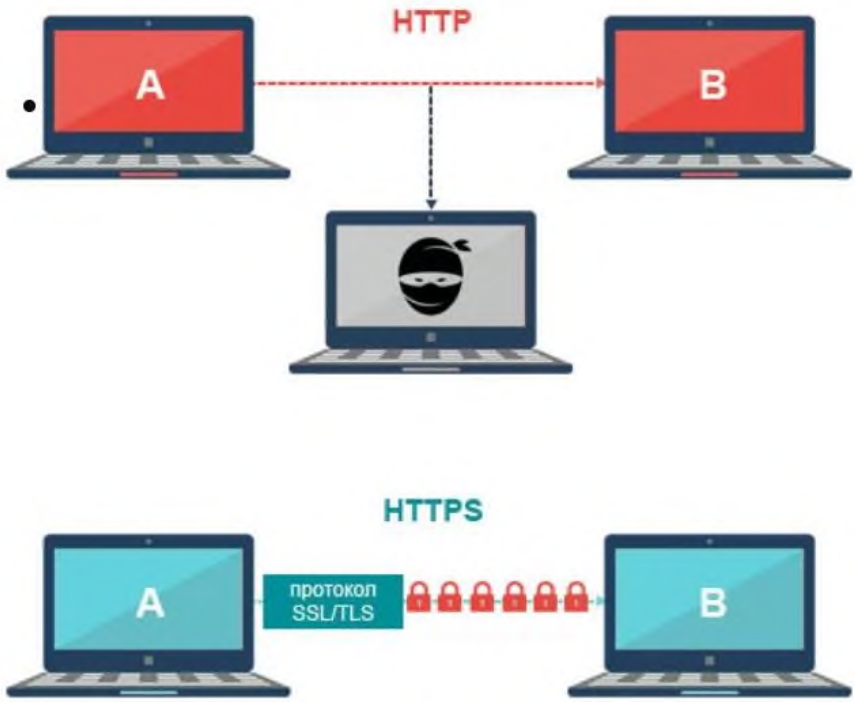
Начнем с определения. HTTP — это аббревиатура, образованная от «Hypertext Transfer Protocol» («Протокол передачи гипертекста»). Веб-браузеры и серверы, используя протокол HTTP, указывают, что они являются

частью Всемирной паутины (WWW). Проблема с HTTP заключается в том, что его изобрели в 1989 году. В то время разработчиков, как и пользователей, не очень беспокоил вопрос интернет-безопасности. Гораздо важнее было разработать унифицированный стандарт, который позволит пользователям со всего мира выходить онлайн и пользоваться нужными сайтами.

Проблемы с HTTP начались уже в 1990-х, одновременно с глобальным распространением интернета. Появилось много популярных сайтов, на некоторых ввели систему онлайн-платежей. Тогда и начали активно действовать злоумышленники. Они похищали пользовательские данные либо рушили серверы с сайтами. Киберпреступность процветала, в частности, из-за уязвимости в протоколе HTTP. Чтобы понять её суть, нам необходимо подробнее рассмотреть матчасть, и узнать, в чём отличие HTTP от HTTPS.

Принцип действия HTTP

Когда интернет-сообщество ввело в употребление протокол HTTP, два основных вопроса были — как осуществить передачу данных без потерь, и как доставить пакеты данных с максимальной скоростью? При таких критериях программисты договорились, что в интернете все данные будут отправлены и получены «открытым текстом». Представьте, что все почтовые отделения мира отправляют и принимают письма без конвертов. По умолчанию предполагается, что почтальоны достаточно надежные, чтобы пересылать корреспонденцию, не читая её.



Аналогичным образом устроен принцип действия HTTP. Если интернет-магазин использует этот протокол, то при отправке данных платежной карты по проводам в точности будет отправлен номер банковской карты.

Злоумышленнику несложно перехватывать данные на промежутке между сайтом, которым пользуется покупатель, и сервером, на котором находится интернет-магазин. Так данные пользователей попадали к мошенникам.

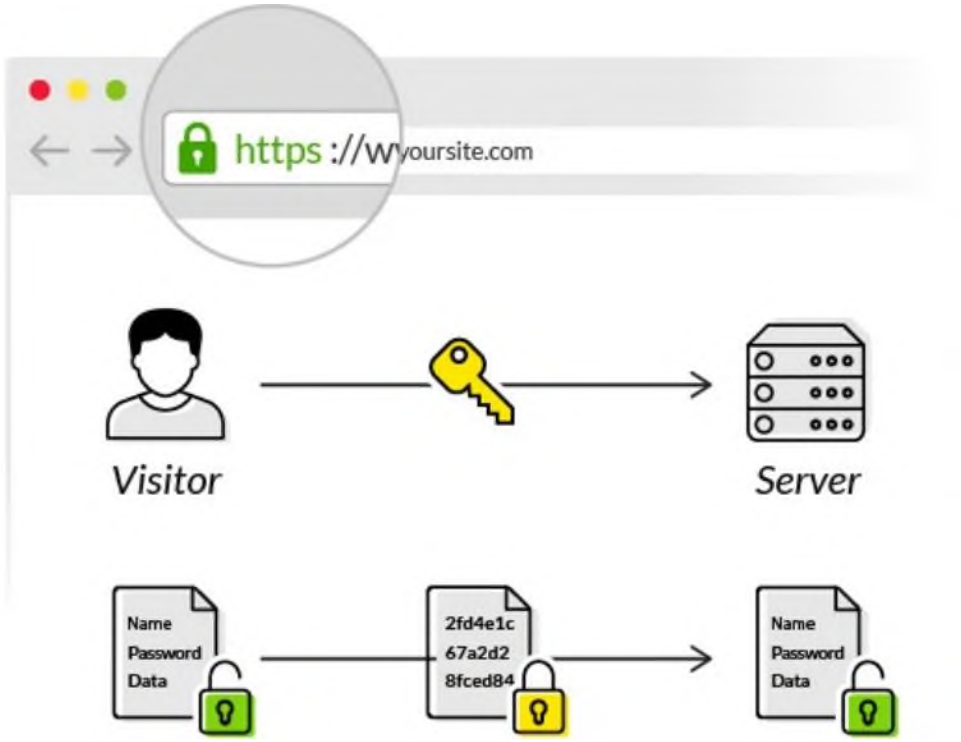
Еще хуже, что хакеры могут перехватить трафик на сайт и добавить небольшие фрагменты кода (сниппеты) к каждому пакету данных. В марте 2015 года по такой схеме хакеры реализовали DDoS-атаку на сайты GreatFire.org и GitHub. Через серверы, которые предположительно имеют связь с «Великим китайским файерволом», хакеры смогли обрушить серверы GitHub. Сайт не работал на протяжении 5 минут, пока системные администраторы подключали резервный сервер. Это крупнейшая, но не единственная атака на сайт, которая стала возможной из-за использования протокола HTTP.

Сейчас, во время повсеместного распространения высокоскоростного интернета, использовать HTTP не нужно даже домохозяйкам, которые ведут небольшой блог о кулинарии. Впрочем, возможная атака на сайт — это не единственная угроза для владельца. Об этом мы скажем ниже.

Принцип действия HTTPS и его отличие от HTTP

Буква «s» в названии протокола HTTPS означает «secure», т. е. защищенный. Веб-браузеры и сайты, которые используют протокол HTTPS, отправляют данные, защищенные криптошифрованием. То есть на пути от компьютера пользователя до сервера веб-приложения информация курсирует в нечитабельном виде. Это рандомный набор знаков. Основное достоинство протокола заключается в том, что дешифровка данных происходит в рамках одной веб-сессии. То есть невозможно подобрать универсальный дешифровщик, который позволит приводить любые данные в удобочитаемый вид.

Рассмотрим подробнее, как функционирует протокол HTTPS. Для передачи данных используется подслой — криптографический протокол, который обеспечивает шифровку/дешифровку данных.



В вебе используют криптографические протоколы SSL (secure sockets layer) и TLS (transport layer security). Эти протоколы построены на алгоритме асимметричного ключа, который состоит из двух ключей — публичного и частного. Публичные ключи доступны и сайтам, и браузерам, частные — хранятся на собственных серверах веб-приложений.

Сайты, которые используют HTTPS, имеют уникальный цифровой сертификат (например, SSL-сертификат), который выдан центром сертификации (Certification authority, CA). Когда интернет-пользователь заходит на такой сайт, сервер передает браузеру данные о сертификате и публичный ключ. Веб-браузер использует публичный ключ, чтобы установить цифровую подпись в сертификате. Затем веб-браузер сравнивает подпись с данными, которые получены от CA. Если сертификат действительный, будет установлено соединение с сайтом, а в адресной строке браузера появится пиктограмма в виде зеленого замка. В случае, если на сайте сертификат отсутствует или он не подтвержден браузером, появится соответствующее уведомление, а в адресной строке появится красный замок.

Осталось разобраться, почему в названии алгоритма присутствует слово «асимметричный». Самым примечательным в методе является тот факт, что для шифровки и дешифровки данных используются разные ключи. Например, пользователь передает данные на сайт. В этом случае веб-браузер шифрует данные при помощи публичного ключа. Но дешифровка на сервере осуществляется при помощи частного ключа, который, как мы уже говорили, всегда остается на сайте. Аналогичным образом информация передается с сервера на веб-браузер.

Таким образом, протокол HTTPS отличается от HTTP сложным многоуровневым методом установки соединения и криптошифрованием переданных пакетов данных.

Почему использование протокола HTTP отрицательно влияет на посещаемость сайта?

Использование протокола HTTP негативно влияет на эффективность веб-сайта. Если интернет-пользователь заходит на страницу, у которой нет протокола безопасности SSL/TLS, десктопные веб-браузеры обязательно покажут уведомление об отсутствии сертификата. Мобильные браузеры, особенно Chrome и Opera, вообще не загрузят сайт.



Рядовые пользователи привыкают к определенной системе. Если у сайта нет сертификата безопасности, — это сигнал. Либо владельцы сайта экономят на безопасности, либо веб-ресурс давно не обновлялся и выданный сертификат устарел.

Для владельцев сайтов отказ от использования протокола HTTPS — это верный путь к утрате трафика. Во-первых, часть пользователей не станет посещать сайт, у которого нет SSL-сертификата. Во-вторых, еще с 2014 года Google использует наличие сертификата как сигнал для повышения сайтов в результатах поиска. Соответственно, веб-ресурс без сертификата рискует откатиться на последние страницы в поисковой выдаче.

Как подключить SSL-сертификат

Подключить SSL-сертификат к сайту очень просто. По умолчанию, эту услугу предлагают компании, предоставляющие хостинг. Такие сервисы, как SSLs.com предлагают пользователям различные подписки: от дешевой версии (для личного блога) до пакетного решения (для группы сайтов). В 2020 году можно найти и сервисы, которые предоставляют бесплатный SSL-сертификат. Например, Let's Encrypt обеспечивает бесплатно сертификатами 225 млн сайтов.

С технической точки зрения подключить сертификат безопасности для домена несложно. Например, хостинговые компании автоматизировали эту услугу. В данном случае владельцу сайта достаточно оплатить установленную сумму, чтобы получить работающий результат. Аналогичным образом предоставляют услугу HTTPS некоторые платформы для создания сайтов,

например, Tilda. В разделе Настройки сайта → SEO → Настройка HTTPS пользователь активирует получение сертификата.

Если вы решили устанавливать сертификат безопасности самостоятельно, это можно сделать через панель администратора. Необходимо добавить в соответствующие разделы полученный сертификат (в виде кода) и приватный ключ. В зависимости от выбранной системы администратор либо загружает файл с сертификатом (.crt), либо вставляет код вручную. Приватный ключ копируется и вставляется вручную. Рассмотрим несколько примеров.

В панели правления хостингом cPanel следует перейти в пункт меню Security (Безопасность)> SSL/TLS Manager (Диспетчер SSL/TLS). Для загрузки сертификата используется раздел Generate, view, upload, or delete SSL certificates.

Для загрузки приватного ключа предназначен раздел Setup a SSL certificate to work with your site. В появившемся поле — добавить соответствующие данные.

В панели управления Plesk подключение SSL-сертификата для веб-домена производится в пункте меню Domains> Certificates. Здесь администратор загружает файл с сертификатом и копирует код приватного ключа. После установки сертификата безопасности необходимо перейти в раздел Service Management, где следует остановить и перезапустить веб-сервер Apache кликом по меню Stop and Start WEB Server.

В ISPmanager установка сертификата безопасности — процесс интуитивно понятный. В меню панели существует отдельный пункт:

Затем кликнуть по кнопке «Создать» и в открывшемся окне вставить данные сертификата безопасности и приватного ключа:

Виды сертификатов безопасности

В действительности существует только один вид сертификата — SSL. TSL является новой версией с более совершенным уровнем криптошифрования. Термин «SSL-сертификат» более распространенный, поэтому именно он используется в панелях администрирования и в обозначении услуги. В настоящее время различные центры сертификации предоставляют TSL- и SSL-сертификаты.

Заключение

К 2020 году использование протокола HTTPS является своего рода негласным правилом. И для небольших блогов, и для крупных корпоративных ресурсов важно поддерживать отличную репутацию. Чтобы не отпугнуть посетителей сайта и не допустить хакерской атаки на сайт, владельцы веб-

ресурсов устанавливают SSL-сертификаты. Эта процедура достаточно простая. Ее можно заказать непосредственно у хостинговой компании.

14. IPv4, IPv6

Протокол сетевого взаимодействия TCP/IPv4 используется для передачи зашифрованных данных в сети интернет и локальных подсетях уже более тридцати лет. На его основании создается и поддерживается уникальная адресация сетевого оборудования (узлов). Еще в начале 90-х годов прошлого века был определен основной недостаток данного протокола – ограничение по количеству возможных ip-адресов, которое не может превысить 4,23 миллиарда. В результате была разработана новая система протоколирования сетевого взаимодействия – интернет-протокол IPv6 (Internet Protocol version 6). Однако массовый переход на более прогрессивную технологию обусловлен некоторыми сложностями. Хотя, например, в Соединенных Штатах уже более половины пользователей применяют именно протокол IPv6.

Основные отличия протоколов IPv4 и IPv6

Как уже было сказано, ключевым недостатком протокола четвертой версии TCP/IPv4 является ограниченная масштабируемость уникальных адресов, присваиваемых для идентификации в сетях взаимодействия. Для создания ip-адресов на уровне программных записей используется 32-битная система в формате 0.0.0.0–255.255.255.255. При построении локальных подсетей вводится дополнительный атрибут «маска подсети», записываемая после символа «/». В результате даже крупные ЛВС, объединенные в Ethernet, чаще всего имеют один публичный ip-адрес, выдаваемый провайдером и закрепленный на уровне шлюза (маршрутизатора). Самостоятельный обмен данными на уровне отдельных устройств частной подсети с выходом в паблик-интернет требует сложного администрирования. Для решения задач маршрутизации, требующих получения статических IP-адресов, понадобятся дополнительные финансовые затраты.

В интернет-протоколе нового поколения IPv6 для создания адресной маршрутизации используется 128-битная система записи. В IPv6-адресе записи представляют собой восемь 16-битных блоков, разделенных двоеточиями: 2dfc:0:0:0:0217: cbff: fe8c:0. Общее количество ip-адресов, возможных для распределения, может составить в общей сложности 2128 (приблизительно 340 282 366 920 938 000 000 000 000 000 000 000).

Повсеместное использование данного стандарта позволит полностью решить задачу нехватки сетевых адресов в обозримом будущем.

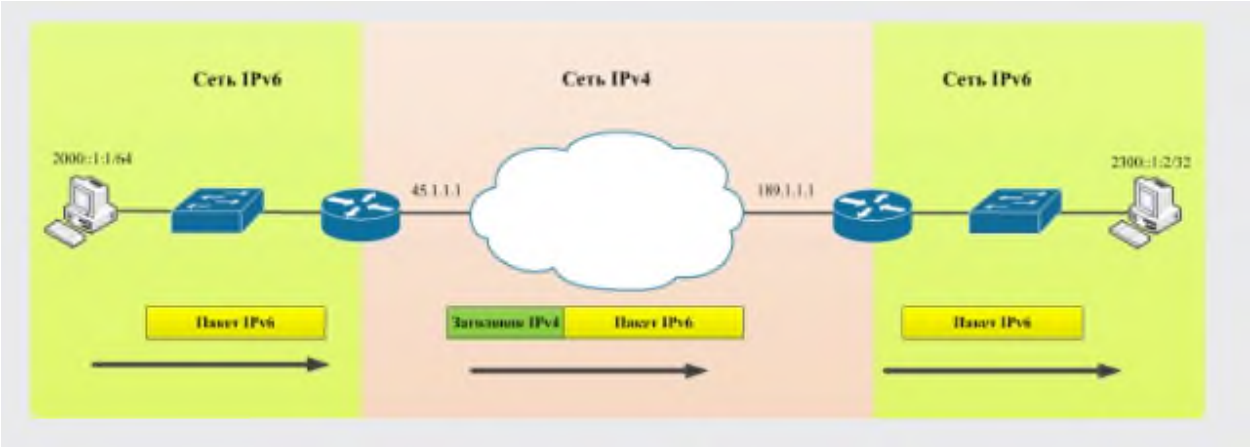
С целью упрощения записи адреса в протоколе IPv6 используется вариант сжатия кода, когда смежные последовательности нулевых блоков заменяются парами символов двоеточия. Например, адрес групповой рассылки FFEA:0:0:0:0:CA28:1012:4254 в сжатой форме будет представлен в укороченном виде FFEA::CA28:1012:4254. Данный механизм упрощает процесс записи, хранения и обработки кода.

По правилам протокола IPv6 назначение сетевых адресов происходит автоматически и уникализируется за счет идентификации на уровне MAC-адреса конкретной единицы оборудования, для которой необходим выход в публичную сеть. Другими словами, каждый домашний компьютер, смартфон, холодильник или стиральная машина с функцией подключения к внешним устройствам получает собственный «белый» ip-адрес для коннекта с другими хостами через интернет. Доступна также произвольная генерация кодов путем администрирования с использованием маршрутизаторов.

Впечатляет минимальный диапазон адресов подсети, получаемых пользователем при подключении по протоколу IPv6. Например, при использовании маски подсети «/128» получаем более 256 адресов.

Спорным является вопрос отличия в скорости передачи трафика по каждому из протоколов. По умолчанию технология протокола IPv6 обеспечивает большую скорость обработки трафика на уровне отдельного оборудования сети в целом. Использование NAT в протоколе IPv4, который обеспечивает трансляцию адресов абонентов и хранение в памяти информации об установленных соединениях, приводит к большой загрузке оборудования. Поэтому в моменты пиковой нагрузки каждый пользователь отмечает резкое падение скорости соединения.

В протоколе IPv6 не применяется обязательная обработка пакетов и отслеживание уже открытых соединений при маршрутизации доступа к хостам. Отсутствие необходимости трансляции значительно снижает ресурсную нагрузку на сетевые устройства. Для пользователя это означает выравнивание скорости интернет-соединения. Провайдеры в такой ситуации могут использовать менее ресурсоемкое, а значит, более дешевое оборудование.



Дополнительные преимущества протокола IPv6

По сравнению с четвертой версией, в протоколе TCP/IPv6 реализован ряд дополнительных функциональных возможностей:

используется более простой заголовок, из него исключены несущественные параметры, что снижает нагрузку на маршрутизаторы при обработке сетевых запросов;

более высокий уровень обеспечения безопасности, аутентификации и конфиденциальности, которые положены в основу данной технологии;

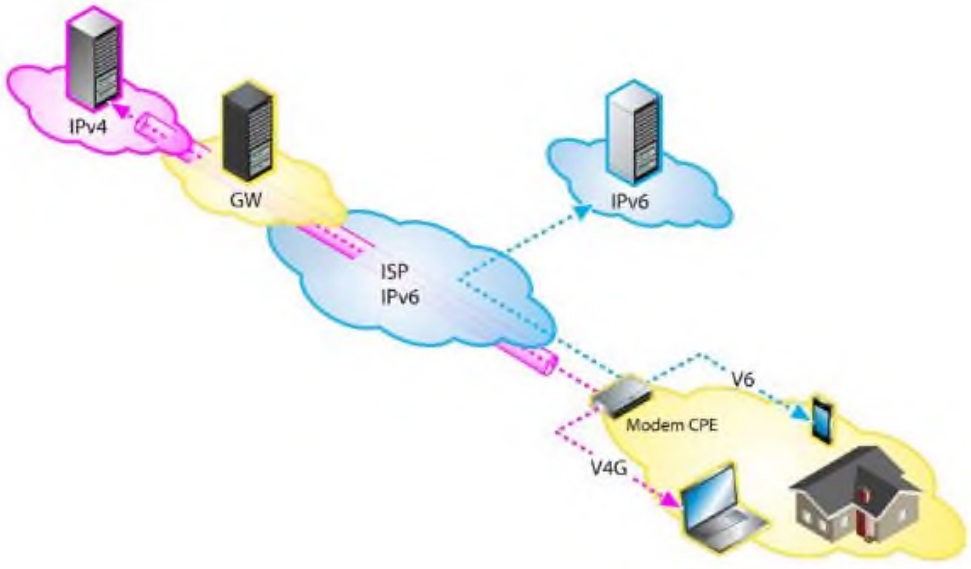
в протоколе реализована функция Quality of Service (QoS), позволяющая определять чувствительные к задержке пакеты;

при передаче широковещательных пакетов используются многоадресные группы;

для реализации технологии мультивещания в IPv6 задействовано встроенное адресное пространство FF00::/8;

для повышения безопасности используется поддержка стандарта шифрования IPsec, который позволяет шифровать данные без необходимости какой-либо поддержки со стороны прикладного ПО.

В настоящее время эксперты ведут дискуссии на предмет обеспечения безопасности данных в случае гибридного применения двух протоколов. Провайдеры выстраивают IPv6-туннели для предоставления пользователям IPv4 доступа к высокоуровневому контенту. Применение данной технологии увеличивает риски хакерских атак. Функция автоконфигурации, когда устройства самостоятельно генерируют IP-адрес на основе MAC-адреса оборудования, может быть использована для незаконного отслеживания конфиденциальных данных пользователей.



Внедрение протокола TCP/IPv6

Несмотря на долгую историю разработки, которая берет начало в 1992 году, тестирование нового протокола состоялось одномоментно 8 июня 2011 года в Международный день IPv6. Эксперимент прошел удачно и предоставил возможность для выработки рекомендаций по дальнейшему совершенствованию данной технологии, ее массовому внедрению.

Первой компанией, внедрившей в 2008 году стандарт протокола IPv6 на постоянной основе, стал Google. Тестирование проводилось в течение четырех лет, было признано успешным. 6 июня 2012 года состоялся Всемирный запуск IPv6. Сегодня мировые лидеры в производстве сетевого оборудования Cisco и D-Link применяют данный сетевой стандарт в своих маршрутизаторах на базовом уровне. В мобильных сетях стандарта LTE поддержка протокола IPv6 является обязательной. IT-компании Google, Facebook, Microsoft и Yahoo используют IPv6 на своих основных web-ресурсах. Протокол получает все большее распространение в корпоративных сетях и при домашнем использовании.

Согласно исследованиям Google, на начало 2020 года доля IPv6 в общемировом сетевом трафике составляла около 30%. В России данный показатель значительно ниже, он составляет приблизительно 4,5% всего трафика. В то же время все большее количество отечественных регистраторов доменов и хостинг-провайдеров переводят свои DNS-серверы на протокол IPv6.

Сложности перехода

Возникает резонный вопрос: если протокол TCP/IPv6 обладает таким количеством преимуществ по сравнению с предшественником, почему бы просто не перейти на него всем миром? Основное препятствие лежит в сфере финансов и временных параметров. Для полномасштабного использования

новой технологии требуются серьезные инвестиции в программно-техническую модернизацию компьютерного парка всех провайдеров.

Использование динамических ip-адресов по протоколу IPv4 позволяет временно сдерживать проблему нехватки уникальных сетевых идентификаторов. Другими словами, проблемы администрирования локальных сетей перекладываются на конечных пользователей, которые вынуждены настраивать сложные схемы маршрутизации подсетей и покупать дополнительные ip-адреса. В то же время рост количества конечных сетевых устройств происходит очень быстро. Внедрение технологий прямой коммуникации даже с обычными бытовыми приборами через интернет требует новых подходов в построении архитектуры их взаимодействия. В связи с этим повсеместный переход на использование стека протокола TCP/IPv6 неизбежен

15. Базовое программирование на Bash. Коды возврата, функции.

BASH (сокращение от Bourne-Again Shell) — самый популярный командный интерпретатор для юниксоподобных операционных систем. Например, окно терминала в Linux работает в том числе благодаря Bash.

Знание BASH пригодится для автоматизации процессов. Допустим, вы любите постоянно пробовать что-то новое и часто меняете операционные системы. Каждый раз переустанавливать все свои конфиги и программы вручную на новую ОС — куча мороки. А благодаря самописным bash-скриптам можно полностью автоматизировать этот процесс и не париться.

Bash — штука мощная и хранит в себе кучу полезных функций. Сегодня мы поговорим о некоторых командах и операторах, которые помогут в автоматизации.

Bash функции

Функция Bash — это, по сути, набор команд, который можно вызывать множество раз. Цель функции — помочь вам сделать ваши сценарии bash более читаемыми и избежать написания одного и того же кода.

По сравнению с большинством языков программирования, функции Bash сильно ограничены. В этом модуле мы рассмотрим основы функций Bash и разберемся, как использовать их в скриптах оболочки shell.

Объявление функции Bash

Синтаксис объявления очень прост и похож на многие популярные языки программирования. Есть два основных формата объявления:

1. Формат может начинаться с имени функции, за которым следуют круглые скобки. Это предпочтительный и наиболее популярный формат.

```
function_name () {
  commands
}
```

Этот же код можно написать в одну строку:

```
function_name () { commands; }
```

2. Второй формат начинается с зарезервированного слова function, за которым следует имя функции.

```
function function_name {
  commands
}
```

Версия в одну строку:

```
function function_name { commands; }
```

Несколько моментов, которые следует запомнить:

- Список команд, находящихся между фигурными скобками {} — это тело функции. Фигурные скобки, окружающие тело функции, должны быть отделены от тела пробелами или начинаться с новой строки.
- Простое определение функции не выполняет её. Чтобы вызвать функцию bash, просто используйте имя созданной функции. Команды между фигурными скобками выполнятся, когда функция вызывается через shell оболочку.
- Соблюдайте порядок написания кода. Определение функции должно быть перед её вызовом.
- При использовании однострочных функций необходимо установить точку с запятой у последней команды.
- Старайтесь давать имена, которые описывают процесс, происходящий внутри тела функции. Чтобы в будущем понимать, что делает программный код, взглянув лишь на название.

Чтобы лучше понять базовые правила, взглянем на следующий пример:

```
#!/bin/bash
hello_world () {
  echo 'привет, мир'
}
hello_world
```

Проанализируем написанный код:

- В третьей мы определяем функцию с именем hello_world и открывая фигурную скобку {, которая отмечает начало тела функции.
- Строка 4 — это тело функции. Обычно, тело занимает более одной строки.
- В пятой строке стоит закрывающая фигурная скобка}. Она определяет конец функции.
- В строке 7 мы выполняем созданную функцию. Выполнять ее можно столько раз, сколько вам нужно.

Если запустить такой скрипт в консоли, он выведет на экран строку:

```
привет, мир
```

Область переменных

Глобальные переменные — это переменные, к которым можно получить доступ из любой точки скрипта независимо от области их объявления. В Bash все переменные по умолчанию определяются как глобальные, даже если они объявлены внутри одной функции.

Локальные переменные могут быть объявлены в теле функции с ключевым словом local, а использоваться могут только внутри неё. Вы можете иметь локальные переменные с одинаковым именем в разных функциях. Чтобы лучше понять, как работает область переменных в Bash, рассмотрим пример:

```
#!/bin/bash
var1='A'
var2='B'
my_function () {
  local var1='C'
  var2='D'
  echo "Внутри функции: var1: $var1, var2: $var2"
}
echo "Перед запуском: var1: $var1, var2: $var2"
my_function
echo "После запуска: var1: $var1, var2: $var2"
```

Этот скрипт начинается с определения двух глобальных переменных var1 и var2. Затем объявление функции, которая присваивает локальной переменной var1 и изменяет глобальную переменную var2.

Если вы запустите сценарий, вы увидите следующие строки:

```
Перед запуском: var1: A, var2: B
Внутри функции: var1: C, var2: D
После запуска: var1: A, var2: D
```

Из этого следует простой вывод:

- Если вы задаёте локальную переменную внутри тела функции с тем же именем, что и существующая глобальная, локальная будет иметь приоритет над глобальной.
- Глобальные переменные могут быть изменены внутри функции.

Возвращаем значения

В отличие от функций на “реальных” языках программирования, функции в Bash не позволяют возвращать значение при вызове. Когда функция bash завершается, ее возвращаемое значение — это значение последнего оператора: 0 — успех, а ненулевое десятичное число в диапазоне от 1 до 255 включительно — для описания сбоя.

Статус возврата значения можно указать с помощью ключевого слова return и оно присваивается переменной $?. Оператор return завершает работу функции. Вы можете думать об этом как о способе выхода из функции.

```
#!/bin/bash
my_function () {
  echo "наш результат"
  return 13
}
my_function
echo $?
```

```
Вывод консоли:
наш результат
13
```

Как же, все-таки, вернуть значение из функции? Нам нужно использовать для этого небольшую хитрость — можно присвоить результат функции глобальной переменной:

```
#!/bin/bash
my_function () {
  func_result="наш результат"
}
my_function
echo $func_result
```

После выполнения этого кода вы должны увидеть строку на экране:

```
наш результат
```

Другой, подходящий вариант для возврата значения из функции — отправить значение в stdout с помощью echo или printf, как показано ниже:

```
#!/bin/bash
my_function () {
  local func_result="это тест"
  echo "$func_result"
}
func_result="$(my_function)"
echo $func_result
```

Результат выполнения будет таким же.

Вместо того, чтобы просто выполнять функцию, которая будет печатать сообщение в stdout, мы присваиваем результат переменной func_result с помощью механизма $ (). Используя этот метод, глобальная переменная func_result будет содержать результат функции.

Передача аргументов

Чтобы передать любое количество аргументов функции bash, просто поместите их сразу после имени функции, разделяя каждую пробелом. Соблюдайте основные рекомендации при передаче параметров:

- Переданные параметры $1, $2, $3 … $n, соответствуют номеру параметра по порядку, заданными после имени функции.
- Переменная $0 зарезервирована для имени функции.
- Переменная $# содержит общее количество аргументов, переданных функции.
- Переменная $* или $@ содержит все параметры, переданные функции.

Пример передачи аргументов:

```
#!/bin/bash

greeting () {
  echo "Привет, $1"
}
greeting "Алиса"
```

Результатом выполнения этого кода будет строка:

```
Привет, Алиса
```

Теперь у вас есть некоторые знания для того, чтобы правильно писать функции bash.

Примеры

Функция таймер, которая считает сколько прошло времени:

```
timer() { local t; t=$(date '+%s'); test $# -gt 0 && let t-=$1; echo "$t"; }
...
t=$(timer)
...
...
t=$(timer $t)
echo "time used = $t secs"
```

Код возврата

Код возврата (англ. Exit status) программы, — это целочисленное значение, которое дочерний процесс возвращает родительскому процессу в момент завершения.

При выполнении компьютерных программ операционная система создаёт сущность, называемую процессом, с помощью которой она отслеживает и ведёт учёт выполнения программы. В многозадачных операционных системах, таких как Unix или Linux, новый процесс может быть порождён существующим процессом. Процесс-создатель называется «родительским», а создаваемый процесс — «дочерним». После создания дочерний процесс выполняется параллельно с родительским. Такая техника порождения дочерних процессов используется, чтобы передать часть работы дочернему процессу, в ситуациях, когда родительскому процессу не обязательно тратить на неё время. Когда дочерний процесс завершает своё выполнение, он автоматически либо явно по указанию программиста совершает системный вызов exit, передавая ему целое число. В результате вызова exit целое число передаётся родительскому процессу, который может получить его с помощью системного вызова wait.

Команда exit в Bash и коды выхода

Часто при написании сценариев Bash вам необходимо завершить сценарий при выполнении определенного условия или выполнить действие на основе кода выхода команды.

Статус выхода

Каждая команда оболочки возвращает код выхода, когда она завершается успешно или безуспешно.

По соглашению нулевой код выхода указывает, что команда завершилась успешно, а ненулевое значение означает, что произошла ошибка.

Специальная переменная $? возвращает статус выхода последней выполненной команды:

```
date &> /dev/null
echo $?
```

Команда date завершена успешно, код выхода равен нулю:

```
0
```

Если вы попытаетесь запустить ls в несуществующем каталоге, код выхода будет отличным от нуля:

```
ls /nonexisting_dir &> /dev/null
echo $?
```

```
2
```

Код состояния можно использовать для выяснения причины сбоя команды. На странице руководства каждой команды содержится информация о кодах выхода.

При выполнении многокомандного конвейера статус выхода конвейера соответствует состоянию последней команды:

```
sudo tcpdump -n -l | tee file.out
echo $?
```

В приведенном выше примере echo $? напечатает код выхода команды tee.

Команда exit

Команда exit закрывает оболочку со статусом N Он имеет следующий синтаксис:

exit N

Если N не задано, код состояния выхода — это код последней выполненной команды.

При использовании в сценариях оболочки значение, указанное в качестве аргумента команды exit возвращается оболочке как код выхода.

Примеры

Статус выхода команд может использоваться в условных командах, таких как if . В следующем примере grep завершит работу с нулем (что означает истину в сценариях оболочки), если «строка поиска» найдена в filename :

```
if grep -q "search-string" filename then
  echo "String found."
else
  echo "String not found."
```

```
fi
```

При запуске списка команд, разделенных && (И) или || (ИЛИ), статус выхода команды определяет, будет ли выполнена следующая команда в списке. Здесь команда mkdir будет выполнена, только если cd вернет ноль:

cd /opt/code && mkdir project

Если сценарий завершается exit без указания параметра, код выхода из сценария — это код последней команды, выполненной в сценарии.

~/script.sh

```
#!/bin/bash

echo "doing stuff..."
exit
```

Использование только exit — это то же самое, что и exit $? или пропуская exit .

Вот пример, показывающий, как завершить сценарий, если он запущен пользователем без полномочий root:

```
#!/bin/bash
if [[ "$(whoami)" != root ]]; then
  echo "Only user root can run this script."
  exit 1
fi
echo "doing stuff..."
exit 0
```

Если вы запустите сценарий как root, код выхода будет нулевым. В противном случае скрипт выйдет со статусом 1.

Выводы

Каждая команда оболочки возвращает код выхода при завершении. Команда exit используется для выхода из оболочки с заданным статусом.

16. Regexp и их использование для синтаксического анализа. Полезные утилиты.

Регулярные выражения

Регулярные выражения (regular expressions) — это текстовый шаблон, который соответствует какому-то тексту. И всё? Да, это всё, для чего они нужны.

Что можно делать с помощью регулярных выражений:

Проверять то, что вводит пользователь, чтобы быть уверенным в правильности данных (например, правильно ли пользователь ввёл e-mail или ip-адрес).

Разбирать большой текст на меленькие кусочки (например, выбирать данные из большого лога).

Делать замены по шаблону (например, убирать непечатаемые символы из XML).

Большинство современных языков программирования и текстовых редакторов поддерживают регулярные выражения. Поддержим их и мы.

/Быть или не быть/ugi ¶

Синтаксис регулярных выражений прост и логичен. Он разделяется на символ-разделитель (он идёт в начале и конце выражения, обычно это /), шаблон поиска и необязательные модификаторы.

Формальный синтаксис такой:

[разделитель][шаблон][разделитель][модификаторы]

Разделителем может быть любой символ, но обычно это / или ~. Важно лишь то, чтобы шаблон начинался и заканчивался одним и тем же разделителем. В самом конце регулярных выражений идут модификаторы, которые нужны, чтобы менять логику работы шаблонов (например, делать регистронезависимый поиск).

Давайте разберём выражение /Быть или не быть/ugi:

/ - начальный символ-разделитель Быть или не быть - шаблон поиска / - конечный символ-разделитель ugi - модификаторы (UTF-8, global, case insensitive)

Данное регулярное выражение будет искать текст. «Быть или не быть» не зависимо от регистра по всему тексту неограниченное количество раз. Модификатор u нужен для того, чтобы явно указать, что текст у нас в юникоде, то есть содержит символы, отличные от латиницы. Модификатор i включает

регистронезависимый поиск. Модификатор g указывает поисковику идти до победного конца, иначе он остановится после первого удачного совпадения.

"Петя любит Дашу".replace(/Дашу|Машу|Сашу/, "Катю")ю

Не трудно догадаться, что результатом работы js-выражения выше будет текст "Петя любит Катю". Даже, если Петя неровно дышит к Маше или Саше, то результат всё равно не изменится.

Регулярные выражения в LINUX

Интересным вопросом в Linux системах, является управление регулярными выражениями. Это полезный и необходимый навык не только профессионалам своего дела, системным администраторам, но, а также и обычным пользователям линуксоподобных операционных систем. В данной модуле постараемся раскрыть, как создавать регулярные выражения и как их применять на практике в каких-либо целях. Основной областью применение регулярных выражений является поиск информации и файлов в линуксоподобных операционных системах.

Для работы в основном используются следующие символы:

- "\text" - слова начинающиеся с text
- "text/" - слова, заканчивающиеся на text
- "^" - начало строки
- "$" - конец строки
- "a-z" - диапазон от a до z
- "[^t]" - не буква t
- "\[" - воспринять символ [буквально
- "." - любой символ
- "a|z" - a или z

Регулярные выражения в основном используются со следующими командами:

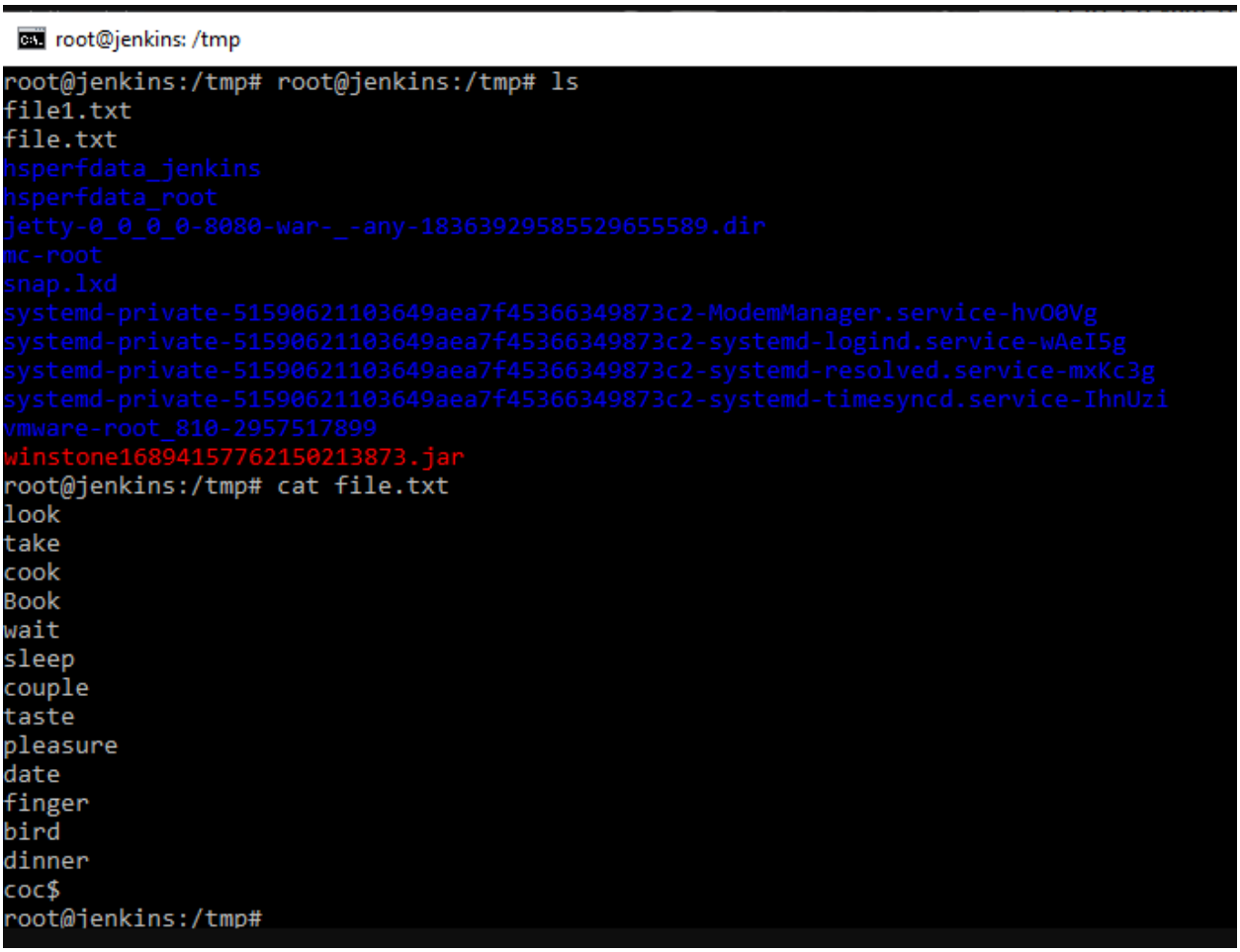
grep - утилита поиска по выражению

- egrep - расширенный grep
- fgrep - быстрый grep
- rgrep - рекурсивный grep
- sed - потоковый текстовый редактор.

А особенно с утилитой grep. Данная утилита используется для сортировки результатов чего либо, передавая ей результаты по конвейеру. Эта утилита осуществляет поиск и передачу на стандартный вывод результат его. ЕЕ можно запускать с различными ключами, но можно использовать ее другие варианты, которые представлены выше.

И есть еще потоковый текстовый редактор. Это не полноценный текстовый редактор, он просто получает информацию построчно и обрабатывает. После чего выводит на стандартный вывод. Он не изменяет текстовый вывод или текстовый поток, он просто редактирует перед тем как вывести его для нас на экран.

Начнем со следующего. Создадим один пустой файл file1.txt, через команду touch. Создадим в текстовом редакторе в той же директории файл file.txt.

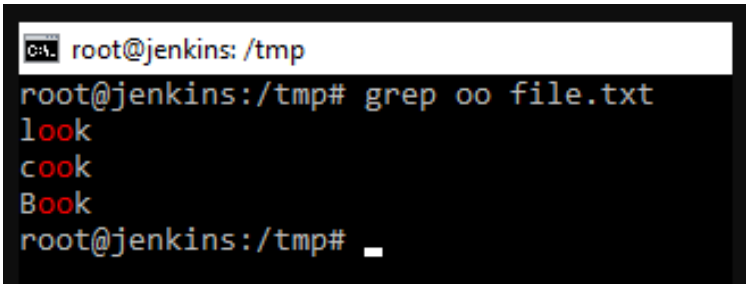


Как мы видим в файле file.txt просто набор слов. Далее мы с помощью данных слов посмотрим, как работают команды.

Первая команда - GREP

man grep

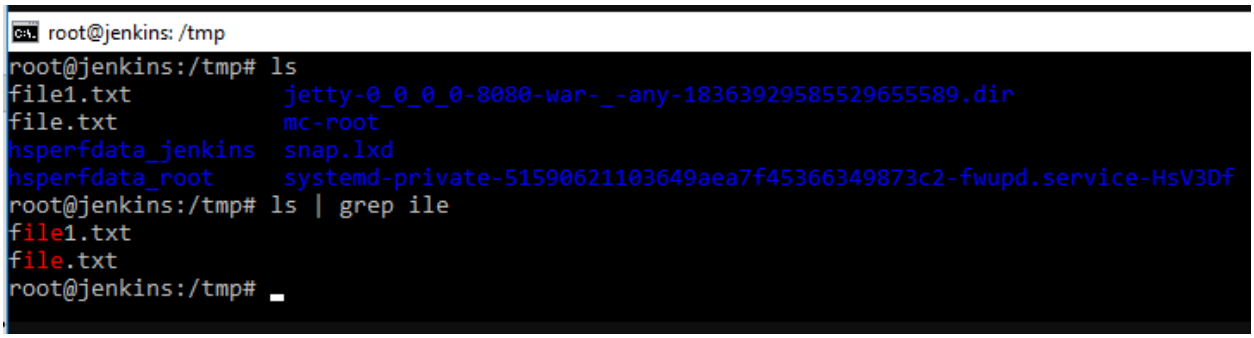
Получаем справку по данной команде. Как можно понять из справки команда grep и ее производные — это печать линий совпадающих шаблонов. Проще говоря, команда grep помогает сортировать те данные, что мы даем команде, через знак конвейера на ввод. Причем в мануале мы можем видеть egrep, fgrep и т.д. данные команды мы можем не использовать. Использовать можно только grep с ключами различными, т. е. ключи просто заменяют эти команды. Можно на примере посмотреть, как работает данная команда. Например, grep oo file.txt



На картинке видно, что команда из указанного файла выбрала по определенному шаблону "oo". Причем даже делает красным цветом подсветку. Можно добавить еще ключик -n, тогда данная команда еще и выведет номер строки, в которой находится то, что ищется по шаблону. Это полезно, когда работаем с каким-нибудь кодом или сценарием. Когда необходимо, что-то найти. Сразу видим, где находится объект поиска или что-то ищем по логам.

При использовании шаблона очень важно понимать, что команда grep, чувствительна к регистрам в шаблонах. Это означает, что Boo и boo это разные шаблоны. В одном случае команда найдет слово, а в другом нет. Можно команде сказать, чтобы она не учитывала регистр. Это делается с помощью ключа -i.

Посмотрим содержимое нашего каталога командой ls, а затем отфильтруем только то, что заканчивается на "ile".



Посмотрим содержимое нашего каталога командой ls

Получается следующее, когда мы даем на ввод команде grep шаблон и где искать, он работает с файлом, а когда мы даем команду ls она выводи содержимое каталога и мы это содержимое передаем по конвейеру на команду grep с заданным шаблоном. Соответственно grep фильтрует переданное содержимое согласно шаблону и выводит на экран. Получается, что команде grep дали, то команда и обработала.

```
root@jenkins:/tmp# cat file.txt | grep oo
look
cook
Book
root@jenkins:/tmp#
```

Наглядно можно посмотреть на рисунке выше. Мы просматриваем командой cat содержимое файла и подаем на ввод команде grep с фильтрацией по шаблону.

Давайте найдем файлы, в которых содержится сочетание "ple". grep ple file.txt в данном случае команда нашла оба слова содержащие шаблон. Давайте найдем слово, которое будет начинаться с "ple". Команда будет выглядеть следующим образом: grep ^ple file.txt. Значок "^" указывает на начало строки. Противоположная задача найти слова, заканчивающиеся на "ple". Команда будет выглядеть следующим образом grep ple$ file.txt. Т.е. применять к концу строки, говорит значок "$" в шаблоне.

```
root@jenkins: /tmp
root@jenkins:/tmp# grep ple file.txt
couple
pleasure
root@jenkins:/tmp# grep ^ple file.txt
pleasure
root@jenkins:/tmp# grep ple$ file.txt
couple
root@jenkins:/tmp#
```

Можно дать команду grep .o file.txt. В данном выражении знак "." , заменяет любую букву.

```
root@jenkins: /tmp
root@jenkins:/tmp# grep .o file.txt
look
cook
Book
couple
coc$
root@jenkins:/tmp# grep .ple file.txt
couple
root@jenkins:/tmp#
```

Как вы видите вывод шаблона ".ple" вывел только одно слово т.к только слово couple удовлетворяло шаблону , т.к перед "ple" должен был содержаться еще один символ любой.

Попробуем рассмотреть другую команду egrep.

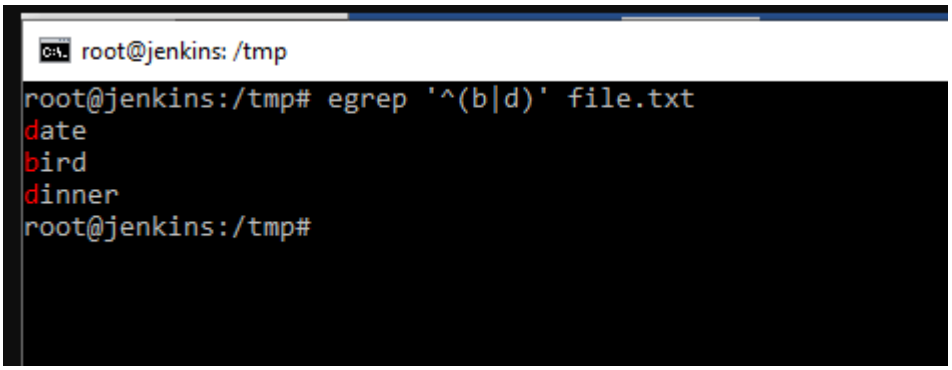
Egrep (extended grep)

man egrep - отошлет к справке по grep.

Данная команда позволяет использовать более расширенный набор шаблонов. Рассмотрим следующий пример команды:

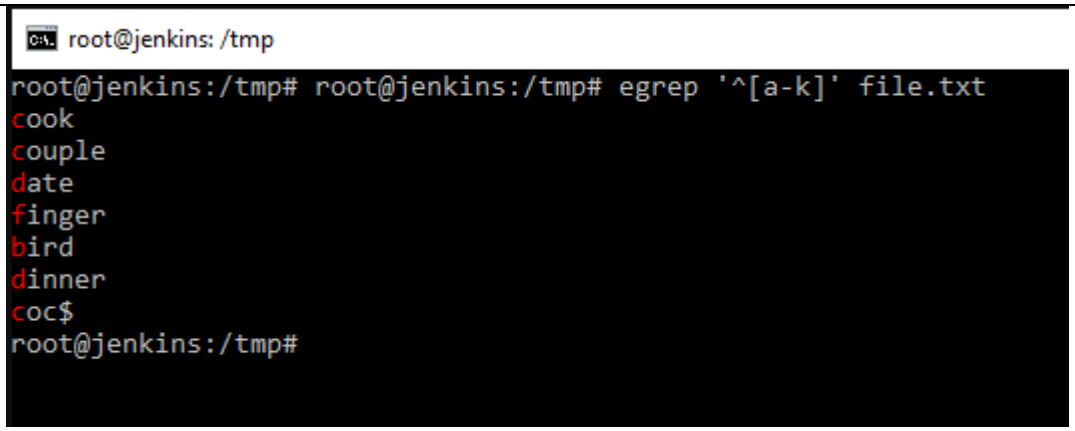
```
egrep '^(b|d)' file.txt
```

Шаблон заключается в одинарные кавычки, для того чтобы экранировать символы, и команда egrep поняла, что это относится к ней и воспринимала выражение как шаблон. Сам же шаблон означает, что поиск будет искать слова, в начале строки (знак ^), содержащие букву b или d.



Мы видим, что команда вернула слова, начинающиеся с буквы b или d. Рассмотрим другой вариант использования команды egrep. Например:

```
egrep '^[a-k]' file.txt
```



Получим все слова, начинающиеся с "a" по "к". Знак "[]" - диапазона. Как мы видим слова, начинающиеся с большой буквы, не попали. Все эти регулярные выражения очень пригодятся, когда мы что-то ищем в файлах логах.

Усложним еще шаблон. Возьмем следующий:

```
egrep '^[a-k]|[A-K]' file.txt
```

Усложняя выражение, мы добавили диапазон заглавных букв сказав команде grep искать диапазон маленьких или диапазон больших букв с начала строки.

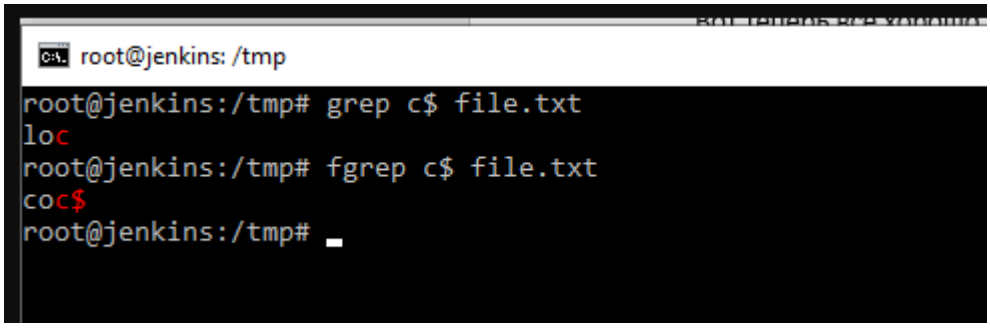
```
root@jenkins:/tmp# egrep '^([a-k]|[A-K])' file.txt
cook
Book
couple
date
finger
bird
dinner
coc$
root@jenkins:/tmp#
```

Вот теперь все хорошо. Слова с Заглавными буквами тоже отобразились. Как вариант egrep можно запускать просто grep с ключиком -e.

Про fgrep

man fgrep - отошлет к справке по grep. Команда fgrep не понимает регулярных выражений вообще.

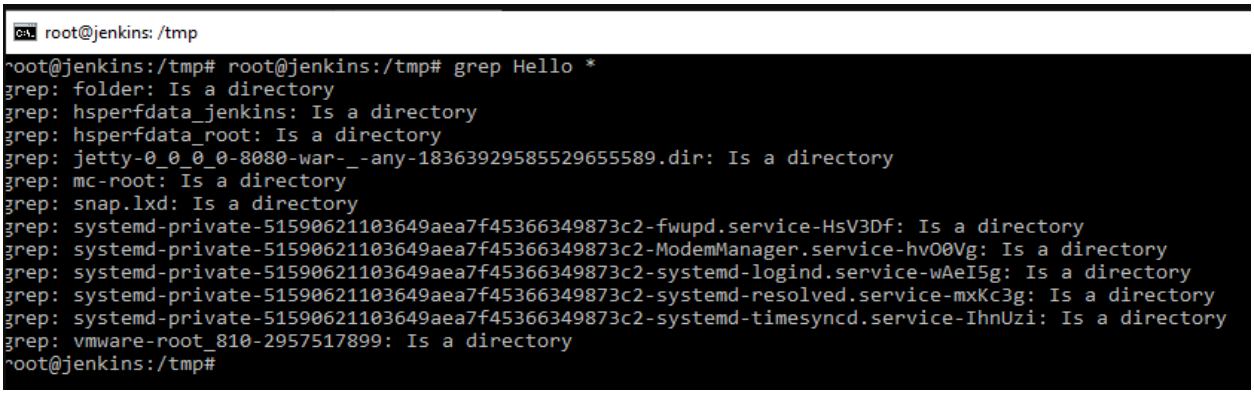
Получается следующим образом если мы вводим: egrep c$ file.txt. То команда согласно шаблону, ищет в файле букву "c" в конце слова. В случае же с командой fgrep c$ file.txt, команда будет искать именно сочетание "c$". Т. е. команда fgrep воспринимает символы регулярных выражений, как обычные символы, которые ей нужно найти, как аргументы.



Рекурсивный rgrep

Создадим каталог mkdir folder . Создадим файл great.txt в созданной директории folder со словом Hello при помощью команды echo "Hello" folder/great.txt

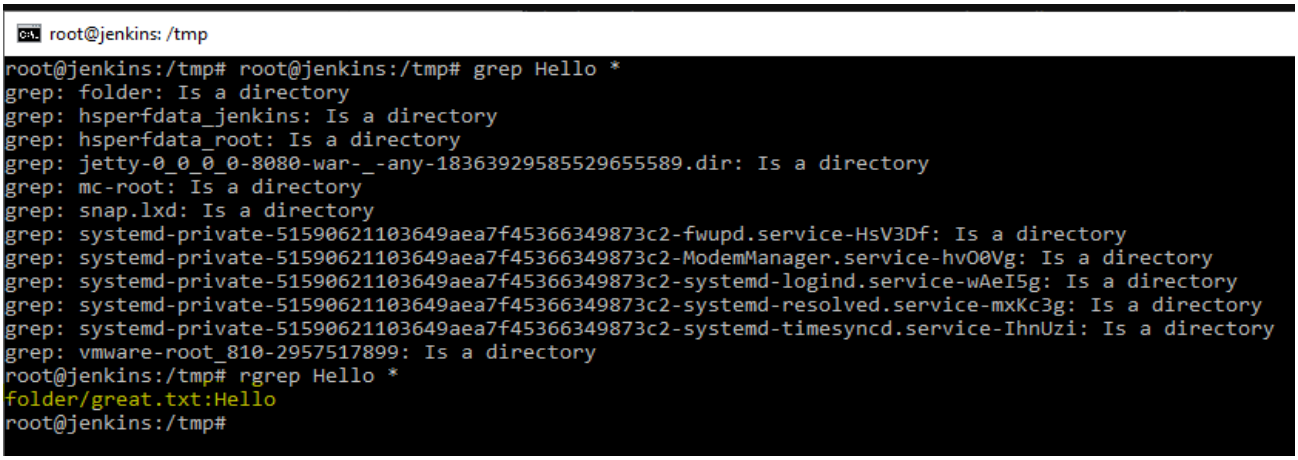
И если мы скажем grep Hello * , поищи слово Hello в текущей директории. Получится следующая картина.



Как мы видим, grep не может искать в папках. Для таких случаев и используется утилита rgrep.

```
rgrep Hello *
```

Дает следующую картину.

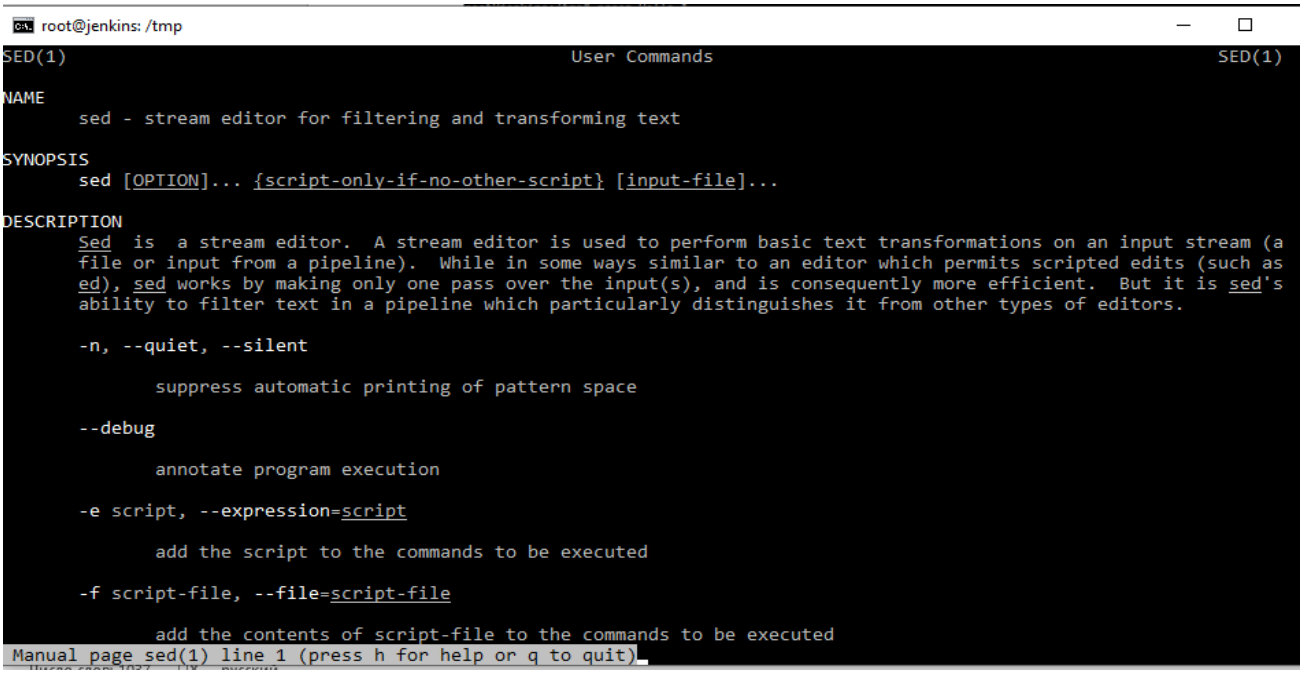


Совершенно спокойно в папке найдено было то, что подходило под шаблон.

Данная утилита пробежалась по всем папкам и файлам в них и нашла подходящее под шаблон слово. Т. е. если нам необходимо провести поиск по всем файлам и папкам, то необходимо использовать утилиту rgrep.

Команда sed

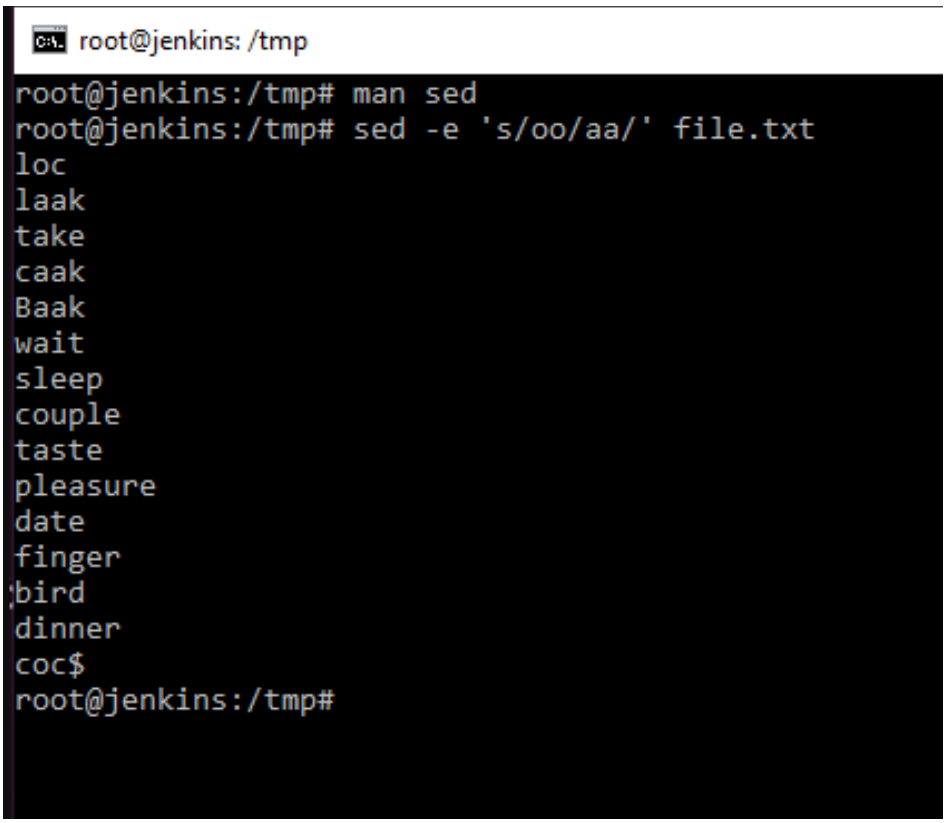
man sed - стрим редактор. Т. е. потоковый редактор для фильтрации и редактирования потока данных.



Например, sed -e ‘s/oo/aa’ file.txt - открыть редактор sed и заменить вывод всех oo на aa в файле file.txt. Нужно понимать, что в результате данной команды изменения в файле не произойдут. Просто данные из файла будут взяты и с изменениями выведены на стандартный вывод, т.е. экран. Для

сохранения результатов мы можем сказать, чтобы вывел в новый файл указав направление вывода.

```
sed -e ‘s/oo/aa’ file.txt   newfile.txt
```



В данном редакторе мы можем ему сказать использовать регулярные выражения, для этого необходимо добавить ключ -r. У данного редактора очень большой функционал.

17. Разбор скриптов и написание своих скриптов. Linter. Shell check.

В простейшем случае шелл скрипт (shell script) — это не что иное, как простая последовательность команд системы, записанная в простой текстовый файл. Создание скриптов поможет сохранить ваше время и силы, которые тратятся на ввод последовательности команд всякий раз, когда необходимо их выполнить.

Написание shell-скриптов — занятие увлекательное. Скрипты командной строки помогают автоматизировать повседневные дела. Можно создать нечто прекрасное (или не очень), однако, если уж что-то писать, хорошо бы точно знать, что код получается именно таким, каким он нужен программисту. Скрипт, написанный некачественно, может представлять опасность. Большинство новичков пишут скрипты, копируя фрагменты кода со StackOverflow, находя то, что им нужно, в Google, или пользуясь сайтами с вопросами и ответами по Linux. Такой подход к программированию

выливается в некачественный код и в появление ошибок. Вот, например, команда rm, выполнение которой приведёт к катастрофе, так как переменная VAR не определена:

```
rm -rf "/$VAR/*"
```

Командная строка и те невообразимые вещи, которые с ее помощью можно творить, — визитная карточка UNIX и всех ее наследников. А где есть командная строка, там есть скрипты. Мы рассмотрим наиболее полезные из них, те, что ты сможешь применять ежедневно для решения самого разного круга задач, начиная от сводки погоды и веб-сервера в одну строку и заканчивая ботом для твиттера в десять строк и скриптом для автоматического запуска любого торрент-клиента.

Для решения многих задач консоль и скрипты годятся гораздо лучше графического интерфейса и поэтому пренебрегать ими никак нельзя. Тем более что любая DE позволяет создать для скрипта иконку, так что для его запуска даже не надо будет открывать консоль.

Простые примеры

Итак, сразу перейдем к примерам:

```
$ curl ifconfig.co
```

Эта простая команда покажет тебе внешний IP — идеальный вариант, если в Сеть ты ходишь через роутер. Все, что она делает, — просто обращается к серверу ifconfig.co, который возвращает обратно IP-шник одной строкой вместо полноценной веб-страницы.

И да, это вовсе не скрипт, это просто команда, но, чтобы превратить команду в скрипт, достаточно поместить ее в текстовый файл и первой строкой добавить так называемый шебанг, то есть символы #!, за которыми следует имя командного интерпретатора:

```
#!/bin/bash
curl ifconfig.co
```

Далее скрипт сохраняем в каталог ~/bin и назначаем права на исполнение:

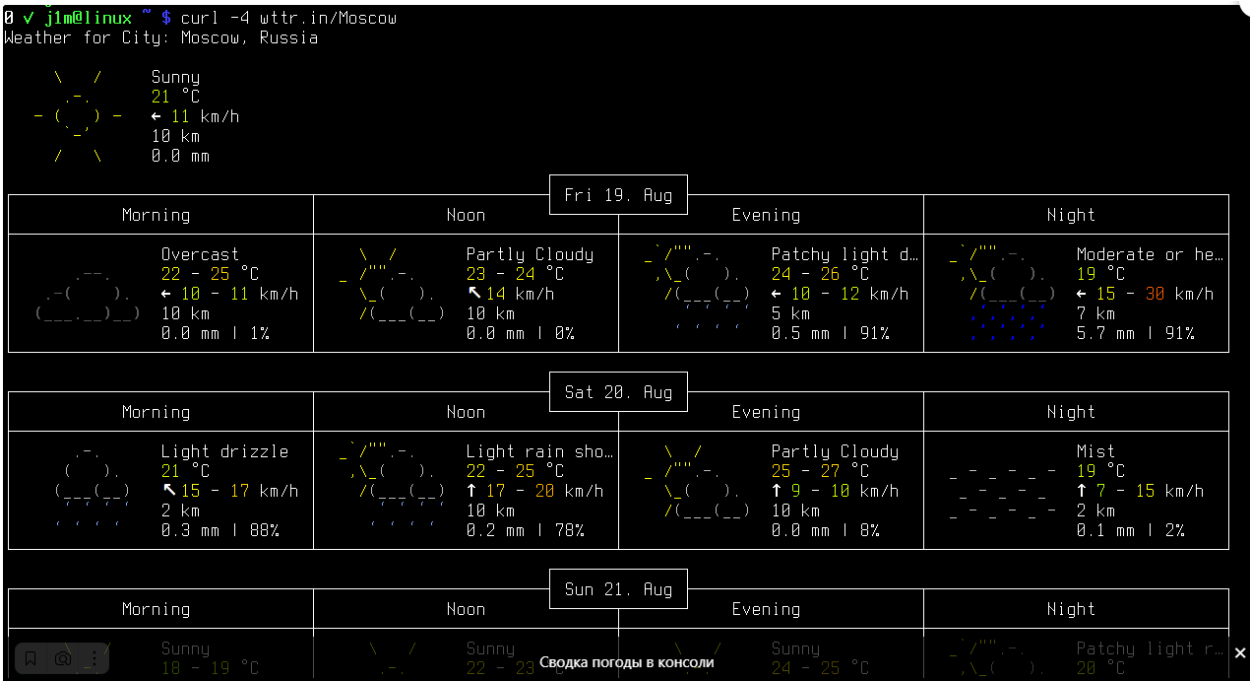
```
$ chmod +x ~/bin/myip.sh
```

Теперь его можно вызывать из командной строки командой myip.sh.

Идем дальше.

```
#!/bin/sh
curl -4 wttr.in/Moscow
```

Этот скрипт позволяет получить сводку погоды на четыре дня. Принцип тут такой же, как в случае с ifconfig.co.




```
#!/bin/sh
dig +short txt $1.wp.dg.cx
```

А так можно получить краткое описание чего-либо в Википедии, причем с помощью DNS-запроса вместо обращения к веб-серверу. Кстати, веб-сервер через командную строку тоже очень легко создать:

```
#!/bin/sh
while ( nc -l 80 < file.html > : ) ; do : ; done
```

Данный скрипт основан на утилите netcat (nc), которую называют швейцарским армейским ножом для сетевых операций. Скрипт запускает цикл, выполняющий команду nc, которая слушает 80-й порт и в ответ на запрос отдает *file.html*, отправляя переданный запрос в никуда (символ означает noop, то есть пустую операцию).

С помощью простых скриптов и команд можно запросто слушать интернет-радио:

```
#!/bin/sh
mpv --volume=50 -playlist ~/16bit.fm_128.m3u
```

Естественно, плей-лист в формате M3U необходимо заранее скачать с сайта радиостанции. Кстати, если запустить MPlayer с аргументом *--input-ipc-server=/tmp/mpvsocket*, им можно будет управлять, записывая команды в файл. Например, настроить громкость:

```
echo 'volume +10' | socat - /tmp/mpvsocket
```

Создай два скрипта: один для запуска, другой для остановки радио (со строкой killall mpv), повесь их на рабочий стол и настрой горячие клавиши DE на управление воспроизведением. Вуаля, у тебя готов плеер для интернет-радио, запустить который можно, просто кликнув по иконке на рабочем столе. И он почти не будет расходовать память или занимать трей.

Но отвлечемся от сетевых операций и вернемся к локальным делам.

```
#!/bin/sh
tar -czf "../${PWD##*/}.tar.gz" .
```

Это один из моих любимых скриптов. Он создает архив tar.gz текущего каталога. Особого внимания здесь заслуживает конструкция ${PWD##*/}, которая берет полный путь до текущего каталога (переменная $PWD) и удаляет из него первую часть вплоть до последнего слеша, оставляя, таким образом, только имя самого каталога. Далее к нему добавляется расширение tar.gz. Более подробно о таких конструкциях ты можешь прочитать в man bash.

```
#!/bin/sh
while true; do
   inotifywait -r -e MODIFY КАТАЛОГ && ТВОЯ_КОМАНДА
done
```

А это уже скрипт, который запускает команду в ответ на изменение файлов в каталоге. Ее можно использовать для множества разных целей, например для автоматического включения плеера при сохранении MP3-файла. Или просто выводить уведомление на десктоп, используя в качестве команды notify-send:

```
notify-send "Файл изменен"
```

Десктоп

Раз уж мы заговорили о десктопе, то продолжим. Как и консоль, его тоже можно заскриптовать. Вот, например, скрипт, загружающий случайные обои, опубликованные на reddit-канале wallpaper:

```
#!/bin/bash
```

```
wget -O - http://www.reddit.com/r/wallpaper |\
  grep -Eo 'http://i.imgur.com[^&]+jpg' |\
  shuf -n 1 |\
  xargs wget -O background.jpg
feh --bg-fill background.jpg
```

Здесь все просто. С помощью wget скрипт загружает страницу www.reddit.com/r/wallpaper, передает ее grep, который ищет на ней ссылки на imgur, выбирает случайную ссылку с помощью shuf, загружает ее опять же с помощью wget и устанавливает в качестве обоев, используя команду feh (это такой миниатюрный просмотрщик изображений, его нужно предварительно установить). Скрипт можно добавить на рабочий стол, и тогда по клику у тебя будут меняться обои.

```
#!/bin/sh
state=`synclient | grep TouchpadOff | cut -d '=' -f 2`
if [ $state = "1" ]; then
  synclient TouchpadOff=0
else
  synclient TouchpadOff=1
fi
```

А это скрипт для включения/выключения тачпада ноутбука: включает, если отключен, и наоборот. В своей работе использует утилиту synclient, позволяющую управлять тачпадами производства Synaptics (90% тачпадов делают они). При запуске без аргументов утилита выводит различную информацию о тачпаде, в том числе строку TouchpadOff = 1, если он активирован, и TouchpadOff = 2, если отключен. Скрипт находит это значение и в зависимости от состояния тачпада включает или отключает его.

```
!#/bin/bash
mpv tv:// -frames 3 -vo jpeg
mv 00000003.jpg photo.jpg
rm -f 0000*.jpg
```

А так можно сделать снимок с помощью веб-камеры. Скрипт использует видеоплеер mpv, чтобы записать первые три кадра, снятые камерой, в JPEG-файлы с именами 0000000.jpg, 00000002.jpg, 00000003.jpg, затем переименовывает третий снимок в файл photo.jpg, а остальные удаляет. Три снимка необходимы для того, чтобы камера успела провести инициализацию, обычно первые два получаются просто черными. Иногда изображение выходит перевернутым; чтобы это исправить, mpv следует запускать с флагом -vf flip:

```
$ mpv tv:// -frames 3 -vf flip -vo jpeg
```

Ту же самую команду можно использовать для создания полноценной камеры слежения, которая делает снимки в моменты, когда юзер прикасается к мыши:

```
#!/bin/bash
while true; do
   sudo cat /dev/input/mouse0 | read -n1
   mpv tv:// -frames 3 -vo jpeg
   mv 00000003.jpg `date +%F-%H-%M`.jpg
   rm -f 0000*.jpg
   sleep 10
done
```

Скрипт входит в бесконечный цикл, ожидая данные на устройстве /dev/input/mouse0. Если данные есть, значит, мышь сдвинулась или была нажата одна из ее клавиш. После этого он использует mpv, чтобы сделать три снимка, дает третьему снимку имя текущей даты и удаляет остальные.

Для записи полноценного видео с веб-камеры можно использовать такой скрипт:

```
#!/bin/bash
mencoder tv:// -tv driver=v4l2:width=800:height=600:device=/dev/video0:fps=30:outfmt=yuy2:force audio:alsa:adevice=hw.2,0 -ovc lavc -lavcopts vcodec=mpeg4:vbitrate=1800 -ffourcc xvid -oac mp3lame -lameopts cbr=128 -o video.avi
```

В результате ты получишь video.avi в формате MPEG4 с битрейтом 1800 и аудиодорожкой в формате MP3 с битрейтом 128.

```
#!/bin/bash
ffmpeg -f x11grab -r 25 -s 1366x768 -i :0.0 screencast.mpg
```

А так ты можешь записать скринкаст. 1366x768 — разрешение рабочего стола. Просто сделать скриншот отдельного окна всегда можно с помощью команды import:

```
import screenshot.png
```

После ее запуска значок мыши изменится на «прицел», с помощью которого можно выбрать окно. Повесив эту команду на клавиатурную комбинацию, ты получишь практически идеальную систему снятия

скриншотов, абсолютно не жрущую память, как это делают специализированные приложения, постоянно висящие в трее.

Подключить и настроить внешний монитор тоже можно из командной строки:

```
#!/bin/sh

if [ -z "$1" ]; then
   exit
fi

if [ $1 == "off" ]; then
   xrandr --output VGA-0 --off
   xrandr -s 0
else if [ $1 == "on"]; then
   xrandr --output LVDS --auto --primary --output VGA-0 --auto --left-of LVDS
   xrandr --newmode "1920x1080" 173.00 1920 2048 2248 2576 1080 1083
1088 1120 -hsync +vsync
   xrandr --addmode VGA-0 1920x1080
   xrandr --output VGA-0 --mode 1920x1080
fi

xrandr --dpi 96
```

Данный скрипт предполагает, что основной монитор носит имя LVDS, а внешний — VGA-0. Это стандартная ситуация для ноутбуков; если ты не уверен, можешь проверить вывод команды xrandr: при передаче скрипту аргумента *off* он отключает внешний монитор, аргумент *on*, в свою очередь, включает его, располагая по левую сторону от основного (аргумент *--left-of LVDS* в первой команде). Далее скрипт добавляет новую конфигурацию для монитора с разрешением 1920 x 1080 и активирует его. В самом конце скрипт устанавливает дефолтное значение DPI — как показывает практика, при подключении монитора с другим разрешением оно часто слетает.

На самом деле в большинстве случаев команды *xrandr --newmode ...* и *xrandr --addmode ...* не нужны, так как Xorg может получить конфигурацию монитора и поддерживаемые им разрешения с помощью EDID. Иногда, однако, этого не происходит, и строку конфигурации, указываемую после аргумента *--newmode*, приходится генерировать самостоятельно с помощью инструмента cvt:

```
$ cvt 1920 1080
```

Он же поможет сгенерировать нестандартное разрешение, «не поддерживаемое» монитором по умолчанию.

Google, Twitter, Dropbox и торренты

Отвлечемся от десктопных дел и поговорим о сетевых сервисах. Начнем, разумеется, с Google. Вот так будет выглядеть скрипт для получения первых десяти результатов поиска:

```
#!/bin/bash
Q="$@"
URL='https://www.google.de/search?tbs=li:1&q='
AGENT="Mozilla/4.0"
stream=$(curl -A "$AGENT" -skLm 10 "${GOOG_URL}${Q//\ /+}" | grep -oP '\/url\?q=.+?&amp' | sed 's|/url?q=||; s|&amp||')
echo -e "${stream//\%/\x}"
```

Скрипт делает запрос к Google с помощью уже знакомого нам curl, заменяя пробелы в поисковой строке на плюсы. Далее выискивает в ответном HTML ссылки и выводит их на экран. Все просто, хоть и кажется сложным.

Второй популярный сервис — YouTube:

```
#!/bin/bash
mpv -fs -quiet `youtube-dl -g "$1"`
```

Здесь все совсем просто. Скрипт всего лишь проигрывает видео с указанным в аргументе ID с помощью плеера mpv. Естественно, youtube-dl придется установить заранее.

Как насчет твиттера? Нет проблем, вот полноценный бот, который на входе принимает команду, выполняет ее с помощью командного интерпретатора и отправляет результат указанному юзеру.

```
#!/bin/bash
USER="ТВОЙ_НИК"
while true; do
  CMD=`echo "/dma +1" | ttytter -script | sed 's/\[.*\]\ //'
  if [ $CMD != $OLD_CMD ]; then
    REPL=`$CMD`
    echo "/dm $USER ${REPL:0:140}" | ttytter -script
    CMD = $OLD_COMD
  fi
  sleep 60
done
```

Скрипт использует консольный клиент ttytter, читая в цикле последнее direct message, далее он проверяет, не была ли такая команда уже выполнена,

и, если нет, выполняет ее и отправляет указанному в переменной USER пользователю, попутно обрезая до 140 символов.

Чтобы все заработало как надо, тебе придется установить ttytter, запустить его, ввести приведенную им ссылку в адресную строку браузера, скопировать показанный браузером ключ аутентификации и ввести его в ttytter. Естественно, перед тем как это сделать, следует завести для бота отдельную учетку.

```
Request from https://api.twitter.com/oauth/request_token ... SUCCEEDED!

1. Visit, in your browser, ALL ON ONE LINE,

https://api.twitter.com/oauth/authorize?oauth_token=R1nZqgAAAAAAAAB8AAABVqGds4w

2. If you are not already signed in, fill in your username and password.

3. Verify that TTYtter is the requesting application, and that its permissions
are as you expect (read your timeline, see who you follow and follow new
people, update your profile, post tweets on your behalf and access your
direct messages). IF THIS IS NOT CORRECT, PRESS CTRL-C NOW!

4. Click Authorize app.

5. A PIN will appear. Enter it below.

Enter PIN>
```

Twitter запрашивает ключ

Твиттер можно использовать не только для выполнения команд, но и для мониторинга машины. Следующий скрипт отправляет в ленту сообщение с информацией о состоянии машины (имя хоста, uptime, нагрузка, свободная память и нагрузка на CPU):

```
#!/bin/bash

HOST=`hostname -s`
UP=`uptime | cut -d" " -f4,5 | cut -d"," -f1`
LOAD=`uptime | cut -d":" -f5,6`
MEM=`ps aux | awk '{ sum += $4 }; END { print sum }'`
CPU=`ps aux | awk '{ sum += $3 }; END { print sum }'`

tweet="Host: ${HOST}, uptime: ${UP}, cpu: ${CPU}%, memory: ${MEM}%,
loadavg ${LOAD}"

if [ $(echo "${tweet}" | wc -c) -gt 140 ]; then
   echo "FATAL: The tweet is longer than 140 characters!"
   exit 1
fi
```

```
echo $tweet | ttytter -script
```

Ну и под конец приведем скрипт, не связанный с сетевыми сервисами, но имеющий прямое отношение к сетям и к тому, зачем мы обычно их используем. Это скрипт для запуска и остановки торрент-клиента во время простоя машины:

```
#!/bin/bash

IDLE=600000
STOPCMD="transmission-remote -S"
STARTCMD="transmission-remote -s"

STOPPED="yes"
while true; do
  if [ `xprintidle` -gt $IDLE ]; then
    if [ $STOPPED = "yes" ]; then
      $STARTCMD
      STOPPED="no"
    fi
  else
    if [ $STOPPED = "no" ]; then
      $STOPCMD
      STOPPED="yes"
    fi
  fi
  sleep 60
done
```

Скрипт уходит в бесконечный цикл, каждую минуту проверяя, сколько миллисекунд прошло с момента, когда юзер что-либо делал (для этого используется команда xprintidle). Если прошло уже 600 000 мс (десять минут), скрипт выполняет команду, указанную в переменной STARTCMD. В противном случае он выполнит команду STOPCMD, но только тогда, когда до нее была выполнена команда STARTCMD. Если кратко: ничего не делаешь за компом десять минут — запускается STARTCMD, в данном случае это команда запуска всех закачек с помощью Transmission, если нет — приостановка всех закачек. Не любишь Transmission? Нет проблем, вот команды для Deluge:

```
STOPCMD="deluge-console pause \*"
STARTCMD="deluge-console resume \*"
```

O ShellCheck

ShellCheck — это инструмент статического анализа, который отображает предупреждения и предложения о неправильном коде в сценариях оболочки bash / sh.

Его можно использовать несколькими способами: из Интернета путем вставки сценария оболочки в онлайн-редактор или, альтернативно, это может быть установка его в вашей системе и запуск из терминала, интеграция с вашим текстовым редактором, а также в ваша подборка или комплекты доказательств.

ShellCheck в первую очередь выполняет три задачи:

- Указывает и объясняет типичные синтаксические проблемы новичкам, которые заставляют оболочку выдавать загадочные сообщения об ошибках.
- Указывает и объясняет типичные семантические проблемы среднего уровня, из-за которых оболочка ведет себя странно.
- Он также указывает на тонкие предупреждения, которые могут привести к сбою работающего расширенного сценария в будущем.

Линтер — это чистый код для ленивых

Многие из проблем скриптов можно решить с помощью линтера, такого, как статический анализатор кода ShellCheck. Он помогает искать ошибки в текстах скриптов и выводить отчёты о проведённых проверках. Это позволяет повысить производительность работы и качество кода.

Линтер — это программа, которая автоматизирует все процессы, и сама «причёсывает» код по определённым правилам. Даёшь ей неуклюжий код, она чистит на уровне каких-то простых правил.

В чём может быть проблема, когда программистов много

Когда вы пишете код самостоятельно и для себя, его можно писать как угодно — в одну строчку без пробелов или отделяя функции друг от друга тремя пустыми строками. Главное, что вы понимаете, что здесь написано.

Как только вы начинаете участвовать с кем-то в совместном проекте, нужно установить общие правила написания кода. Сюда может относиться:

- как форматировать код;
- ставить табы или пробелы;
- если пробелы — сколько штук;
- можно ли использовать числа в формулах или нужны только константы и переменные;
- можно ли использовать переменные из одной буквы или имя переменной обязательно должно быть со смыслом;
- как объявлять классы и работать с методами и т. д.

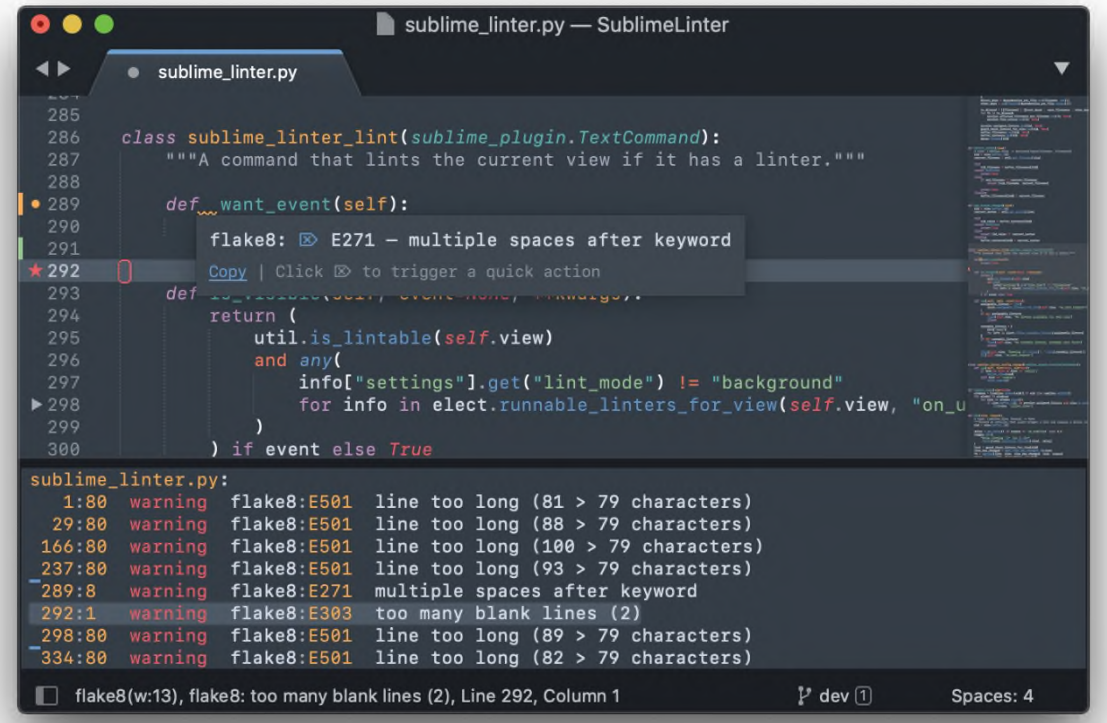
Чтобы эти правила применялись и работали, за ними кто-то должен следить. Можно выделить отдельного программиста, можно проверять код друг друга по очереди, а можно поручить это линтеру.

Что делает линтер

Линтер на основании заданных правил перерабатывает код за секунду:

- расставляет нужные табы и пробелы;
- проставляет точки с запятой и скобки в нужных местах;
- в HTML делает красивую вложенную структуру тегов и выносит стили из тегов в отдельный блок в <style>;
- в других языках может проставить большие буквы в названиях функций, чтобы было удобнее читать;
- убирает лишние пробелы и пустые строки.

Линтер проверил код и нашёл лишние пробелы и много слишком длинных строк Линтер проверил код и нашёл лишние пробелы и много слишком длинных строк.

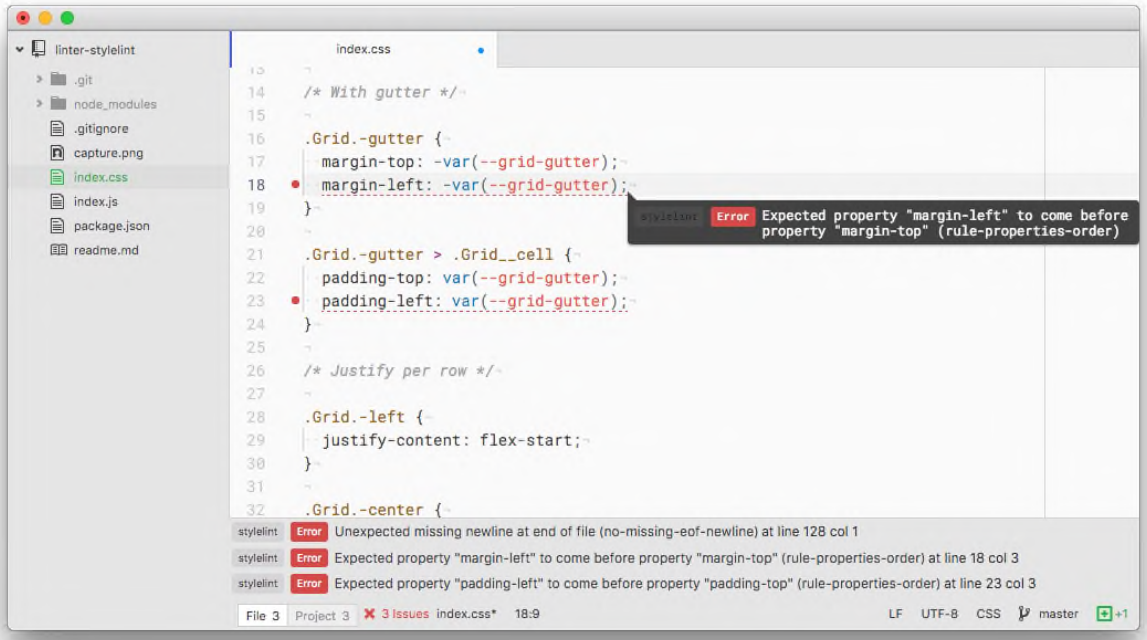


Умные линтеры

Есть линтеры попроще и поумнее. Умные могут не только делать код красивым внешне, но и проверять логику работы программы. Например, линтер может найти все переменные, которые объявлены, занимают память, но ни разу не используются. Или он может заменить одни стандартные функции на другие, если в компании есть такие правила.

Отдельные умники умеют находить потенциальные утечки памяти, циклические зависимости, неполные вызовы и коряво объявленные классы. А ещё — проверять соответствия типов, связность классов, количественные параметры вызова функции и передачи аргументов, цикломатическую сложность и так далее. Но если вам нужны все эти проверки, то вы уже и так знаете про то, как устроены линтеры и для чего они нужны.

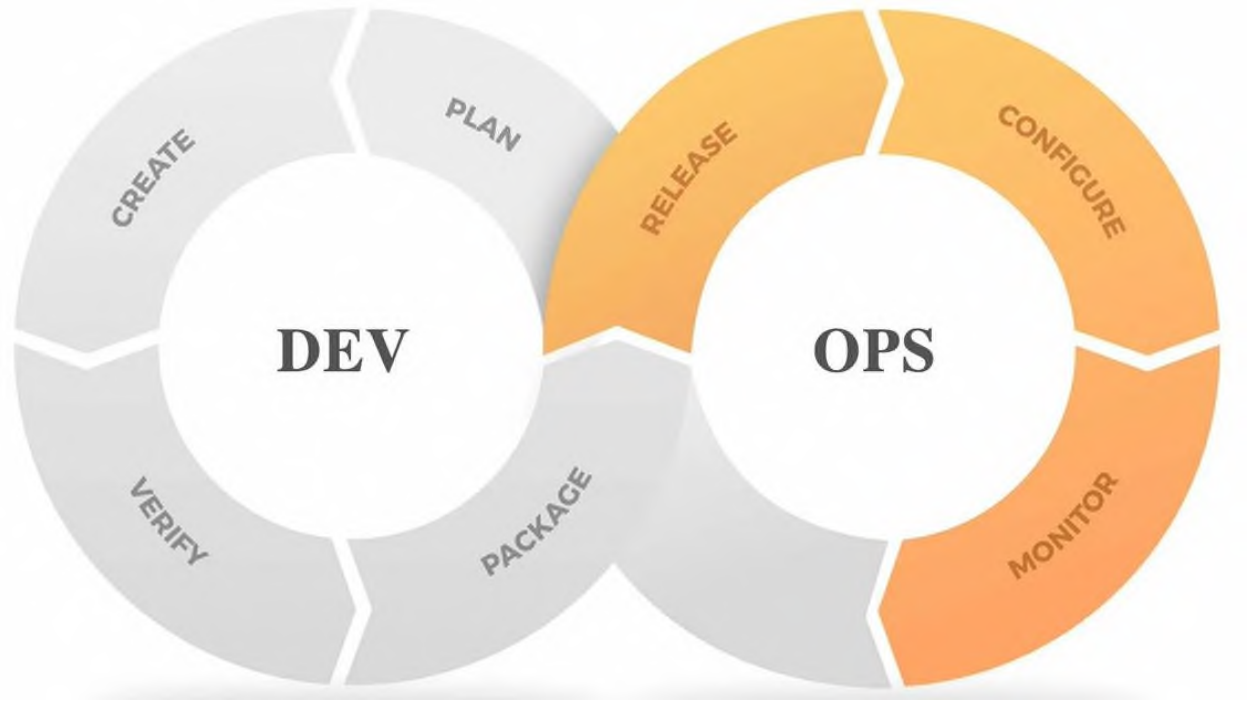
Линтер нашёл стилистическую ошибку в CSSЛинтер нашёл стилистическую ошибку в CSS — нужно поменять эти команды местами, потому что в компании принято сначала описывать верхние отступы, а потом уже боковые.



Один линтер — один язык

Есть много разных линтеров для каждого языка программирования. Они отличаются по глубине настройки и возможностям анализа кода — как глубоко они могут его анализировать. Например, для JavaScript есть JSLint, JSHint и ESlint — это всё линтеры, которые делают плюс-минус одно и то же и различаются в деталях.

Иногда программисты используют несколько линтеров — один смотрит синтаксис, второй следит за памятью и безопасностью, а в третьем мощная система правил автозамены кода.



МОДУЛЬ 4. Системы хранения данных: типы и особенности

1. Теория жёстких дисков (что такое IOPS)
2. Контроллеры и дисковые массивы
3. Кеширование Redis/memcached
4. Протоколы СХД
5. Типы СХД
6. Достоинства и недостатки типов СХД

1. Теория жёстких дисков (что такое IOPS)

Что такое жесткий диск HDD

Жесткий диск (HDD, hard disk, магнитный диск) — это устройство для хранения данных, в котором используются магнитные пластины для записи информации. Применяется в большинстве настольных компьютеров и ноутбуках в качестве основного накопителя.

На сленге часто называется — винчестер, так его прозвали в 1973 году из-за сходства с винтовкой Winchester. Все потому, что у модели HDD 3340 было 2 модуля по 30Мб, а у винтовки патроны назывались 30–30 Winchester. Эта «удивительная» схожесть и породила такое название. Хотя в России чаще его просто называют — винт.



Память на таком устройстве не энергозависима, как, например, у оперативной памяти, это означает, что данные не будут стираться при отключении питания.

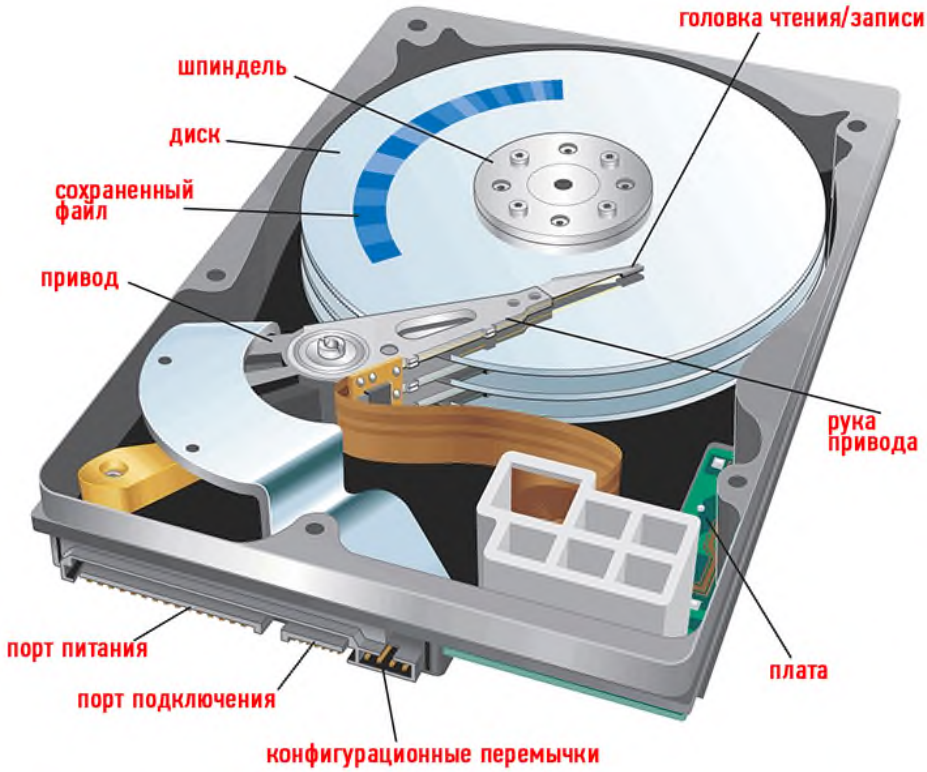
На компьютере или ноутбуке просто необходимо, чтобы был установлен какой-либо накопитель информации, это может быть, как раз жесткий диск, или твердотельный накопитель. Такой накопитель выполняет следующие функции:

- Хранит на себе всю операционную систему, ее файлы и настройки
- Все файлы, пользователи, музыка, фото, видеоролики, документы и т. д.
- Используется для хранения временных файлов самой системой, чтобы разгрузить оперативную память

Конструкция

Конструкция HDD выглядит одновременно простой и довольно сложной. Собирается он из следующих основных компонентов:

- Корпус — защищает от пыли, влаги
- Магнитные пластины — на них, как раз и производится запись информации
- Головки записи/чтения — они считывают информацию с магнитных пластин
- Двигатель — он вращает магнитные пластины
- Контроллер — управляет работой, по сути, это обычная микросхема.



В конструкции есть и другие элементы, но эти основные из них.

Как работает жесткий диск

По сути, жесткие диски записывают и считывают информацию, примерно также, как происходить запись и чтение на виниловых пластинках. Т. е. не углубляясь в множество терминов. Головки чтения и записи информации при помощи магнитных импульсов записывают и считывают данные с магнитных пластин.

Записанная информация хранится на секторах, которые для большего удобства объединяются в кластеры. Выглядит это примерно, как разрезанная пицца. Запись информации происходит в непрерывной последовательности кластеров, т. е. головка для записи/чтения двигается по пластине без лишних сдвигов ровно по кластерам.



Фаргментированный диск Дефрагментированный диск

Это дает возможность быстрого доступа к записанной информации. Но, к сожалению, при удалении, какой-либо информации кластера пустеют и происходит фрагментация файлов подробнее об этом в материале — что такое дефрагментация диска.

Виды жестких дисков

Магнитные диски делают много разных производителей, основные из них это: Seagate, Western Digital и Toshiba. Делят их на следующие виды:

- 3.5 дюймов — используют HDD такого размера обычно в персональных компьютерах и на серверах
- 2.5 дюймов — в таком варианте можно чаще всего встретить в ноутбуках, но и на ПК тоже ставят
- Внешние HDD — работают по USB, можно подключить к любому устройству, например, к телевизору, что реально удобно.



3.5 HDD 2.5 HDD USB HDD

По другим параметрам будем делить их по характеристикам.

Хара*ктеристики жестких дисков*

Основные характеристики Hard Disk это:

- Объем памяти — для домашнего ПК рекомендуются на данный момент модели от 1 Тб т. к. игры, кино и другой медиа контент будет занимать довольно много места.
- Форм фактор — для ПК стоит использовать или 3.5 или 2.5, а для ноутбука 2.5. Если нужен, например, для телевизора — используйте внешний.
- Скорость работы — использовать только от 7200 об/мин, остальное будет очень медленно, я про 5400 об/мин, особенно по сравнению с SSD. Лучше если найдете вариант с 10000 об/мин.
- Интерфейс подключения — на данный момент в материнских платах для HDD предусмотрен интерфейс SATA, раньше использовался IDE. Внешние же — подключаются просто по USB.
- Время произвольного доступа — промежуток в течение, которого жесткий гарантированно выполнит запись или чтение. Обычно это от 3 до 15 миллисекунд.

Также их можно еще разделить на: уровень издаваемого шума, защищенность от физического воздействия, надежность, на сколько циклов рассчитан.

Преимущества HDD дисков

Самое главное преимущество HDD перед SSD — это их долговечность в плане использования. Практически неограниченный цикл записи/чтения. Это идеальный вариант для хранения информации в виде: видео, фото, музыки. Если вы часто смотрите кино с телевизора, то рекомендую купить именно внешний жесткий диск. Разницы в скорости вы все равно не заметите, а прослужит он вам реально намного больше.



Минусы HDD

Одним из самых главных минусов по сравнению с новыми SSD является — скорость работы. Да они действительно работают намного медленнее.

Поэтому для операционной системы лучше брать именно ССД, а вот уже для хранения файлов — жесткий диск.

Еще один минус, это то, что они сильно подвержены механическому воздействию, такие накопители лучше не ронять. А если берете внешний, то лучше ищите модель, защищенную от падений и ударов, такой Travel вариант.



Что такое IOPS

IOPS — количество операций ввода-вывода в секунду. Одна из основных характеристик для оценки производительности проектируемой или существующей системы хранения данных, RAID-массива, HDD или SSD диска. Другими словами, это количество блоков, которое успевает записаться или считаться с устройства в единицу времени. Чем больше IOPS, тем более производительная система.

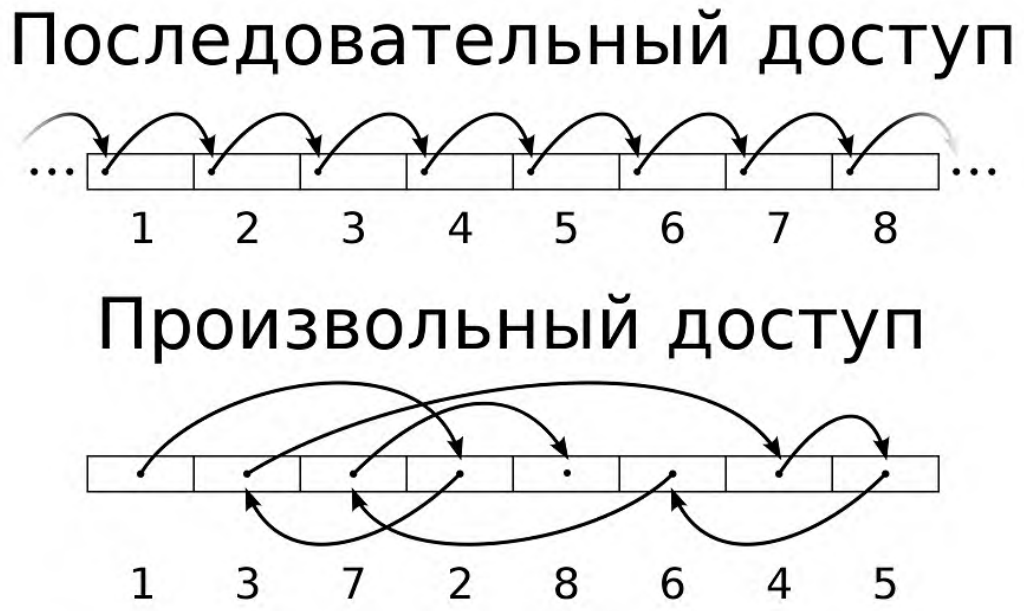
IOPS — понятие расплывчатое, указанные производителем значения могут на практике отличаться в разы. Дело в том, что величина IOPS зависит от многих факторов:

- Конструкция диска.
- Настройка диска.
- Конструкция массива.
- Настройка массива.
- Настройка файловой системы.
- Драйвера.
- Конфигурация и тип RAID-массива.
- Количество дисков в массиве.
- Наличие и размер кеша.
- Гибридные диски.
- Качество диска.
- Типы интерфейсов.
- Количество интерфейсов.
- Пропускная способность интерфейсов.
- Размер блока.

- RAID-контроллер.
- Условия запуска программы, которая измеряет IOPS.
- Наличие других фоновых задач.
- Размеры очередей и буферов.

Принято выделять несколько типов IOPS:

1. IOPS последовательного чтения.
2. IOPS последовательной записи.
3. IOPS произвольного чтения.
4. IOPS произвольной записи.
5. TOTAL IOPS — суммарное значение IOPS чтения и записи (отличается для последовательного и произвольного доступа).



IOPS для HDD

Для жёстких дисков величина IOPS более или менее постоянная.

Обороты	**Интерфейс**	**IOPS**
7200 об/мин	SATA 3	~75-100
10000 об/мин	SATA 3	~125-150
10000 об/мин	SAS	~140
15000 об/мин	SAS	~175-200

IOPS для SSD и гибридных дисков

Зависит от модели и интерфейса. PCIe на порядок быстрее чем SATA.

Вычисление IOPS для RAID массива

Если мы знаем IOPS диска, то можем вычислить IOPS RAID массива.

RAID пенальти. При операции чтения данных из любого массива происходит одна операция, собственно операция чтения, поэтому на чтение

RAID пенальти отсутствует. При записи ситуация иная для каждого типа массива.

Тип массива	Пенальти	Операции
RAID0	1	запись данных
RAID1	2	запись данных + запись данных
RAID10	2	запись данных + запись данных
RAID5	4	чтение данных + чтение чётности + запись данных + запись чётности
RAID6	6	чтение данных + чтение чётности * 2 + запись данных + запись чётности * 2

$$T = (I * N * W)/P + I * N * R$$

Где T — вычисляемое значение IOPS массива, I — IOPS диска, N — количество дисков, P — пенальти, W — процент записи, R — процент чтения.

2. Контроллеры и дисковые массивы

RAID (Redundant Array of Independent Disks или «избыточный массив независимых дисков») — метод виртуализации, позволяющий объединять несколько дисков в единый логический том, имеющий лучшие характеристики. Чтобы описать, чем RAID может быть полезен на практике, рассмотрим теоретические основы, классификацию и особенности использования данной технологии.

Для чего применяется RAID

RAID позволяет превратить несколько дисковых накопителей в один большой и быстрый диск. Его можно использовать в качестве хранилища данных с функцией автоматического резервного копирования или настроить как системный диск повышенной отказоустойчивости.

У технологии RAID-массивов существуют и минусы. Платой за быстродействие и надежность становится усложнение системы, а также необходимость закупать дополнительное оборудование. Однако эта цена невелика по сравнению с потенциальными убытками, которые может понести пользователь при потере информации или внезапной поломке накопителя.

Преимущества технологии

1. **Увеличенный объем**. Первоначальное назначение RAID — получение диска большей емкости.

2. **Повышение быстродействия** системы через параллельное подключение в массив нескольких физических дисков.

3. **Отказоустойчивость** и **надежность хранения данных** обеспечиваются выделением на цели резервирования отдельного устройства. При повреждении одного из дисков RAID-массива информация не будет утеряна.

Условие применения

Технологию можно использовать не во всех случаях. Для этого требуется ее аппаратная и программная поддержка. BIOS должен содержать настройку вида «**SATA Configuration**: **RAID**». Если же ее по каким-либо причинам нет, то необходимо «перепрошить» базовую систему ввода-вывода.

В случае, когда поддержка RAID программным методом невозможна, нужно подключить дополнительное устройство — RAID-контроллер и установить соответствующий драйвер. В последних версиях ОС Linux (Ubuntu 20.04, POP-OS 20.04 и т. д.) драйвер для включения режима RAID инсталлируется автоматически.

Основные понятия

В основе функционирования RAID-массивов лежит несколько базовых терминов, без которых нельзя понять принципы работы этой технологии.

1. **Массив** — объединение нескольких физических или виртуальных накопителей в один большой диск с возможностью единой настройки, форматирования и управления.

2. **Метод зеркалирования** — способ повысить надежность хранения информации через создание копии исходного диска на другом носителе, входящем в массив.

3. **Дуплекс** — один из методов зеркалирования, в котором используется вдвое большее количество накопителей для создания копий.

4. **Чередование** — увеличение производительности диска, благодаря блочной разбивке данных при записи.

5. **Четность** — технология, сочетающая в себе чередование и зеркалирование.

RAID-контроллер

На плечах RAID-контроллеров лежит ответственная задача — управление дисковой подсистемой, то есть всей информацией, хранимой на сервере. Именно они отвечают за работу дисковых массивов, позволяя повысить производительность сервера или надёжность хранения данных.

Поэтому давайте поговорим о RAID-контроллерах, установленных в серверы вендоров «большой тройки», об их возможностях и особенностях.

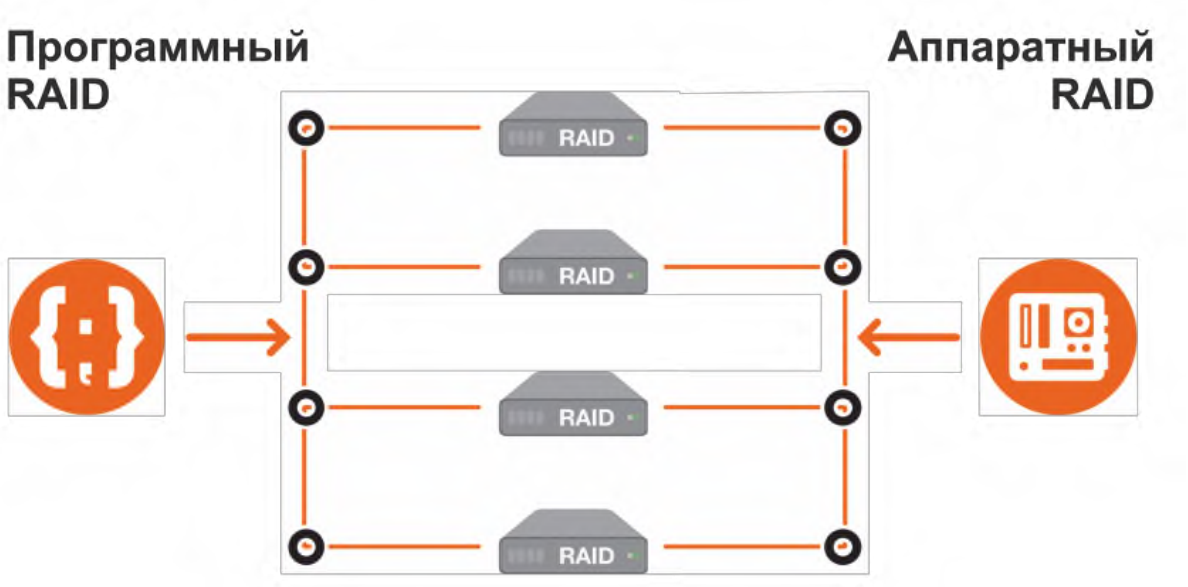
Что такое RAID-контроллер?

Чаще всего задачи, выполняемые серверами, требуют высокой скорости чтения/записи данных и/или необходимость сохранить данные при выходе из строя самих накопителей. Поэтому установка в сервер единственного диска редко имеет смысл. Этот вариант можно рассматривать, если нагрузка будет совсем небольшой, а сохранность данных не волнует вовсе. Да и объёмы информации, которыми оперируют серверы, часто требуют куда больше пространства для хранения, чем может дать один диск. А чем больше накопителей, тем выше вероятность выхода из строя, особенно при высокой нагрузке.

Проблемы производительности и отказоустойчивости дисковой подсистемы решаются с помощью создания массивов: логических структур, в которые с помощью RAID-контроллера объединяется несколько накопителей — жёстких дисков и SSD. При этом массив выглядит для системы единым пространством для хранения данных.

Существует много видов массивов, отличающихся производительностью, надёжностью хранения данных и минимально необходимым количеством дисков. Выбор конкретного вида зависит от ваших задач и потребностей, а также от возможностей самого RAID-контроллера.

RAID-контроллеры делятся на:



Программные. Вся нагрузка по управлению массивом ложится на центральный процессор. Наименее производительное и отказоустойчивое решение.

Интегрированные. Встроены в материнскую плату. Отдельный чип выполняет часть задач по управлению, но всё же тоже задействует центральный процессор. Интегрированные контроллеры могут иметь собственную кэш-память. По сравнению с программными, поддерживают больше видов массивов, работают куда быстрее и надёжнее.

Аппаратные. Выполнены в виде плат расширения или отдельных устройств, размещаемых вне сервера (внешние, или мостовые контроллеры). Оснащены собственным процессором, выполняющим все необходимые вычисления, и, как правило, кэш-памятью. Модульные контроллеры могут иметь внешние и внутренние порты:

Внутренние — предназначены для подключения накопителей, установленных в сам сервер.

Внешние — используются для подключения внешних дисковых хранилищ.

Даже если ваш сервер оснащён интегрированным RAID-контроллером, то при необходимости можно будет установить и модульный, если имеющихся возможностей, скорости и надёжности вам будет недостаточно.

Если на борту RAID-контроллера есть кэш-память, то она может использоваться для промежуточного хранения записываемых или считываемых данных. Это позволяет эффективнее управлять операциями ввода/вывода.

Чтобы при сбое питания не потерять данные, находящиеся в кэше, используется два разных подхода:

- контроллер оснащается собственной батарейкой (BBU — Battery Backup Unit), позволяющей хранить данные в памяти до 3 суток,
- либо дополнительной флэш-памятью, питаемой от ёмкого конденсатора. При сбое питания в неё выгружает содержимое кэша. А поскольку флэш-память потребляет очень мало энергии, то и данные в ней сохраняются месяцами. Обратите внимание, что флэш-память используется только при сбое питания.

И как только сервер заработает, контроллер первым делом скидывает содержимое кэша на диск. При наличии батарейки можно активировать режим WriteBack: при записи на диск контроллер сообщит об успешном выполнении операции уже в тот момент, когда данные попадут в кэш, а затем

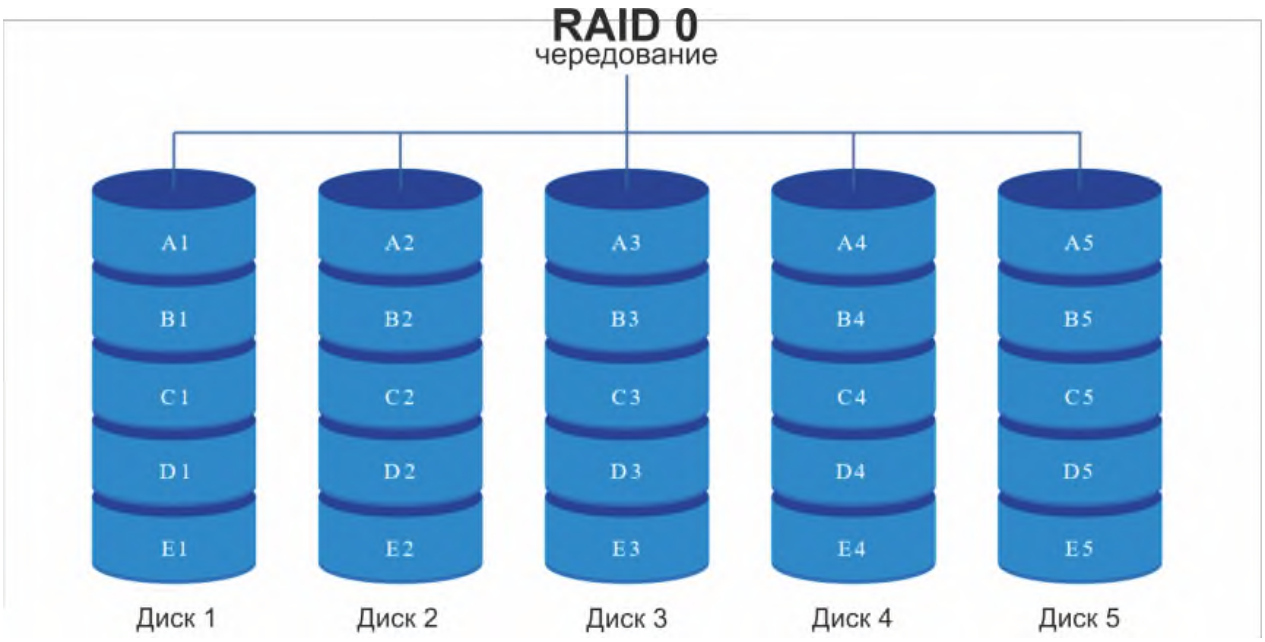
в «фоновом» режиме скинет их на диски. Поэтому другие процессы проведут меньше времени в ожидании подтверждения.

Некоторые RAID-контроллеры позволяют увеличить объём кэш-памяти и установить батарейку, если они её не имеют. Чем больше размер кэша контроллера, тем выше производительность RAID-массивов.

Классификация RAID по уровням

Основные отличия между конфигурациями или уровнями RAID заключаются в методах формирования и размещения данных, а также в алгоритмах распределения информации на носителях. Базовые типы RAID-массивов — RAID 0 и RAID 1. Остальные уровни считаются их производными, сочетающими в себе достоинства той или иной базовой модели.

RAID 0



Технология виртуализации RAID 0 называется striping («чередование»). Для ее реализации применяется от 2 до 4 накопителей, которые совместно выполняют процедуру «чтения/записи».

При записи информация разделяется на блоки, которые одновременно сохраняются на накопители. Первый блок — на один, второй — на другой жесткий диск и так далее. Производительность массива возрастает прямо пропорционально количеству накопителей в системе. То есть, 4 диска будут работать в 2 раза быстрее, чем два.

Однако, такая конфигурация RAID-массива чревата потерей данных, что уменьшает безопасность хранения информации. Это объясняется структурой каждого файла. Последний состоит из определенной последовательности

блоков (байт), поскольку каждый из них записывается на разные диски и происходит «нарушение» его целостности. Если один накопитель выходит из строя, то блок «теряется». При этом получается «битый» файл, который практически невозможно восстановить.

Достоинства

- Дисковый RAID-массив уровня 0 обеспечивает ощутимый прирост скорости, который прямо пропорционально зависит от кратности количества накопителей.
- Использование всего дискового объема, т. е. при установке четырех дисков по 2 ТБ общий объем RAID-массива будет равен 2*4=8 ТБ.

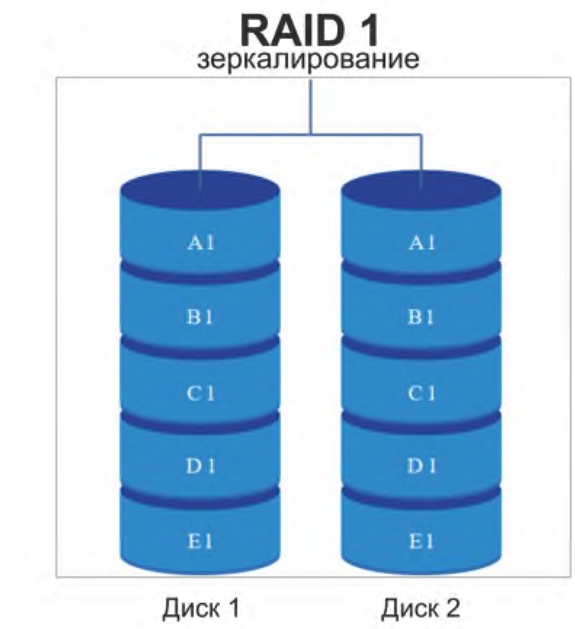
Недостатки

- Нарушение отказоустойчивости. Иногда возможен отказ в операциях чтения или записи.
- При выходе из строя одного накопителя информация полностью теряется.

Использование

Применяется в приложениях для скоростного обмена информацией, в хранилищах временных файлов. Также RAID 0 нужен для систем, использующих некритичные по важности массивы данных.

RAID 1



Технология RAID 1 называется mirroring («зеркалирование»). Она подразумевает использование от 2 до 4 накопителей. Однако при этом теряется половина объема дисков, поскольку это пространство используется резервированием данных.

Простыми словами, если RAID-система состоит из 2 жестких дисков, то при выходе одного из них информация не потеряется полностью, поскольку один накопитель является точной копией другого.

Достоинства

- Надежность хранения информации.
- Простота реализации.
- Высокая производительность при выполнении операции чтения.
- Минимальная комплектация составляет всего 2 жестких диска.

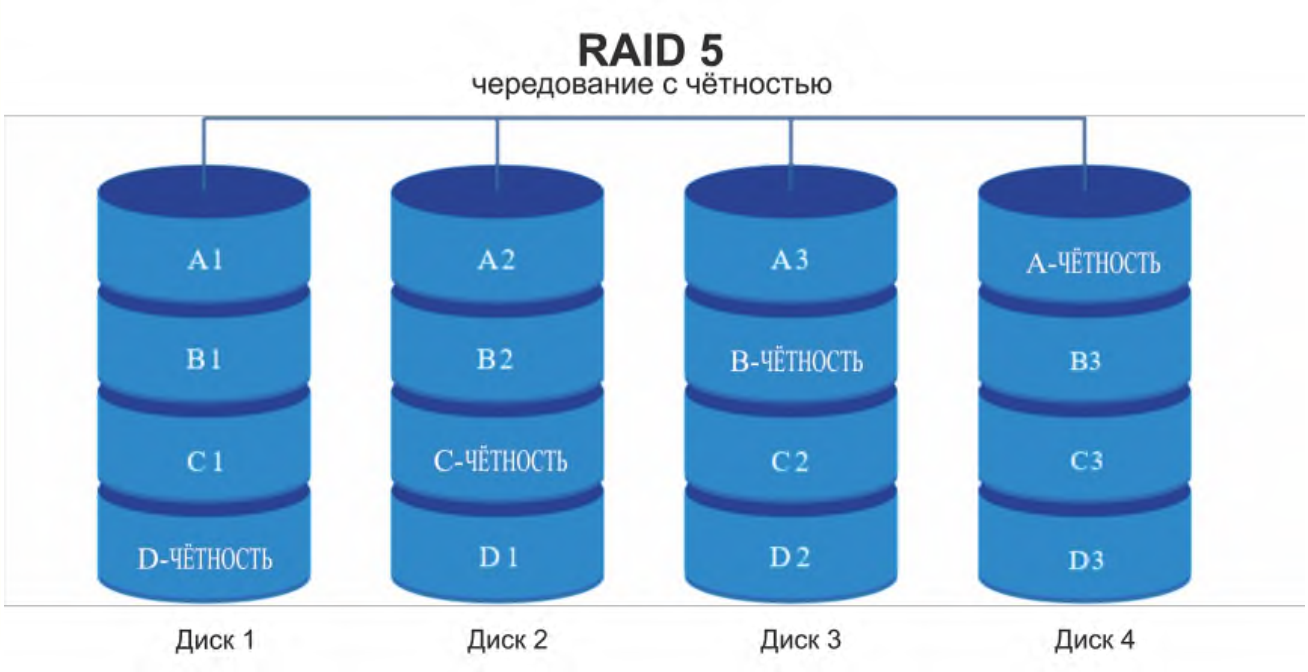
Недостатки

- Низкая производительность.
- Емкость RAID-массива делится на 2, что обусловлено резервированием информации.
- Замена неисправного накопителя требует полное отключение системы.

Использование

Уровень RAID 1 необходимо применять для увеличения надежности хранения информации на серверах.

RAID 5



Технология RAID 5 («чередование с чётностью») считается наиболее распространенной и безопасной. Для подобной конфигурации необходимо минимум 3 диска, а максимальное допустимое количество — 16.

При записи информации происходит разделение на блоки данных, но с одним условием — на один из дисков, называемый блок «чётность данных» (**Parity Drive**, **PD**), происходит запись информации для восстановления. Этот подход позволяет спасти данные при повреждении одного из накопителей.

RAID 5 может реализовываться программным методом при помощи специальных утилит, но IT-специалисты рекомендуют все же отдать предпочтение аппаратному способу.

Достоинства

- Увеличена скорость чтения за счет одновременной обработки данных с нескольких независимых потоков от дисков массива.
- Информация не «потеряется» при повреждении одного накопителя.
- При замене неисправного диска происходит автоматическое восстановление информации.

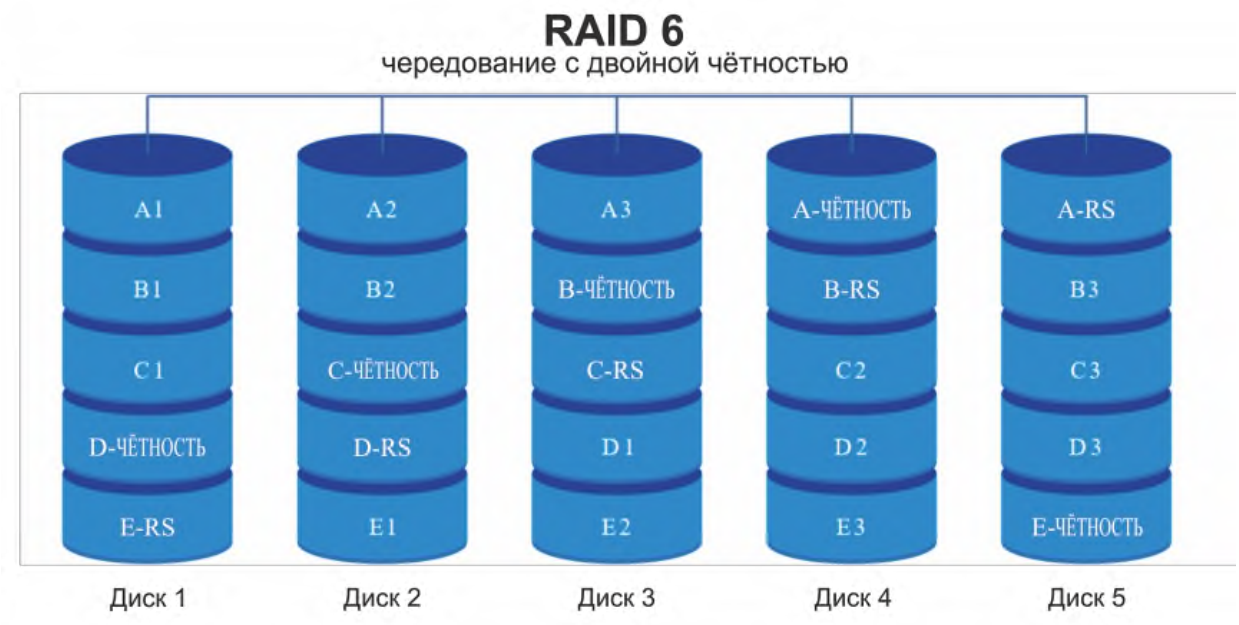
Недостатки

1. Иногда происходят отказы дисков.
2. Если объем поврежденного накопителя 4 ТБ и более, при замене его на идентичный диск, восстановление может занять более одного дня.
3. Если диск «чётности» вышел из строя при выполнении процедуры восстановления, то информация будет окончательно утеряна.
4. Минимальное количество накопителей — 3.

Использование

Технология виртуализации 5 уровня (RAID 5) прекрасно подойдет для безопасного хранения данных, но при этом не будет утрачена производительность. Очень часто ее используют файловые серверы.

RAID 6



Технология виртуализации 6 уровня («чередование с двойной чётностью») похожа на RAID 5. Отличие состоит в записи информации для

восстановления на два диска. Первый — блок «чётность данных» (**PD**) используются в архитектуре RAID 5 для резервного хранения данных. Второй диск «чётности» дублирует работу первого. Его работа основана на коде Рида-Соломона (Reed-Solomon), поэтому диск часто имеет краткое обозначение — **RS** или **Q**.

Благодаря использованию принципа двойной чётности система может перенести без потерь информации отказ сразу двух жестких дисков. Однако для создания RAID 6 потребуется минимум четыре накопителя.

Достоинства

- Высокая скорость считывания и записи данных.
- Поддержка двух, одновременно вышедших из строя накопителей.

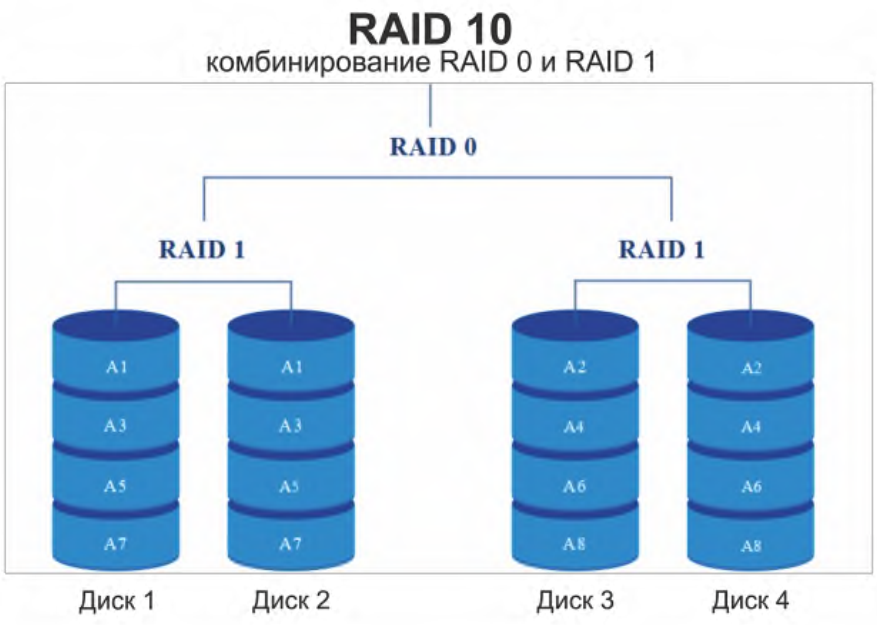
Недостатки

- Время на операцию записи на 20% больше, чем для RAID 5.
- Минимальная вероятность отказа дисков.
- Восстановление после сбоя занимает много времени.
- Для реализации необходимо 4 накопителя.

Использование

RAID 6 является более надежной конфигурацией, чем RAID пятого уровня. Она часто применяется на файловых серверах, где используются большие объемы данных.

RAID 10



Технология виртуализации 10 — «гибрид» RAID нулевого и первого уровней, сочетающая в себе все их преимущества.

Достоинства

- Высокая скорость восстановления данных.

- Высокая надежность.
- Быстродействие.

Недостатки

- Дороговизна реализации.
- Емкость, уходящая на зеркалирование, эквивалентна 50 % от всего объема дисков.

Использование

Гибридная технология RAID 10 используется в тех же случаях, что и RAID 0 и RAID 1.

Утилиты для создания

В операционной системе Windows есть встроенная утилита для создания RAID. Однако она поддерживает только RAID-массивы первого. Поэтому для более сложных операций, а также для платформ на базе Unix/Linux требуется установка стороннего ПО.

Перед выбором соответствующей конфигурации RAID-массива специалисты рекомендуют сохранить информацию на отдельный носитель. При создании или удалении RAID-системы данные на дисках уничтожаются.

Mdadm

Для операционных систем на основе Linux рекомендуется использовать штатную утилиту «mdadm», которую необходимо предварительно установить через терминал.

Основные возможности

- Создание и сброс RAID-массивов.
- Монтирование файловых систем.
- Сохранение топологии массива.
- Удаление отдельных элементов из RAID.

3. Кеширование Redis/memcached

Кэширование — инструмент, позволяющий совладать с первыми скачками нагрузки на приложение.

Кэширование — это сохранение данных в высоко доступных местах на временной основе для того, чтобы их можно было получать быстрее, чем из оригинального источника. Самый распространенный пример применения кэша — получение данных из базы. При первом получении, допустим, продукта из базы данных, он сохраняется в кэш на определённое время,

поэтому каждый следующий запрос к этому продукту уже не будет тревожить БД: данные будут получены из другого хранилища.

Memcached иногда более эффективен, но Redis - почти всегда лучший выбор.

Memcached или Redis?

Это вопрос, который почти всегда возникает в любой дискуссии о достижении большей производительности в современных веб-приложениях, управляемых базами данных. Когда производительность должна быть улучшена, кэширование часто является первым шагом такой оптимизации, и Memcached или Redis, как правило, являются первыми претендентами в списке программистов и администраторов.

Эти известные механизмы кэширования имеют ряд сходств, но они также имеют важные отличия. Механизм **Redis**, более новый и универсальный и **почти всегда является лучшим выбором**.

Redis против Memcached для кеширования данных

Начнем со сходств. И Memcached, и Redis служат в качестве размещающихся в оперативной памяти хранилищ данных с ключом, хотя Redis более точно описывается как хранилище структуры данных. Как Memcached, так и Redis относятся к семейству решений для управления данными NoSQL, и оба они основаны на модели данных с ключевыми значениями. Они сохраняют все данные в ОЗУ (памяти), что, конечно же, делает их чрезвычайно полезными в качестве слоя кеширования. С точки зрения производительности два хранилища данных также очень похожи, демонстрируя почти идентичные характеристики (и показатели) в отношении пропускной способности и латентности.

Как Memcached, так и Redis являются зрелыми и чрезвычайно популярными проектами с открытым исходным кодом. Memcached был первоначально разработан Брэдом Фицпатриком в 2003 году на сайте LiveJournal. С тех пор Memcached был переписан на C (исходная реализация была в Perl) и помещена в общественное достояние, где она стала краеугольным камнем современных веб-приложений. Текущая разработка Memcached ориентирована на стабильность и оптимизацию, а не на добавление новых функций.

Redis был создан программистом Салваторе Санфилиппо (Salvatore Sanfilippo) в 2009 году, и он остается ведущим разработчиком проекта сегодня. Redis иногда описывается как «Memcached на стероидах», что неудивительно с учетом того, что части кода Redis были построены в ответ на уроки,

извлеченные из использования Memcached. Redis имеет больше возможностей, чем Memcached, и, таким образом, более мощный и гибкий.

Используемые многими компаниями и в бесчисленных критически важных производственных средах, Memcached и Redis поддерживаются клиентскими библиотеками на всех мыслимых языках программирования и включаются в множество пакетов для разработчиков. Фактически, это редкий веб-стек, который не включает встроенную поддержку для Memcached или Redis.

Почему Memcached и Redis так популярны? Они не только чрезвычайно эффективны, но и относительно просты. Начало работы с Memcached или Redis считается простой работой для разработчика. Требуется всего несколько минут, чтобы настроить и заставить их работать с приложением. Таким образом, небольшая инвестиция времени и усилий может иметь непосредственное и внушительное влияние на производительность - обычно на порядок. Простое решение с огромной выгодой; это так близко к магии, которой вы можете с легкостью воспользоваться.

Когда использовать Memcached

Поскольку Redis новее и имеет больше возможностей, чем Memcached, Redis почти всегда лучший выбор. Тем не менее, Memcached может быть предпочтительнее при кэшировании относительно небольших и статических данных, таких как фрагменты кода HTML. Управление внутренней памятью Memcached, хотя и не столь сложное, как решение Redis, более эффективно в простейших случаях использования, поскольку оно потребляет сравнительно меньше ресурсов памяти для метаданных. Строки (единственный тип данных, поддерживаемый Memcached) идеально подходят для хранения данных, которые только читаются, потому что строки не требуют дальнейшей обработки.

Тем не менее, эффективность управления памятью Memcached быстро уменьшается, когда размер данных является динамическим, после чего память Memcached может стать фрагментированной. Кроме того, большие наборы данных часто включают в себя сериализованные данные, которые всегда требуют большего пространства для хранения. Хотя Memcached эффективно ограничивается хранением данных в его сериализованной форме, структуры данных в Redis могут сохранять любой аспект данных изначально, тем самым уменьшая накладные расходы на сериализацию.

Второй сценарий, в котором Memcached имеет преимущество перед Redis, находится в масштабировании. Поскольку Memcached многопоточен, вы можете легко увеличить его, предоставив ему больше вычислительных

ресурсов, но вы потеряете часть или все кэшированные данные (в зависимости от того, используете ли вы постоянное хеширование). Redis, который в основном однопоточный, может масштабироваться горизонтально посредством кластеризации без потери данных. Кластеризация представляет собой эффективное масштабирующее решение, но его сравнительно сложно создавать и управлять им.

Когда использовать Redis

Вам почти всегда придётся использовать Redis из-за своих структур данных. С Redis в качестве кеша вы получаете богатый функционал (например, возможность тонкой настройки содержимого кеша и долговечность) и бОльшую эффективность в целом. Как только вы используете структуры данных, повышение эффективности становится огромным для конкретных сценариев приложений.

Превосходство Redis проявляется почти во всех аспектах управления кешем. Кэши используют механизм, называемый выселением данных (синоним: вытеснение данных), чтобы освободить место для новых данных, удалив старые данные из памяти. Механизм выселения данных Memcached использует алгоритм Least Recent Used (LRU) и несколько произвольно вытесняет данные, похожие по размеру на новые данные.

Redis, напротив, позволяет осуществлять мелкомасштабный контроль за выселением, позволяя вам выбирать из шести различных политик выселения. Redis также использует более сложные подходы к управлению памятью и выбору кандидата на выселение. Redis поддерживает как пассивное, так и активное выселение, когда данные вытесняются только тогда, когда требуется больше места или применяется упреждающее вытеснение. Memcached, с другой стороны, обеспечивает пассивное выселение.

Redis дает вам большую гибкость в отношении объектов, которые вы можете кэшировать. Хотя Memcached ограничивает имена ключей до 250 байтов и работает только с обычными строками, Redis позволяет имена и значения ключей размером до 512 Мбайт каждый, и они могут быть бинарными. Кроме того, Redis имеет пять основных структур данных на выбор, открывая миру возможности для разработчика приложений посредством интеллектуального кэширования и манипулирования кэшированными данными.

Redis для сохранения данных

Использование структур данных Redis может упростить и оптимизировать несколько задач - не только при кешировании, но даже когда вы хотите, чтобы данные были постоянными и всегда доступны. Например,

вместо хранения объектов в виде сериализованных строк разработчики могут использовать Redis Hash для хранения полей и значений объекта и управлять ими с помощью одного ключа. Redis Hash сохраняет за разработчиками возможность извлекать всю строку, десериализовать ее, обновлять значение, повторно инициализировать объект и заменять всю строку в кеше новым значением для каждого тривиального обновления, что означает более низкое потребление ресурсов и повышенную производительность.

Другие структуры данных, предлагаемые Redis (такие как списки, наборы, сортированные наборы, гиперлогологи, растровые изображения и геопространственные индексы), могут использоваться для реализации еще более сложных сценариев. Сортированные наборы для обработки данных временных рядов и анализа являются еще одним примером структуры данных Redis, которая обеспечивает чрезвычайно низкую сложность и низкое потребление потребляемых ресурсов.

Другим важным преимуществом Redis является то, что хранящиеся в нем данные не являются непрозрачными, поэтому сервер может манипулировать ими напрямую. Значительная доля команд (более 180), доступных в Redis, обращена к операциям обработки данных и логике внедрения в самом хранилище данных с помощью сценариев Lua на стороне сервера. Эти встроенные команды и пользовательские скрипты дают вам гибкость в решении задач обработки данных непосредственно в Redis без переноса данных по сети в другую систему для обработки.

Redis предлагает опциональную и настраиваемую постоянную сохранность данных, предназначенную для загрузки кэша после запланированного отключения или незапланированного сбоя. Хотя мы склонны рассматривать данные в кэшах как изменчивые и временные, сохраняющиеся данные на диске могут быть весьма полезными в сценариях кэширования. Наличие данных кэша, доступных для загрузки сразу после перезапуска, позволяет значительно сократить время «прогрева» кеша и удаляет нагрузку, связанную с повторной платой за пересчитывание содержимого кеша из первичного хранилища данных.

Redis для репликации данных в памяти

Redis также может копировать данные, которыми он управляет. Репликация может использоваться для реализации высокодоступной настройки кеша, которая может противостоять сбоям и обеспечивать бесперебойное обслуживание приложения. Сбой кэша падает лишь незначительно, если не считать неудачи приложений с точки зрения влияния на пользовательский опыт и производительность приложения, поэтому

наличие надежного решения, гарантирующего достоверность и доступность кеша, является главным преимуществом в большинстве случаев.

И последнее, но не менее важное: с точки зрения оперативной видимости Redis обеспечивает множество показателей и множество интроспективных команд, с помощью которых можно мониторить и отслеживать использование кэша и ненормальное поведение. Статистика в реальном времени по каждому аспекту базы данных, отображение всех выполняемых команд, листинг и управление клиентскими соединениями - у Redis есть все это и многое другое.

Когда программисты осознают эффективность Redis и возможности репликации в памяти, они часто используют его как базу данных с первым ответчиком, как правило, для анализа и обработки высокоскоростных данных и предоставления ответов пользователю, в то время как вторичная (часто более медленная) база данных поддерживает исторический отчет о том, что произошло. При использовании таким образом Redis также может быть идеальным для случаев использования в сфере оперативной аналитики.

Redis для анализа данных

Три сценария аналитики сразу приходят в голову. В первом сценарии при использовании чего-то типа Apache Spark**:** для итеративной обработки больших наборов данных вы можете использовать Redis в качестве обслуживающего слоя для данных, предварительно рассчитанных Spark. Во втором случае использование Redis в качестве общего хранилища распределенных данных в памяти может ускорить скорость обработки Spark в 45–100 раз. Наконец, самый распространенный сценарий — это тот, в котором отчеты и аналитика должны настраиваться посредством пользователя, но получение данных из встроенных хранилищ данных пакетов (например, Hadoop или RDBMS) занимает слишком много времени. В этом случае хранилище данных в памяти, такое как Redis, является единственным практичным способом получения микромиллисекундного времени подкачки и времени ответа!

При использовании чрезвычайно больших наборов операционных данных или аналитических нагрузок работа по всему внутри памяти может оказаться неэффективной. Чтобы достичь производительности в миллисекундах при меньших затратах, Redis Labs создала версию Redis, которая работает на комбинации оперативной памяти и флэш-памяти, с возможностью настройки коэффициентов RAM-to-flash. Хотя это открывает несколько новых возможностей для ускорения обработки рабочей нагрузки,

это также дает разработчикам возможность просто запускать свой «кеш на flash».

Программное обеспечение с открытым исходным кодом продолжает предоставлять одни из лучших технологий, доступных сегодня. Когда дело доходит до повышения производительности приложений посредством кэширования, Redis и Memcached являются наиболее известными и проверенными на практических проектах кандидатами. Однако, учитывая богатую функциональность Redis, более продвинутый дизайн, много потенциальных применений и сравнительно большую экономическую эффективность, Redis должен быть вашим безоговорочным выбором практически в каждом случае.

4. Протоколы СХД

В традиционной схеме организации локальных сетей, которую используют многие современные предприятия, информация обычно записывается на жесткие диски, подключенные напрямую к серверам через соответствующие интерфейсы. Архитектура SAN существенно увеличивает быстродействие и производительность ЛВС, так как позволяет любому компьютеру получить доступ к любому хранилищу, не нагружая рабочие каналы. Кроме того, за счет применения специальной топологии и организации единого пула корпоративных ресурсов значительно повышается эффективность, а также отказоустойчивость системы.

Как хранятся данные в сетях SAN

Доступ к пользовательскому контенту организован следующим образом:

На физическом уровне в качестве устройств хранения используются жесткие диски и (или) твердотельные накопители. Для удобства и повышения надежности их можно объединять в RAID-массивы.

На программном уровне каждой единице оборудования (физической или виртуальной) присваивается уникальный идентификатор (LUN), аналогичный по смыслу буквенному обозначению логического диска.

Серверы могут подключаться к любым логическим дискам, форматировать и разбивать их на разделы, а пользователи получают доступ к высокопроизводительным ресурсам, на которых они могут хранить свои данные, а также считывать, записывать и передавать их далее без каких-либо задержек. Важно отметить, что трафик, передаваемый через SAN, не конкурирует за ресурсы ЛВС с IP-трафиком и не создает помех для работы критически важных приложений, потому что перенаправляется на выделенные каналы.

Пользовательский контент хранится в виде одинаковых по размеру областей — блоков. Таким образом обеспечивается высокая скорость доступа при максимальной степени безопасности и конфиденциальности на всех уровнях архитектуры сети.

Общие тенденции

Существующие сети FC 8Гб/с и Ethernet 10Гб/с уже не справляются с современными СХД, даже Ethernet 25/50 Гб/с не может обеспечить приемлемые задержки при работе с последними моделями СХД, использующими NVMe диски.

Многие ИТ-специалисты, занимающимся настройкой и администрированием инфраструктуры хранения, задаются вопросом о модернизации SAN-сетей. Почему же это так важно и необходимо на сегодняшний день? Сформулируем несколько основных причин:

- Во-первых, плотность хранения данных постоянно растёт, увеличиваются объёмы как HDD дисков, так и флэш-накопителей. Уже сейчас на рынке доступны HDD объёмом 10ТБ и SSD до 3.2ТБ.
- Во-вторых, скорость флэша и гибридных СХД растёт в двух аспектах: увеличивается пропускная способность и уменьшаются задержки. Например, последние модели NVMe-накопителей имеют производительность порядка 3ГБ/с и задержки в 20 мкс.
- В-третьих, потребность в обработке миллионов IOps. Уже сейчас один NVMe-диск способен выдавать до 700+ тысч IOps.
- В-четвёртых, потребность увеличивать производительность на один порт подключения. Если раньше с одного порта QSFP+ на протоколе InfiniBand получали максимум 56Гб/с, то уже в этом году есть решения с производительностью 200Гб/с на порт.

С учетом всех этих факторов ключевыми характеристиками протоколов доступа становятся высокая пропускная способность и быстрое время отклика.

По существующим прогнозам, в перспективе двух лет (по данным Gartner, в течение 18 месяцев) all-flash массивы будут комплектоваться более производительными SSD-дисками, а в сочетании с быстрым интерконнектом (например, NVMe) и новыми протоколами (например, iWARP) производительность all-flash массивов повысится ещё на два порядка, что приведёт к необходимости модернизировать SAN.

Сравнение протоколов

Все SAN сети, будь то Ethernet, PCI Express, SAS, NVMeoF, FC, FCoE или InfiniBand, должны поддерживать одинаковый функционал, а именно:

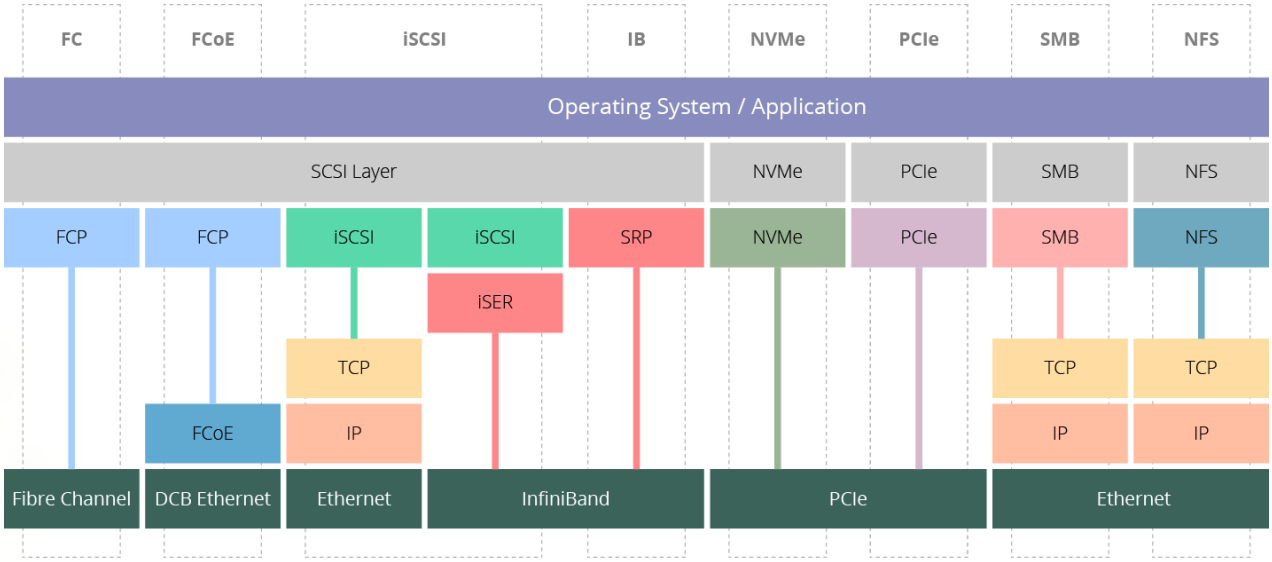
- идентифицировать хост и СХД;
- иметь возможность маршрутизировать траффик;
- разделять сеть на подсети и изолировать в них траффик;
- обеспечивать возможность использования нескольких путей к СХД;
- управлять подключением устройств к сети;
- приоритизировать траффик.

Учитывая схожий реализуемый функционал, можно сказать, что различия между протоколами сводятся к удобству использования, параметрам производительности и издержкам реализации. Так, некоторые

протоколы имеют более развитые средства управления и мониторинга, что делает их более простыми для внедрения, администрирования и эксплуатации. Другие протоколы имеют лучшую производительность, но представляются более сложными в плане интеграции в текущие SAN-сети.

На сегодняшний день протоколы хранения данных можно разделить на две условные группы:

1. используемые для подключения серверов приложений (FC, FCoE, iSCSI, NFS, SMB);
2. используемые для подключения в рамках кластера или в качестве интерконнекта внутри СХД (InfiniBand, NVMe, PCIe).



raidix.ru

Fibre Channel (FC)

Fibre Channel — популярный протокол хранения, обеспечивающий низкие задержки и высокую пропускную способность за счёт своих архитектурных особенностей. Fibre Channel не требователен к ресурсам и отлично подходит для передачи большого объёма данных, так как все операции FC выполняются на стороне HBA, разгружая центральный процессор.

Новые версии протокола Fibre Channel обратно совместимы с прошлыми редакциями, что открывает хорошие перспективы для модернизации и масштабирования. Например, если внедрять FC 32Гб/с, то всё ещё можно будет использовать FC 8Гб/с и 16Гб/с, т. е. можно поэтапно менять FC-коммутаторы и FC адаптеры.

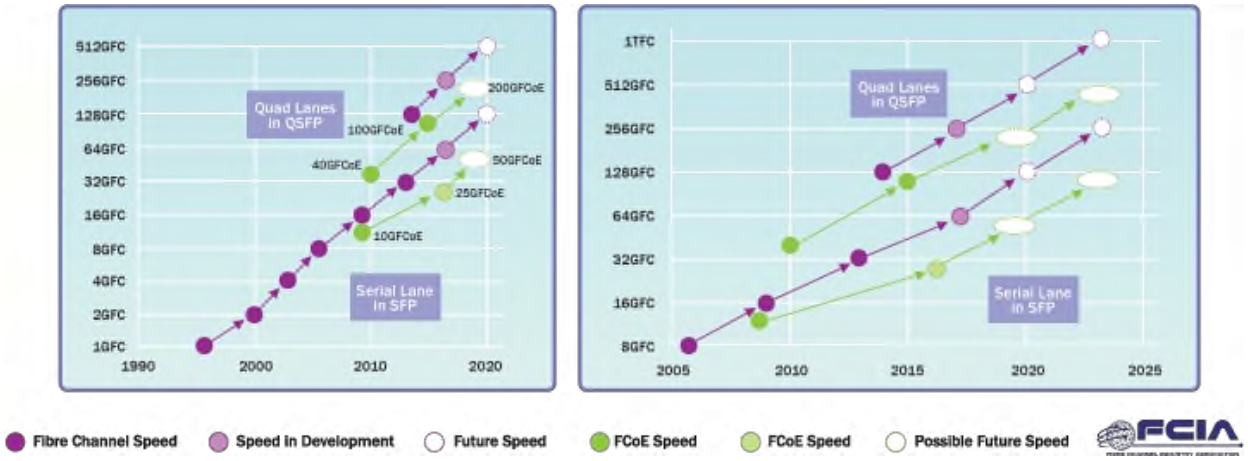
В ближайшее время FC будет обновлён до 64Гб/с и 128Гб/с (уже сейчас есть коммутаторы, поддерживающие агрегацию 4-х портов 32Гб/с в один канал 128Гб/с для соединения коммутаторов).

Простота настройки и удобство в администрировании позволили FC стать одним из наиболее распространенных протоколов хранения. Большинство администраторов SAN-сетей во всем мире знает, как он устроен и какие преимущества обеспечивает при решении различных задач. При этом FC всё ещё сложнее, чем Ethernet, хотя и обладает большим количеством средств управления и мониторинга.

FCoE (Fibre Channel over Ethernet)

Идея создания FCoE заключалась том, чтоб консолидировать операции ввода-вывода и, как следствие, обеспечить безопасное размещение в одном «проводе» различных типов траффика. Такая задумка подразумевала снижение совокупной стоимости владения системой (TCO) за счет уменьшения числа кабелей и адаптеров, а также снижения показателей по энергопотреблению. При этом показатели доступности и производительности FCoE не сопоставимы с показателями FC, так как передача данных требует дополнительных накладных расходов на инкапсуляцию в Ethernet.

К тому же FCoE добавляет сложность в развёртывании и администрировании всей системы, повышая уровень требований к обслуживающему персоналу и поддержке решения. Несмотря на надежды производителей FCoE на повсеместное внедрение, до сих пор данный протокол не смог вытеснить FC и развивается очень медленными темпами. Распространение FCoE на рынке SAN в настоящий момент минимально.



По данным Fibre Channel Industry Association (FCIA) в прогнозных показателях FCoE скорость протокола зависит от реализаций Ethernet.

iSCSI (Internet Small Computer System Interface)

iSCSI строится на двух наиболее часто используемых протоколах:

- SCSI — протоколе обмена блоками данных между компьютером и хранилищем
- IP — сетевом транспортном протоколе, широко применяемом в корпоративных сетях Ethernet.

iSCSI — это низкобюджетное решение для внедрения. Администрирование таких инсталляций очень простое, хотя для обеспечения отказоустойчивости необходимо строить выделенную сеть для iSCSI, что приближает нас к сетевой реализации, очень похожей на FC SAN.

Считается, что iSCSI 10Гбит обеспечивает такое же количество IOps и пропускную способность, как и сопоставимый ему FC 8Гбит, но это не совсем так. Хотя пропускная способность iSCSI и выше, но его эффективность ниже, чем у FC за счёт дополнительных накладных расходов.

Производительность iSCSI зависит от существующий инфраструктуры Ethernet (на сегодняшний день минимально рекомендованная сеть для iSCSI – 10Гбит). В ближайшем будущем (по данным Gartner, 10–12 месяцев) стоит планировать переход на 25/40/50GbE, если будет необходимость использовать высокопроизводительные all-flash СХД.

SAN-сети на основе сетей Ethernet

Протоколы iSCSI, NFS, SMB, FCoE используют Ethernet-сети для передачи данных, поэтому использовать данные протоколы в общих сетях не целесообразно. Это приводит нас к необходимости развёртывания выделенных сетей для использования их в качестве SAN.

Такие сети должны работать параллельно с другими основными сетями и отделять пользовательский траффик от траффика данных серверов. Однако из-за разделения сетей увеличивается сложность администрирования и общая стоимость владения.

Единственное исключение — это использование общих сетей для объектного хранения, т. к. в данном случае обеспечение минимальных задержек и максимальной производительности не так критично.

NF**S (Network File System)**

Network File System (NFS) — протокол сетевого доступа к файловым системам, первоначально разработанный Sun Microsystems в 1984 году. NFS предоставляет клиентам прозрачный доступ к файлам и файловой системе сервера.

NFS просто конфигурируется и администрируется, т. к. используется поверх сетей Ethernet. Как и в других протоколах, использующих Ethernet, скорость и задержки целиком зависят от реализации сети на нижнем уровне и чаще всего «упираются» в ограничения стандарта 10GbE.

NFS часто используется как протокол начального уровня для построения SAN-сети для виртуализации. По прогнозам Gartner, такой тренд сохранится в течение последующих 10 лет.

SMB (Server Message Block)

SMB — это сетевой протокол для общего доступа к файлам, который позволяет приложениям компьютера читать и записывать файлы, а также запрашивать службы серверных программ в компьютерной сети. Разработан компанией Microsoft для реализации «Сети Microsoft Windows» и «Совместного использования файлов и принтеров». С увеличением пропускной способности сетей передачи данных SMB стал одним основных протоколов файлового доступа, применяемых в СХД. В системах хранения SMB чаще всего используется в сетях 10GbE, из-за этого его производительность сильно зависит от реализации, настроек и используемых компонентов сети.

До версии SMB 3.0 протокол в основном использовался для передачи файлов в локальных сетях. С новой версией, поддерживающей технологию SMB-Direct (использование RDMA), стал активно применяться в кластерах виртуализации на базе MS Hyper-V как протокол доступа к общему пулу хранения.

Как и NFS, SMB часто используется в качестве протокола начального уровня при построении SAN-сети для виртуализации. Эта тенденция также должна сохраниться в ближайшее десятилетие.

InfiniBand (IB)

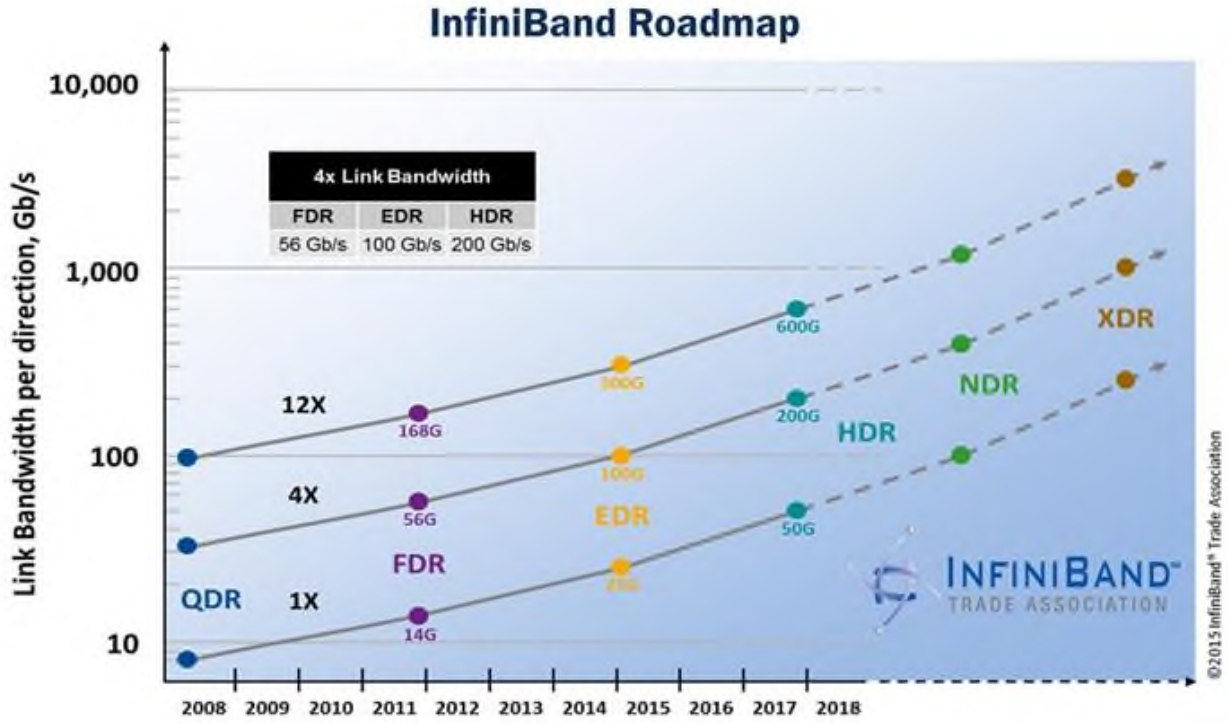
InfinBand — высокоскоростной протокол, обеспечивающий очень большую пропускную способность и низкие задержки. Используется, преимущественно, в отрасли высокопроизводительных вычислений (HPC) и в качестве интерконнекта при создании высокопроизводительных СХД.

Наиболее явным недостатком этой технологии является сложность в настройке и администрировании. Как следствие, работа с IB требует квалифицированного персонала.

Среди трудностей повсеместного распространения InfinBand можно отметить ограниченные средства для мониторинга производительности и диагностики проблем, а также не лучшую совместимость с разными операционными системами.

В сравнении с FC и Ethernet протокол InfiniBand более «дорог» для внедрения и обслуживания. К тому же существует лишь несколько считанных

компаний, которые производят оборудование и программное обеспечение для работы с IB.



NVMe (NVM Express)

NVMe — это достаточно новый высокопроизводительный протокол доступа к твердотельным накопителям, подключенным по шине PCI Express. Аббревиатура «NVM» в названии обозначает энергонезависимую память (Non-Volatile Memory), в качестве которой в SSD повсеместно используется флеш-память типа NAND.

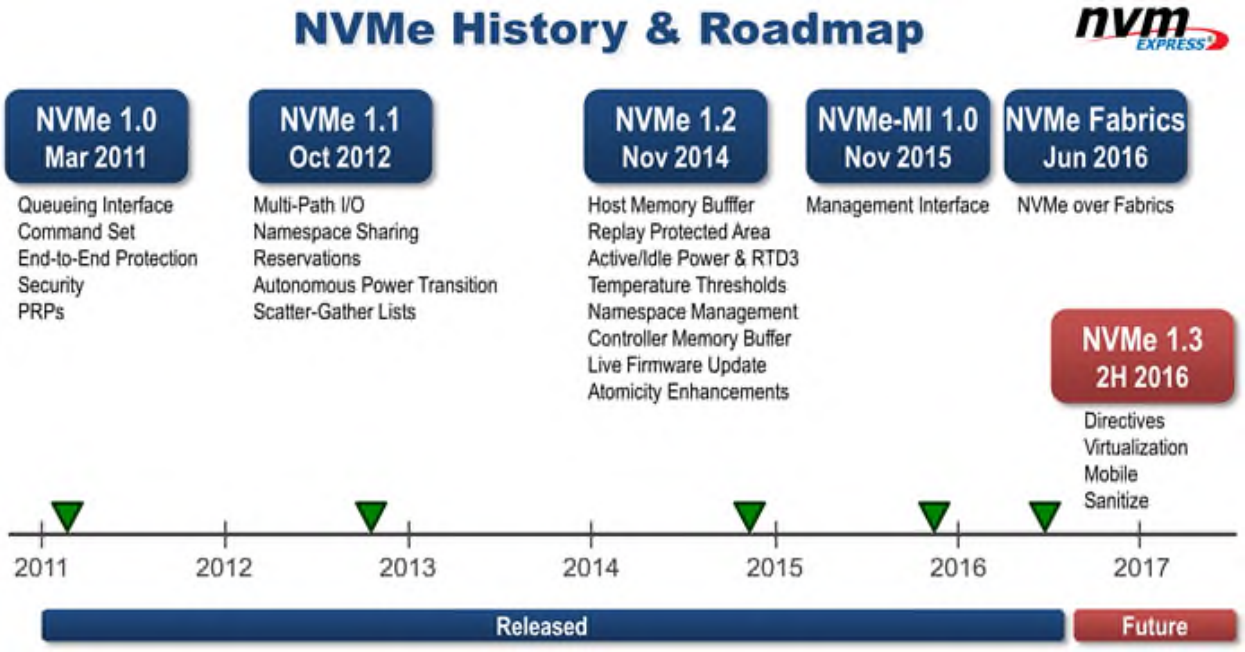
Протокол был разработан с нуля. Основные цели его разработки – низкие задержки и эффективное использование высокого параллелизма твердотельных накопителей. Последняя задача решается за счёт применения нового набора команд и механизма обработки очередей ввода-вывода, оптимизированного для работы с современными процессорами.

На данный момент технология NVMe еще не получила широкого распространения. В основном протокол используется для внутреннего соединения в серверах и СХД. Спецификация NVMe также позволяет инкапсулировать его в другие протоколы, такие как Ethernet, FC и InfiniBand, и масштабировать протокол в более крупных сетях. Впрочем, поскольку NVMe использует прямой доступ к памяти (RDMA), латентность протокола несущей должна быть очень низкой для нормальной работы протокола.

Ожидается активное внедрение NVMe, так как многие производители серверных платформ представляют новые модели с поддержкой

двухпортовых NVMe-устройств, что позволит проектировать отказоустойчивые решения для хранения.

Ожидается, что в ближайшие несколько лет NVMe будет использоваться в качестве внешнего межсетевого интерфейса, аналогичного PCIe и InfiniBand. Например, он может использоваться в небольших специализированных сетях хранения из десятков узлов или устройств в однородной среде хранения. Гораздо шире NVMe будет использоваться в качестве внутреннего интерконнекта.



PCIe

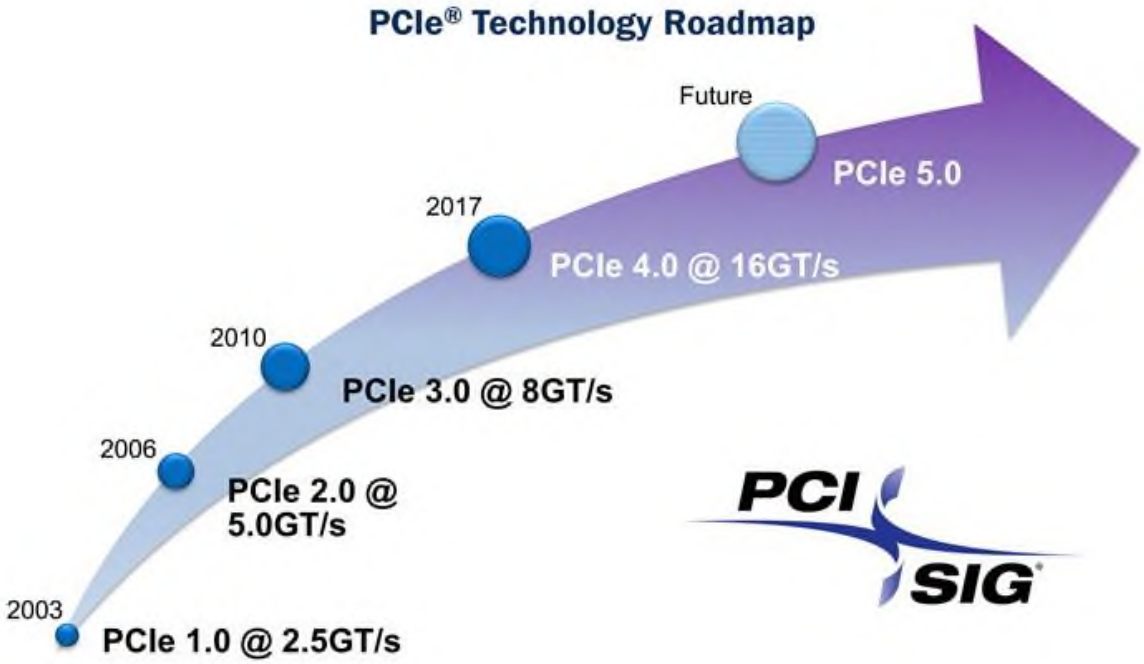
PCIe имеет очень низкие задержки и используется преимущественно в серверах для подключения плат расширения, в том числе и высокопроизводительных NVMe дисков. Некоторые новые продукты от известных производителей используют PCIe в качестве протокола подключения серверов через небольшие PCIe коммутаторы, т. е. PCIe получается использовать только в небольших SAN-сетях.

Только несколько производителей СХД в мире используют PCIe для внешних подключений, что и определяет узкоспециализированное применение данного протокола.

Так как PCIe не использует SCSI и требует собственного протокола для передачи данных, он может увеличить пропускную способность за счёт уменьшения задержек, по сути, работая на скорости чистой PCIe-линии. Такие решения требуют применения проприетарных драйверов, что делает их сложно администрируемыми и приводит к невозможности создавать гетерогенные инфраструктуры, а также встраивать такие решения в существующие SAN сети.

На сегодняшний день основная используемая реализация технологии — это PCIe 3.x, в которой производительность увеличена на 40% по сравнению PCIe 2.x.

По количеству линий PCIe масштабируется от 1ГБ/с до 16ГБ/с. В 2017 году выходит новый стандарт PCIe 4.x, который увеличит производительность в 2 раза, т. е. максимальная производительность достигнет 32ГБ/с.



Протоколы объектного хранения

К объектному хранению относится большое количество новых протоколов, которые работают поверх HTTP, через API. На сегодняшний день многие новые приложения разрабатываются с учётом использования данных протоколов, это особенно актуально для программного обеспечения для резервного копирования.

Производительность этих протоколов сильно зависит от нижнего уровня, чаще всего это Ethernet. Основное функциональное применение протоколов объектного хранения — работа с неструктурированными данными. Ввиду этого они используются в рамках ЦОДов, а также широко применяются для доступа к данным и записи в облачные пулы хранения.

Протоколы объектного доступа требуют передачи и хранения большого объёма метаданных, что всегда добавляет накладные расходы, особенно в сравнении протоколами блочного доступа.

За последние несколько лет было разработано множество протоколов объектного доступа. Наиболее популярные из них – SOAP, S3, OpenStack Swift. В ближайшем будущем (по данным Gartner, в течение 5 лет) данные технологии будут вызывать особый интерес.

Вывод

Развитие SAN-протоколов напрямую зависит от развития приложений и профилей нагрузки.

Такие приложения и типы нагрузок, как в HPC, аналитике Big Data и активных архивах, будут двигать SAN в сторону создания СХД с очень низкими задержками и высочайшей пропускной способностью, а также с поддержкой разделяемого доступа с использованием NVMe.

Протоколы, которые будут внедряться в ближайшем будущем (такие как 40/100GbE), файловые протоколы (такие как NFS over RDMA и SMB-Direct) и текущие блочные протоколы (такие как FC 16Gb/s), уже сейчас слишком медленные для следующего поколения Flash и гибридных СХД.

Основным протоколом на ближайшее десятилетие останется FC, так как под него создана вся необходимая инфраструктура. Многие будут переходить на FC 16Гб/с, затем на FC 32Гб/с и более новые версии.

InfiniBand, PCIe и NVMe останутся протоколами для подключения конечных устройств, межузловых подключений в кластерах или интерконнекта с малыми задержками. При этом будут и небольшие узкоспециализированные решения для SAN-сетей, в которых необходимы минимальные задержки и максимальная пропускная способность. FCoE, iSCSI, NFS, SMB или объектные протоколы будут использоваться преимущественно в качестве внешних протоколов.

5. Типы СХД

Какие бывают типы СХД?

Есть три больших типа систем хранения данных: файловое хранилище, объектное хранилище и блочное хранилище.

Блочное хранилище

Работает по блочным протоколам SAN: Iscsi и Fibre Channel. Они подключаются к серверам с нарезанием томов и файловых систем для дальнейшего использования. Блок данных – это не какой-то законченный объект, а кусок фиксированного размера, содержащий некие данные. Файл может занимать несколько блоков, и, если последний из этих блоков заполнен не до конца, на его размер это не повлияет. Блочное хранилище может быть использовано любой операционной системой в качестве дискового тома этой системы, а пользователь видит этот диск в интерфейсе

ОС сервера. Она, в свою очередь, может быть как физической, так и виртуальной. Однако физическим серверам в отличие от виртуальных нужны дополнительные контроллеры для того, чтобы получать доступ к исходным блокам. Преимуществом блочных систем является высокое быстродействие.

Файловые хранилища

Могут работать как с конечными пользователями (например, сотрудниками, которые со своего компьютера могут открыть файлы компании), так и с серверными мощностями. Такие хранилища внешне больше всего напоминают организацию информации в наших компьютерах: файлы вложены в папки. Доступ к данным осуществляется по file ID, содержащему имя сервера, путь к директории, а также имя искомого файла в NAS. Чаще всего в качестве протоколов доступа к файлам через NAS используют протоколы NFS (для Unix и Linux) и CIFS (для Windows). Последний является публичным вариантом более специализированного протокола SMB (Server Message Block), который использует сетевой протокол TCP/IP. Сервер файловой СХД использует внутри локальной файловой системы блочное хранилище, в то время как пользователь работает лишь с внешним протоколом, определяющим путь к файлу.

Объектное хранилище

Можно использовать в разных целях. Основные его задачи – сохранять большие объемы данных в виде объектов, которые можно моделировать и которым можно присвоить метаданные. В СХД этого типа применяются примерно такие же технологии, что и в публичном облаке (HTTP, API). Объектные хранилища легко масштабируются до объемов петабайта в одном домене, без снижения производительности. В отличие от традиционных СХД, в объектных хранилищах есть функционал управления данными – например, кастомизации метаданных и встроенной аналитики. На внешнем уровне размещены средства управления, а на внутреннем уровне каждый диск форматируется простой локальной файловой системой (например, EXT4). Это позволяет пользователю управлять функциями внешнего уровня через стандартный интерфейс прикладного программирования API, а все элементы СХД интегрированы в единый унифицированный том.

Технологии хранения

Говоря о технологиях хранения, невозможно обойти вниманием термин RAID. Redundant array of independent disks — избыточный массив независимых дисков — это технология виртуализации данных, которая объединяет несколько дисков в логический элемент для повышения производительности.

В зависимости от выбранного типа RAID, технологии хранения делятся на два класса:

С использованием аппаратного RAID. Более дорогое и не всегда оправданное решение, связанное с покупкой дополнительного компьютерного «железа» с собственной памятью и выделенным процессором. Аппаратный RAID требуется при наличии в системе как минимум четырех и более накопителей.

С использованием программного RAID. В этой технологии используются контроллеры на материнской плате, которые не имеют своей памяти и выделенного процессора. Они используют от 2–5% ресурсов центрального процессора сервера. Не менее надежны, чем аппаратные решения, используются в небольших системах.

Устройства хранения

DAS. Накопители ставятся непосредственно в сервер для получения дополнительного пространства со сравнительно быстрым доступом. Самый простой и недорогой вариант.

NAS. Хранилище, подключаемое по сети. Отличается гибкостью и централизованным управлением, однако скорость доступа ограничена скоростью сети.

SAN. Хранилище, подключаемое через оптико-волоконный кабель. Сочетает в себе все плюсы NAS с высокой скоростью доступа.

Создание системы хранения данных

Для создания хранилищ данных требуется разработка логической модели, которая будет полностью отражать ожидания клиента и возможности разработчика. После этого можно рассматривать технологические аспекты – например, размеры хранилища. Логическая модель может содержать тысячи атрибутов и связей.

Стоимость СХД варьируется в зависимости от масштаба, логической модели и оборудования. В одних случаях речь идет о сотнях тысяч рублей, в других – о десятках миллионов. На создание СХД может уйти от одного месяца до полугода. Важным фактором, который следует учитывать, является необходимость сервисной поддержки оборудования. Ее можно заказать непосредственно в представительстве мирового производителя или у локальной IT-компании. Во втором случае стоимость владения СХД заметно снизится. Оригинал статьи: https://www.kp.ru/guide/sistemy-khranenija-dannykh.html

Как новые технологии влияют на развитие СХД?

СХД – это довольно классические решения, которые меняются эволюционно, а не революционно. Можно заметить изменения в интерфейсе: с 16 ГБит/с на 32 Гбит/с, если говорить про SAN, и с 6 Гбит/с на 12 Гбит/с, если говорить про SAS. Считается, что NVMe существенно понизит задержки и обеспечит точки роста, поэтому многие аналитики связывают с ним будущее СХД.

Есть технология FCoE (Fibre Channel over Ethernet) – многие продвигали эту технологию и фактически технология не то, чтобы умерла, но сейчас о ней уже никто не вспоминает».

Как выбрать СХД?

Выбор СХД зависит от нескольких критериев: это и объем компании, и ее бизнес-задачи, и тип системы. Если мы говорим о блочной СХД, то в первую очередь нужно определиться с тем, какие объемы данных будут на ней хранить, а также с тем, какая для этих объемов требуется производительность и какие требования по уровню надежности предъявляет заказчик. Иногда выдвигаются такие критерии, с которыми не могут справиться никакие решения, кроме большого SATA–райзера с дисками SSD. Соответственно, именно из них и следует выбирать. Однако бывает, что у компании нет необходимости защищать решение даже от падения контроллера, –в таком случае можно выбрать недорогую СХД и надеяться, что она будет работать без критических нарушений».

Есть ли специальные требования к СХД?

Заказчики могут предъявлять специфические требования к самым разным аспектам СХД: к защите данных, к производительности, к масштабированию, к поддержке приложений, к интеграции в системы частного облака, к интерфейсам взаимодействия. Так, кому-то нужно описать скрипты и управлять ими из какого-нибудь портала СХД. При выборе решений создается некий шорт-лист производителей с их плюсами и минусами, а также соответствием требованиям, и уже из этого списка отбирается решение. Как правило, основной критерий, которым руководствуются компании – это цена

Из чего складывается цена на СХД?

Если говорить про формирование внутренних цен, то в стоимость входят все компоненты СХД: как те, что производит сам вендор, так и те, которые он закупает. Набор закупаемых компонентов может быть достаточно широким, включая в себя процессоры, платы, HBA-адаптеры, диски, стоимость которых зависит от объемов, целей и договоренностей. Таким образом определяется

первичная стоимость СХД, которая при этом не составляет и половины итоговой цены.

Дело в том, что кроме неё в конечную стоимость СХД входит цена разработки и сервисного обслуживания. Иногда у заказчиков возникает вопрос, почему решения с одинаковыми компонентами стоят по-разному – все зависит от того, сколько затрат вложено в разработку и каково качество программного обеспечения.

Какие возможности по защите данных предлагают разные СХД?

Разные модели СХД обладают примерно одинаковой системой защиты данных. Может возникнуть вопрос по поводу репликации: есть ли она, нужна ли, и стоит ли ее вообще делать на уровне СХД.

Есть несколько способов защищать данные, один из них состоит в защите на уровне приложений. С точки зрения целесообразности от СХД в данном случае требуется трудоемкость и производительность, а не высокая надежность.

Есть ли среди СХД те, у которых риск потери данных выше?

Чем ниже класс решения, тем выше риски потери данных, и наоборот. Зачастую компании, которые только начинают расти, используют решения Entry Level. Поначалу все идёт хорошо, но потом возникают вопросы: а что, если сломается контроллер? Как его менять? Что делать, если нужен рост производительности? В связи с этим решение меняется на уровень Enterprise, с более низкими рисками потери данных. «В наших решениях используются диски, в которых может быть до 12 контроллеров, и даже если четверть из них выйдет из строя, данные останутся неповрежденными», – отмечает Алексей Никифоров.

Однако при всех достижениях в технологиях защиты данных, нужно учитывать, что далеко не для каждой компании риски их потери могут быть критичными. Если в организации хранятся данные, которые «не жалко», то и беспокоиться по поводу надежности системы лишний раз не стоит.

6. Достоинства и недостатки типов СХД

Достоинства DAS:

легкость развёртывания и администрирования;

высокая скорость передачи данных;

низкая стоимость оборудования.

Недостатки DAS:

неоптимальное расходование ресурсов (требует выделенного сервера);

ограничения в подключениях (не больше двух серверов);

низкая надёжность и слабая распределённость хранимой информации (в случае выхода управляющего сервера из строя вся система хранения становится недоступной).

Достоинства NAS:

доступность и низкая стоимость;

простое совместное использование информации;

простота подключения и управления;

гибкость, возможность быстро увеличить объём для хранения данных;

универсальность клиентов (компьютер под управлением любой операционной системы может получить доступ к файлам).

Недостатки NAS:

хранение данных только в виде файлов;

медленный доступ к информации по сетевым протоколам (по сравнению с локальной системой);

высокая загрузка ЛВС;

невозможность работы некоторых приложений с сетевыми дисками.

Достоинства SAN:

высокая скорость работы, низкая задержка;

гибкость и масштабируемость;

хранение данных блоками (как, например, для почтовой базы Exchange);

высокая скорость работы;

простое использование кэширования данных;

универсальность (доступ к данным может получить сервер приложений под управлением любой ОС);

использование территориально разнесённых устройства хранения;

высокая надёжность обмена и хранения данных;

организация удалённого архивирования и восстановления данных без дополнительных затрат;

агрегирования каналов;

“горячее” подключение устройств;

централизованное управление коммутацией и логистикой данных,

разгрузка подсети от служебного трафика.

Недостатки SAN:

сложность проектирования;

высокая стоимость;

сложность настройки протокола FC;

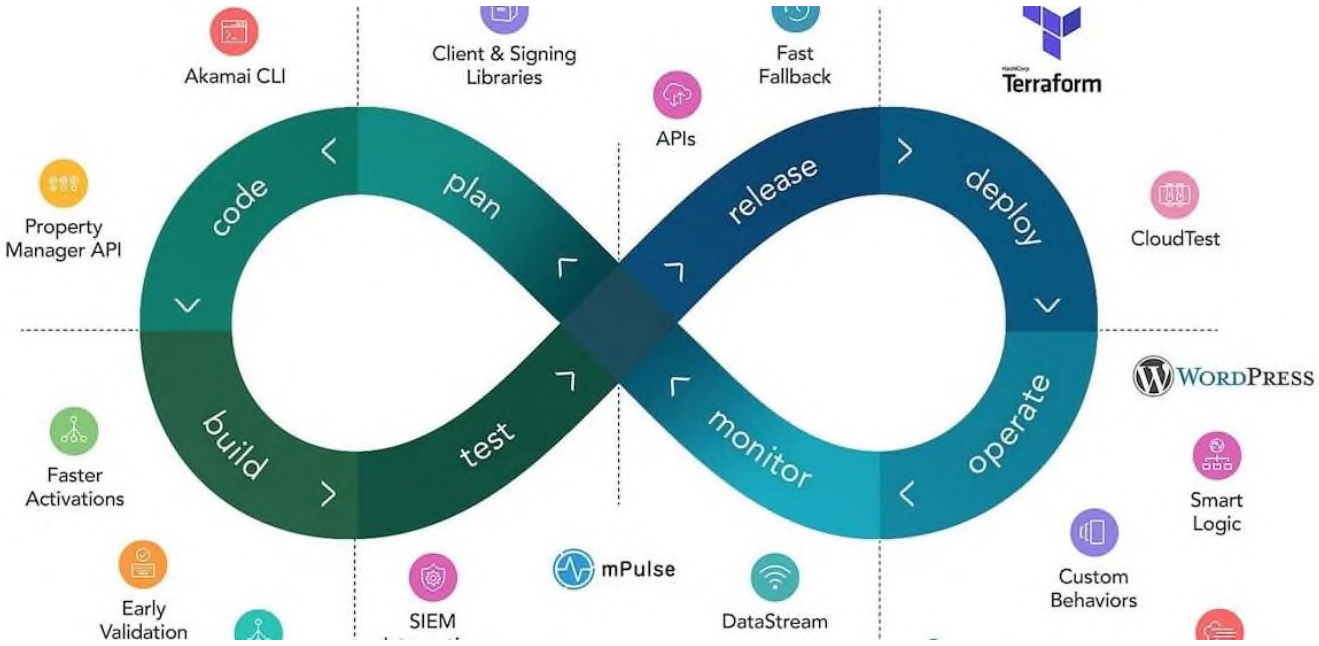
невозможность некоторых приложений и систем работать с протоколом iSCSI;

необходимость подготовки специалистов по протоколу FC и их сертификация;

жесткие требования по совместимости оборудования в сети.

Заключение

Каждая из систем заняла определённую нишу. DAS чаще используется в небольших компаниях и у хостинг-провайдеров. NAS применяется там, где ограничения по ёмкости в DAS играют решающую роль. SAN популярна у больших компаний, имеющих филиалы, а также у фирм, занимающихся виртуализацией.



МОДУЛЬ 5. Базы данных

1. Базы данных (иерархические, реляционные, объектно-реляционные и пр.)
2. Работа с данными (DDL/DML)
3. NoSQL
4. SQL
5. Индексы
6. Репликация и масштабирование
7. Резервное копирование
8. База данных в облаке

1. Базы данных (иерархические, реляционные, объектно-реляционные и пр.)

Что такое база данных

База данных — это набор сведений об объектах, структурированный определенным образом. Обычно базы данных управляются специальным ПО, или системами управления базами данных (СУБД).

В зависимости от вида логическая структура базы данных может иметь различное описание. Это различие влияет на то, какая именно БД используется в разработке конкретного продукта или технологии.

Простейшие типы баз данных

К таким базам данных относятся БД, где хранятся данные с простой структурой: например, список разрешенных IP-адресов для доступа к сети, настройки окружения проекта, список подписчиков на рассылку компании и прочее. Они все еще широко распространены.

Текстовые файлы

Информация об объектах собирается в простых по структуре файлах различных форматов – txt, csv и др. Для разделения полей применяются пробелы, табуляция, запятые, точка с запятой и двоеточие.

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
syslog:x:102:106::/home/syslog:/usr/sbin/nologin
```

Примеры: etc/passwd и etc/fstab в Unix-подобных системах, csv-файлы, ini-файлы и др.

Особенности:

Просто использовать. Для работы с файлами достаточно примитивного текстового редактора.

Удобно работать с конфигурационными данными приложений (учетные данные, настройки подключения к удаленным серверам и устройствам, порты и пр.).

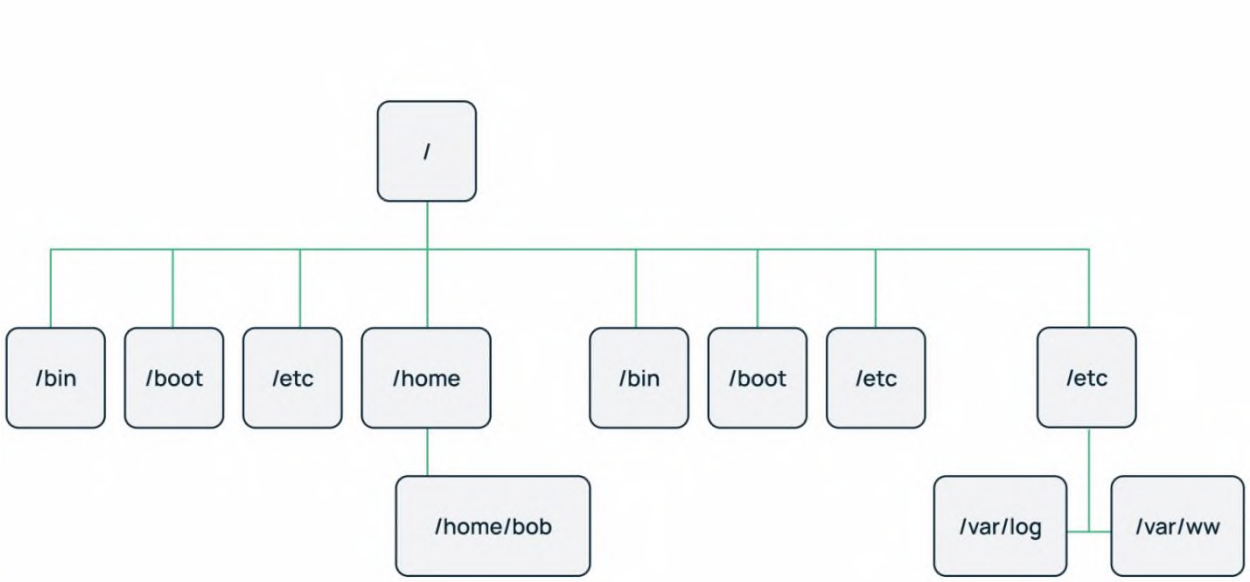
Ограничения:

Сложно установить связи между компонентами данных.

Не для всех типов информации.

Иерархические базы данных

В отличие от текстовых файлов здесь между хранимыми объектами устанавливаются связи. Объекты делятся на родителей (основные классы или категории объектов) и потомков (экземпляры этих классов или категорий). При этом у каждого потомка может быть не более одного родителя.



Пример иерархической базы данных.

Графическим представлением такой базы данных является древовидная структура.

Примеры: Организация файловых систем; DNS и LDAP-соединения.

Особенности:

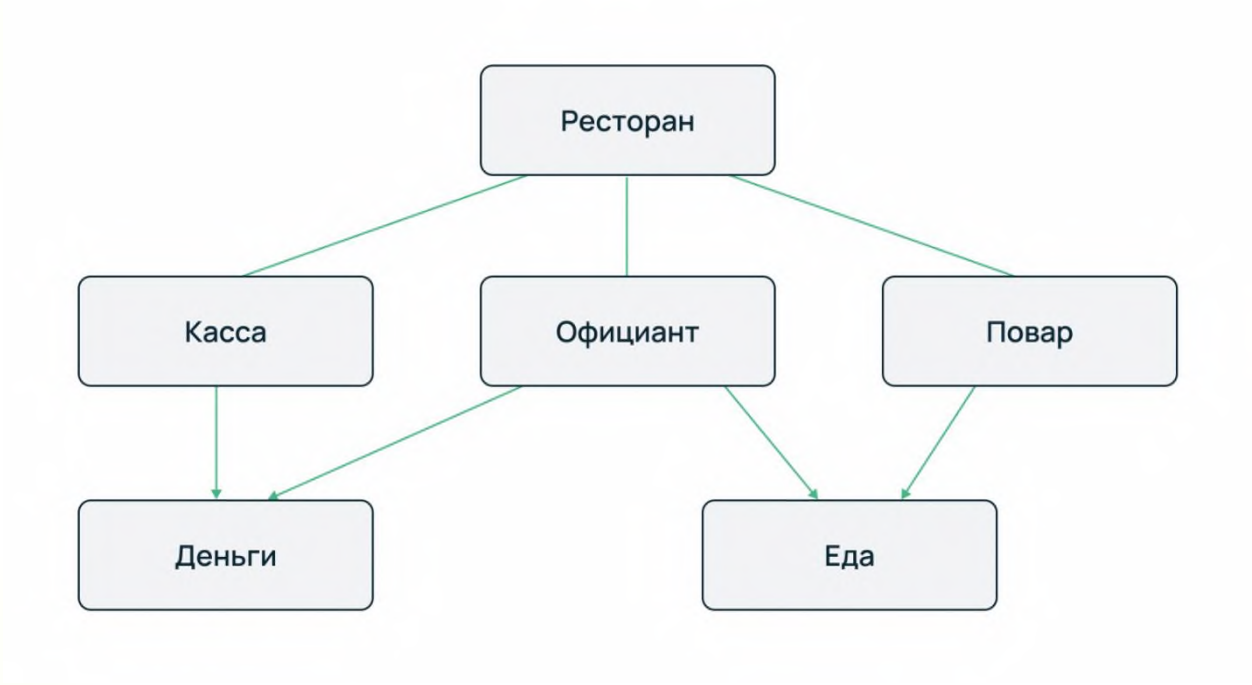
Отношения между объектами реализованы в виде физических указателей. Например, в файловой системе путь к папке или файлу строится из имен корневых и вложенных каталогов;

Моделирование отношений вложенности и подчиненности.

Ограничения: Технология иерархической организации не предполагает связи «многие-ко-многим», а значит, система хранения данных довольно ограничена.

Сетевые базы данных

Эта технология развивает иерархический подход за счет моделирования сложных отношений между объектами. Здесь потомки могут иметь более одного родителя, однако ограничения иерархического подхода сохраняются.



Пример сетевой базы данных.

Пример: IDMS — специализированная СУБД для мейнфреймов.

Реляционные базы данных

Данный тип БД является старейшим: теоретические основы подхода заложены британским ученым Эдгаром Коддом в 1970 году. Здесь данные формируются в таблицы из строк и столбцов. В строках приводятся сведения об объектах (значения свойств), а в столбцах — сами свойства объектов (поля).

Нормализация

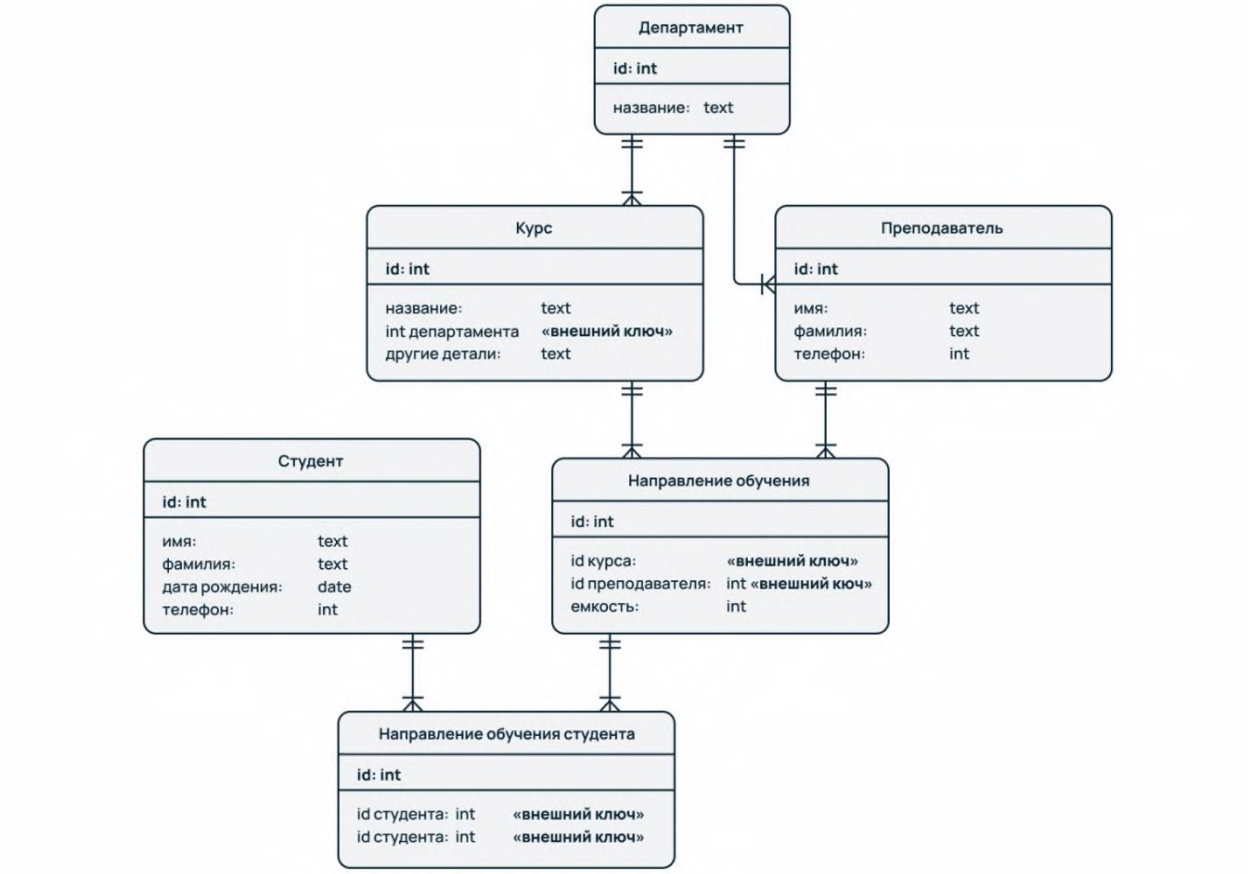
Сложные взаимоотношения объектов в реляционных БД моделируются с помощью внешних ключей – ссылок на другие таблицы. Это позволяет подходить к вопросу проектирования базы данных с позиций нормализации – минимизации избыточности при описании свойств объектов.

Например, если речь идет о меню ресторана, то у каждого блюда есть вес, цена, наименование, калорийность и категория, к которой оно относится — горячие закуски, холодные закуски, первые блюда, десерты, салаты и так далее. Связь между блюдами и категорией выполняется посредством ссылочного поля индекса категории в таблице блюд.

Такой подход позволяет:

- Минимизировать объем базы данных: не нужно каждому блюду прописывать название категории.

- Повысить целостность системы: в указанном примере все блюда привязаны к категориям меню. Добавление блюда без категории невозможно, равно как и указание в качестве ссылки индекса несуществующей категории.
- Упростить масштабирование: новые блюда могут быть добавлены в существующие категории. Также не исключается добавление новых категорий, привязка новых блюд к ним и перераспределение блюд по категориям.
- Повысить отказоустойчивость: за счет оптимальной организации схемы таблиц запросы на выборку и агрегацию будут работать с меньшим объемом данных, а значит, быстрее, чем без нормализации. При увеличении числа записей в таблицах со временем это позволит поддерживать положительный пользовательский опыт.



Наглядный пример моделирования сложных взаимоотношений в реляционных БД приведен на рисунке выше. Здесь мы видим модель базы данных учебного заведения, где есть следующие объекты: ученик, курс, преподаватель, отдел, направление обучения.

Связь преподавателя с отделом организована через секцию и курс (внешние ключи id курса и id преподавателя в таблице Секция, а также Отдел в таблице Курс). Связь ученика с направлением обучения реализована через таблицу Направление обучения студента (внешние ключи id студента и id направления обучения).

Таким образом, чтобы посчитать, например, количество студентов на курсе и детализировать статистику по преподавателям, необходимо написать запрос с присоединением учеников к направлению, курсу и преподавателям, сделав соответствующую группировку по преподавателям.

Язык запросов SQL

Запросы в реляционных базах данных формируют с помощью структурированного языка SQL. Его предложения позволяют:

делать выборки,

проводить агрегации и группировки,

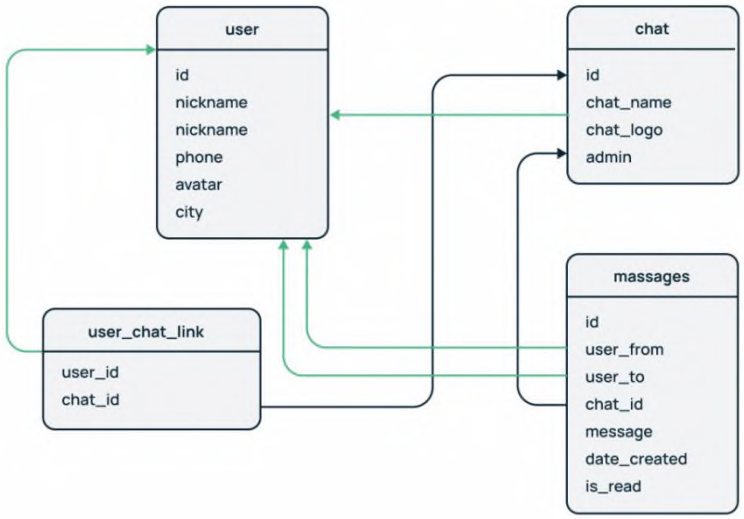
изменять и удалять данные,

модифицировать структуру БД (создавать таблицы, поля),

управлять доступом пользователей к тем или иным операциям и пр.

Денормализация

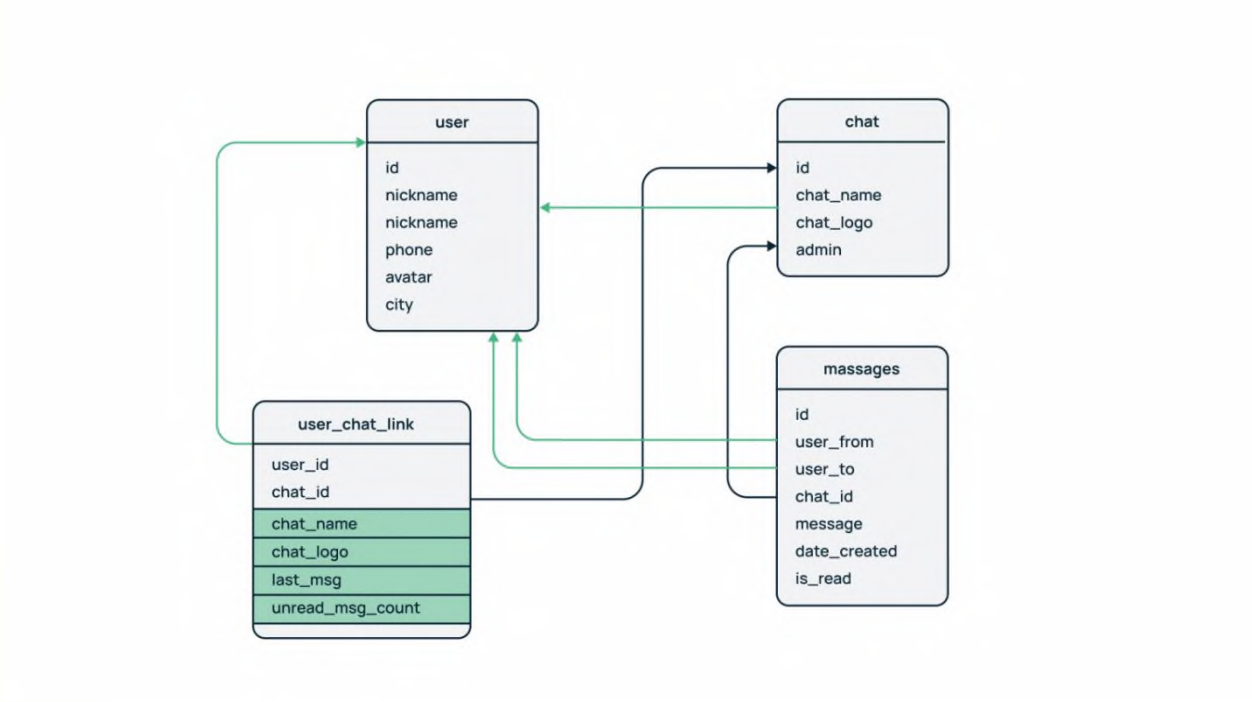
Помимо нормализации, в реляционных БД существует и обратный процесс — денормализация. Он направлен на перенос наиболее часто используемых полей из внешних таблиц во внутренние. Рассмотрим это на примере мессенджера.



Пользователь (user) оставляет сообщения (messages) в чатах (chat). Структура данных такова, что сообщения связаны с пользователем и чатом через внешние ключи (user_from и user_to, а также chat_id в таблице сообщений; user_id и chat_id в таблице user_chat_link). Поскольку схема нормализована, то различные запросы на выборку, подсчет и агрегацию статистики по чатам, пользователям и сообщениям необходимо выполнять с помощью присоединения внешних таблиц.

На относительно небольших объемах данных эти запросы будут отрабатывать быстро, а с увеличением размера базы – замедляться. Причина

кроется в механизме присоединения. Он основан на построчном сравнении двух и более таблиц по условию соединения — например, равенство chat_id в messages и id в chat. А это дает нагрузку на сервер базы данных, которая с ростом ее размера только увеличивается. Для оптимизации такого рода запросов и существует механизм денормализации.



В таблицу связи пользователя и чата user_chat_link добавлены дублирующие поля имени чата (chat_name) и аватара (chat_logo). Также туда выводятся последнее сообщение (last_msg) и количество непрочитанных сообщений (unread_msg_count).

Теперь для получения указанных выше полей и проведения аналитики по ним можно использовать таблицу user_chat_link без необходимости соединения с таблицей сообщений. Тем не менее, такой подход имеет ограничения.

За счет дополнительных полей оптимизируются запросы на чтение и агрегацию данных, однако ценой этого является вынужденная избыточность и усложнение бизнес-логики приложения. В частности, усложняется написание запросов изменения данных (update и delete), а также модификации структуры базы (create).

Использование денормализации должно быть тщательно осмыслено. Нужно быть уверенным в том, что нормализованная структура, оптимизированные запросы и правильно настроенные индексы более не способны удовлетворять критерию быстродействия.

Преимущества реляционного подхода:

определение сложных отношений между объектами,

нормализация и денормализация данных,

структурированный язык запросов,

богатая история развития и широкое распространение (основной инструмент при разработке различных приложений и сервисов).

Недостатки подхода: жесткая структура сведений об объектах.

Примеры: MySQL, MariaDB, PostgreSQL, SQLite и др.

NoSQL и нереляционные базы данных

Все преимущества и недостатки реляционных БД основаны на жесткой структуризации и типизации сведений об объектах. С одной стороны, можно оптимизировать хранение и индексирование данных за счет нормализации или же денормализации. С другой — сложно организовать хранение и обработку плохо структурированных (например, объекты кэша) или вовсе не структурированных данных (например, данные из нескольких источников).

Для борьбы с этими ограничениями было разработано семейство нереляционных БД. Рассмотрим их подробнее.

Базы данных «Ключ-значение»

Это простейшая разновидность нереляционных БД. Данные хранятся в виде словаря, где указателем выступает ключ.



Особенности:

Хранение и обработка разных по типу и содержанию данных: в одном хранилище под разными ключами могут находиться файлы, строки, текст, числа, JSON-объекты и другие типы данных.

Высокая скорость доступа к данным за счет адресного хранения.

Легкое масштабирование. Можно создать правила шардирования по определенным ключам – например, сессии пользователей разных сайтов хранятся в различных сегментах БД.

Ограничения: поскольку подход не предполагает жесткой типизации и структуризации данных, то контроль их валидности, а также нейминг ключей отдаются на откуп разработчику.

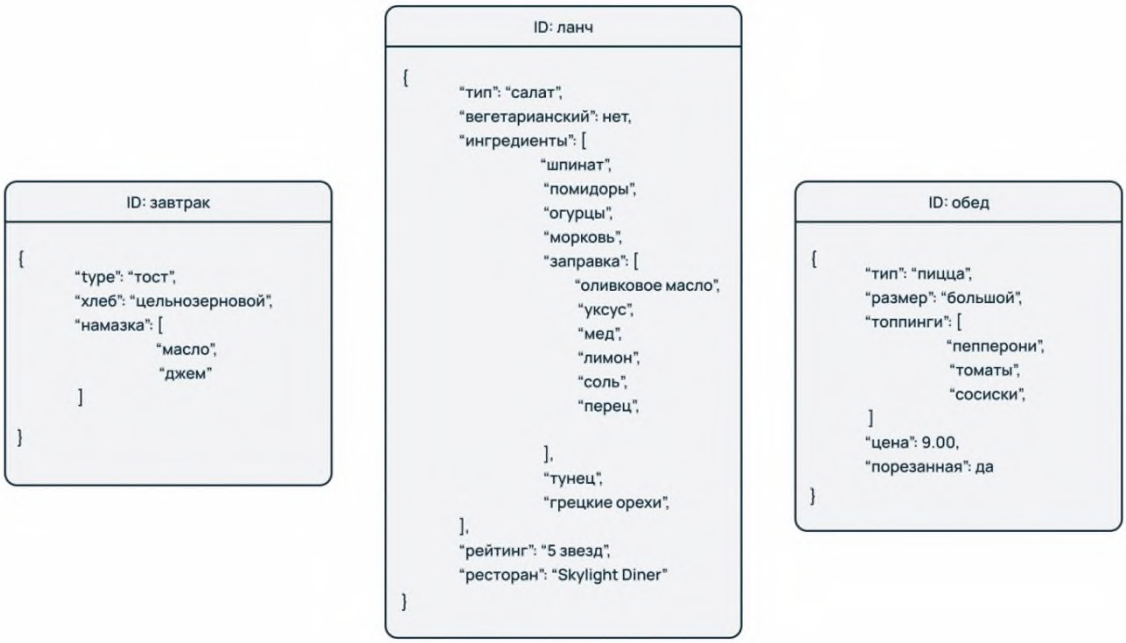
Примеры: Amazon, DynamoDB, Redis, Riak, LevelDB, различные хранилища кэша – например, Memcached и пр.

Документно-ориентированные БД

В отличие от баз типа «Ключ-значение» данные здесь хранятся в структурированных форматах – XML, JSON, BSON. Тем не менее, сохраняется

адресный доступ к данным по ключу. При этом содержимое документа может иметь различный набор свойств.

Например, каталог профилей пользователей: один в качестве предпочтений указал любимое блюдо, а другой – видеоигру. Поскольку эти сведения нельзя хранить в одном поле ввиду логической и структурной разобщенности, они записываются в отдельные свойства отдельных документов. При необходимости можно добавить в документы новые свойства, не нарушив при этом общей целостности данных.



Особенности:

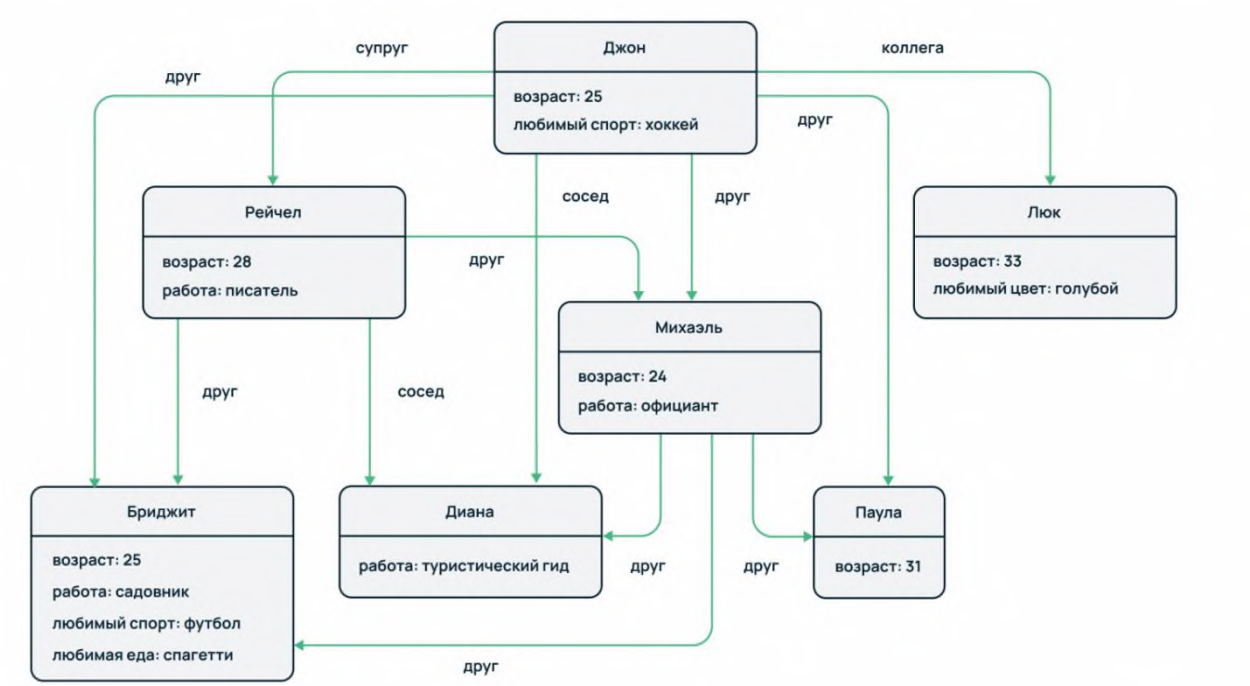
хорошо подходят для быстрой разработки систем и сервисов, работающих с по-разному структурированными данными,

легко масштабируются и меняют структуру при необходимости.

Примеры: MongoDB, RethinkDB, CouchDB, DocumentDB.

Графовые базы данных

Это семейство баз предназначено для моделирования сложных отношений с помощью теории графов, где связями выступают ребра графа, а сами объекты – это узлы или вершины.



Пример структуры графовой базы данных.

Такой подход может пригодиться при анализе профилей пользователей социальных сетей. Один пользователь подписан на обновления второго, другой пользователь подписан на определенное сообщество и так далее. Также технология может использоваться при анализе экономической активности контрагентов для выявления различных схем мошенничества. Например, можно отследить использование определенных счетов, карт или реквизитов контрагентов в различных операциях.

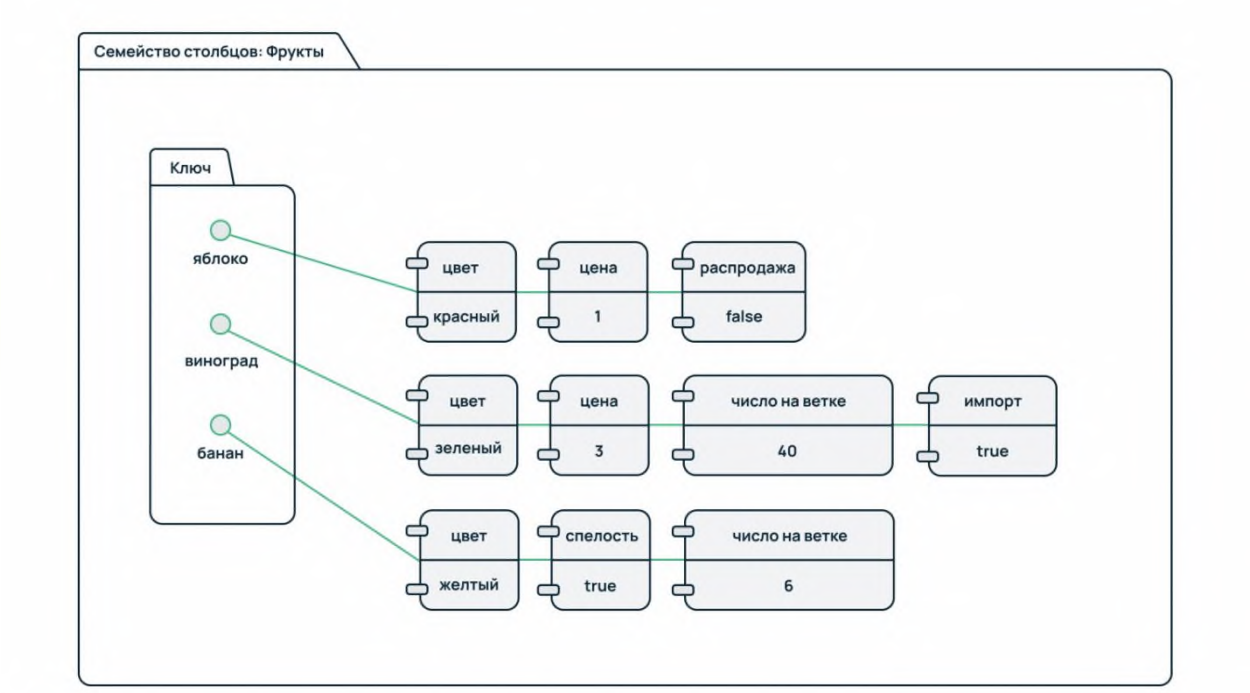
Особенности: высокая производительность, поскольку обход ребер и вершин значительно быстрее анализа множества внешних и внутренних таблиц и их соединения по условию отбора в реляционных БД.

Примеры: Neo4J, JanusGraph, Dgraph, OrientDB.

Колоночные базы данных

Как можно понять из названия, записи в таких базах хранятся не по строкам, а по столбцам (колонкам). Вместо таблиц здесь используются колоночные семейства. Они содержат ключи, указывающие на формат строки записи информации об объекте. Каждая строка имеет свой набор свойств, что позволяет хранить в рамках одного семейства разно структурированные данные.

Технология активно используется при построении аналитических систем и сервисов, работающих с большими объемами данных.



На рисунке приведен пример колоночного хранения информации о фруктах. Известно три типа фруктов: яблоки, виноград, бананы. Все они объединены в семейство фруктов.

У каждого фрукта индивидуальный набор свойств. Для яблок это цвет, цена и наличие. У винограда это цвет, цена, число ягод в связке и происхождение (импортный или нет). У бананов же это цвет, цена, число в связке и зрелость.

Чтобы получить детальную сводку по одному типу фруктов, достаточно в запросе указать его идентификатор. При этом можно построить аналитический запрос по общим для всего семейства признакам – например, посчитать число фруктов с группировкой по цвету, вычислить среднюю цену на все фрукты в магазине и т. д.

Особенности:

С группировкой свойств по колонкам при запросе индексируется меньший объем данных, что обеспечивает высокую скорость его выполнения.

Широкие возможности масштабирования и модификации структуры — так, при добавлении новых колонок не придется их жестко формализовывать, как в случае с реляционными базами.

Примеры: Cassandra, HBase, ClickHouse.

Базы данных временных рядов

Данный тип БД можно использовать при необходимости отслеживания исторической динамики по ряду показателей. Здесь данные группируются по временным меткам. Базы временных рядом чаще ориентированы на запись, чем на построение сложных аналитических запросов.

Время	Температура CPU	Значение нагрузки	Утилизация памяти %
2019-10-31T03:48:05+00:00	37	0.85	92
2019-10-31T03:48:10+00:00	42	0.87	90
2019-10-31T03:48:15+00:00	33	0.74	87
2019-10-31T03:48:20+00:00	34	0.72	77
2019-10-31T03:48:25+00:00	40	0.88	81
2019-10-31T03:48:30+00:00	42	0.89	82
2019-10-31T03:48:35+00:00	41	0.88	82

На рисунке выше приведен пример использования такой БД для отслеживания состояния ПК во времени по ряду показателей – температуре процессора, загрузке системы и потреблению оперативной памяти.

Особенности: можно обрабатывать постоянный поток входных данных.

Ограничения: Производительность зависит от объема поступающей информации, количества отслеживаемых метрик, а также временного лага между записью новых данных и запросами на чтение

Примеры БД: OpenTSDB, Prometheus, InfluxDB, TimescaleDB

Комбинированные базы

Эта разновидность баз совмещает в себе SQL- и NoSQL-подходы к организации хранения и обработки данных. Этот класс баз включает в себя NewSQL и многомодельные решения. Рассмотрим их подробнее.

Базы данных NewSQL

Данный тип решений для хранения информации стремится обеспечить компромисс между масштабируемостью и согласованностью при сохранении реляционного подхода.

Термин предложил в 2011 году аналитик компании 451 Group Мэтью Аслет. Он отмечал высокую потребность в таких системах для сфер, работающих с критическими данными, — здравоохранение, FinTech и пр. Характерными признаками этих решений являются: использование алгоритмов обеспечения консенсуса (алгоритм Paxos, Raft и др.), шардирование и заточка под горизонтальное масштабирование.

Особенности:

- широкие возможности масштабирования,
- высокая производительность и доступность данных.

Ограничения: Высокие требования к аппаратным ресурсам разработчиков. Но если разрабатываемый продукт является высоконагруженной системой, то применение такой БД имеет смысл.

Примеры баз такого типа: MemSQL, VoltDB, Spanner и др.

Многомодельные базы

Такие БД сочетают в себе несколько подходов к организации данных одновременно. Это обеспечивает функциональное разнообразие при разработке систем с их использованием.

Особенности:

- возможность в одном запросе работать с данными, хранящимися в разных типах баз, не нарушая при этом согласованности;
- обширные возможности масштабирования за счет легкой интеграции новых моделей баз данных в существующую инфраструктуру проекта.

Пример решения данного типа: ArangoDB.

Базы данных в Selectel

В Selectel вы можете запустить готовые облачные базы данных — поддерживаем такие СУБД, как PostgreSQL (в том числе для 1С: Предприятие), MySQL, Redis, TimescaleDB.

Облачные базы данных позволяют исключить работу с инфраструктурой: поднять нужное количество нод можно за несколько минут в панели управления компании. Решение отказоустойчивое и легко масштабируется. На экстренный случай создаются резервные копии для отката состояния базы на срок до семи дней.

Большинство рутинных операций по системному администрированию (настройка, конфигурация, обслуживание и обеспечение безопасности) выполняются специалистами Selectel.

Заключение

Мы рассмотрели 11 видов баз данных. Каждый имеет свои особенности и ограничения. Решение о выборе того или иного вида необходимо принимать с учетом:

- сложности хранимых данных и взаимосвязей между ними,
- производительности операций чтения/записи и модификации структуры БД на планируемом объеме данных,
- опыта команды разработки,
- стадии жизненного цикла разрабатываемого продукта (производите ли вы доработку действующего решения либо создаете что-то принципиально новое, каковы ваши текущие и перспективные ресурсные возможности).

2. Работа с данными (DDL/DML)

DDL и DML

Задачи, решаемые с помощью SQL, делятся на две группы:

- задачи создания и изменения структуры базы данных (создание и изменение таблиц и связей между ними);
- задачи работы с данными (добавление, изменение, удаление и выборка данных).

В соответствии с решаемыми задачами SQL делится на два языка:

- DDL - Data Definition Language - язык определения (или описания) данных;
- DML - Data Manipulation Language - язык манипулирования данными.

В SQL насчитывается около ста операторов. Одна часть из них относится к DDL, другая - к DML.

Типы данных

Каждый столбец и домен, из которого берутся значения в этот столбец, имеют свой тип данных. В стандарте SQL определены следующие семь основных типов данных:

- **CHARACTER STRING** - символьные (текстовые) данные;
- **NATIONAL CHARACTER** - символьные данные, соответствующие национальному языку;
- **BIT STRING** - двоичные данные;
- **EXACT NUMERIC** - числа, точно соответствующие своей записи;
- **APPROXIMATE NUMERIC** - числа в научной нотации (мантисса и порядок);
- **DATETIME** - дата и время;
- **INTERVAL** - период времени между двумя точками отсчёта.

Каждый основной тип имеет один или несколько подтипов.

Разработчики СУБД нарушают стандарты SQL. Особенно сильно отличаются от стандартных типы данных. Более того, наборы типов данных в разных СУБД тоже отличаются друг от друга. Например, в СУБД *MS SQL Server* и *MS ACCESS* есть тип данных *MONEY*, а в *Oracle* и в *MySQL* отсутствует. Различие типов данных является одной из причин несовместимости различных СУБД и возникающих трудностей при переносе базы данных из одной СУБД в другую. В таблице 1 перечислены типы данных СУБД ACCESS.

Таблица 1. Типы данных в ACCESS

Тип данных	Размер	Описание
BINARY	1 байт на знак	В поле этого типа могут храниться данные любого типа. Данные не преобразуются (например, в текстовые). Данные отображаются в том же виде, в каком они вводятся в это поле.
BIT	1 байт	Значения «Да» (Yes) и «Нет» (No), а также поля, содержащие одно из двух возможных значений.
TINYINT	1 байт	Целое значение от 0 до 255.
MONEY	8 байтов	Масштабируемое целое от –922 337 203 685 477,5808 до 922 337 203 685 477,5807.
DATETIME (см. DOUBLE)	8 байтов	Дата или время; допустим любой год от 100 до 9999.
UNIQUEIDENTIFIER	128 битов	Уникальный идентификатор, используемый при вызовах удаленных процедур.
REAL	4 байта	Число с плавающей точкой и одинарной точностью от –3,402823E38 до –1,401298E-45 для отрицательных значений, от 1,401298E-45 до 3,402823E38 для положительных значений или значение 0.
FLOAT	8 байтов	Число с плавающей точкой и двойной точностью от –1,79769313486232E308 до –4,94065645841247E-324 для отрицательных значений, от 4,94065645841247E-324 до 1,79769313486232E308 для положительных значений или значение 0.
SMALLINT	2 байта	Короткое целое от –32 768 до 32 767 (см. «Примечания»).
INTEGER	4 байта	Длинное целое от –2 147 483 648 до 2 147 483 647.

DECIMAL	17 байтов	Тип данных для хранения точных числовых значений от -10^28 - 1 до 10^28 - 1. Точность (1 - 28) и фактор масштабирования (от 0 до заданной точности) определяются пользователем. По умолчанию точность и фактор масштабирования равны соответственно 18 и 0.
TEXT	2 байта на знак	От 0 до 2,14 Гбайт.
IMAGE	Не ограничено	От 0 до 2,14 Гбайт. Используется для объектов OLE.
CHARACTER	2 байта на знак	От 0 до 255 знаков.

DDL - язык описания данных

Структура реляционной базы данных

Реляционная база данных состоит из связанных между собой таблиц. Каждая таблица состоит из строк, имеющих одинаковую структуру. Все клетки строки имеют имена (идентификаторы). Совокупность клеток с одним именем, расположенных в разных строках, образует столбец. К сожалению, клетку и столбец часто называют одинаково - *полем* и из контекста не всегда ясно, что имеется в виду под полем. Наименьшей единицей информации в реляционной базе данных является клетка.

Электронная база данных — это склад информации в памяти ЭВМ. Структура базы данных похожа на структуру склада физических предметов, например склада бытовой техники. Близки к электронным базам данных хранилища (склады) информации на бумажных носителях - библиотеки и архивы. Вся база данных аналогична зданию, в котором находится склад. Помещениям склада соответствуют таблицы реляционной базы данных, а стеллажам - строки таблицы.

Для работы с базами данных создаются *информационные системы*, состоящие из прикладных программ, СУБД и базы данных. Пользователи, как правило, называют базой данных всю информационную систему. Такая терминологическая путаница очень мешает в ситуациях, когда пользователь пытается рассказать разработчику о неполадках в информационной системе.

Так как все строки таблицы имеют одинаковую структуру, то для создания таблицы достаточно описать структуру одной строки. Связь между таблицами задаётся с помощью потенциальных и внешних ключей. Ключ,

состоящий из одного поля, описывается в структуре строки, а составной ключ - отдельно в структуре таблицы.

Описание структуры базы данных называется *метаданными* и хранится в базе вместе с основными данными.

Нотация Бэкуса-Наура

Нотация, или форма, Бэкуса-Наура используется для описания синтаксиса операторов языков программирования. В ней применяются следующие обозначения:

- | - *ИЛИ*, например *день | ночь* (читается: *день или ночь*);
- ::= - *это есть*, например, *время года ::= зима | весна | лето | осень*;
- [ ] - выражение в квадратных скобках не обязательно;
- { } - выражение в фигурных скобках рассматривать как единое целое;
- ... - предшествующая многоточию часть оператора может быть повторена любое число раз;
- .,.. - точка, запятая, двоеточие указывают, что предшествующая часть оператора может быть повторена любое число раз через запятую, причём, после последнего повторения запятая не ставится.

Более полно смысл перечисленных обозначений станет ясен после их использования для описания операторов.

Операторы языка описания данных

Перечислим основные операторы языка SQL, с помощью которых создаётся и изменяется структура базы данных:

- CREATE TABLE - создание структуры таблицы;
- ALTER TABLE - изменение структуры таблицы;
- DROP TABLE - удаление структуры таблицы;
- CREATE VIEW - создание представления;
- DROP VIEW - удаление представления;
- CREATE INDEX - создание индекса (не входит в стандарт);
- DROP INDEX - удаление индекса;
- GRANT - предоставление привилегий (прав) пользователям;
- REVOKE - удаление привилегий;
- CREATE SCHEMA - задание последовательности операторв
- DROP SCHEMA - удаление схемы.

Часть языка SQL, в которой используются эти операторы, называют языком описания данных (DDL - Data Definition Language).

Оператор CREATE TABLE

Создание таблицы — это запоминание в базе данных структуры таблицы, описанной в операторе *CREATE TABLE*. Запоминание происходит в результате выполнения этого оператора. Рассмотрим простейший пример создания таблицы.

Пример. Создание таблицы из четырёх столбцов.

```
CREATE TABLE knigi
(naim CHAR (100),
kvo_str INTEGER,
god_izd SMALLINT,
avtor CHAR (60)
);
```

Обозначения:

knigi - имя таблицы;

naim, kvo_str, god_izd, avtor - имена полей (столбцов);

INTEGER, SMALLINT, CHAR (100), CHAR (60) - типы данных

100, 60 - длина поля в символах.

SQL - язык свободной формы. Это значит, что можно делать перенос на следующую строку в любом месте оператора, только переносить нужно целыми словами (лексемами). В примере каждое поле описано в отдельной строке. В конце оператора можно, но не обязательно, ставить точку с запятой. Опишем в нотации Бекуса-Наура таблицу, подобную приведённой в примере, только с любым количеством строк:

CREATE TABLE *имя таблицы*

({*{имя столбца} тип*[(*длина*])} .,..)[;]

Эта синтаксическая формула описывает простую структуру таблицы. В ней большими буквами написаны ключевые слова CREATE и TABLE языка SQL, а курсивом - переменные *имя столбца, тип, длина, ограничение столбца* и *ограничение таблицы*, на место которых подставляются их значения. Переменная *имя столбца* заключена в фигурные скобки для того, чтобы отделить её от переменной *тип*. Полная синтаксическая формула оператора CREATE TABLE имеет вид:

CREATE [{GLOBAL | LOCAL} TEMPORARY] TABLE *имя таблицы*

({{{*имя столбца} тип*[(*длина*)] [ограничение столбца.,]} |

[ограничение таблицы]

} .,..

[ON COMMIT {DELETE | PRESERVE} ROWS]

)[;]

Таблица может быть временной. Описание структуры временной таблицы хранится в базе данных постоянно, а данные удаляются, когда исчезает необходимость в них. Таблица объявляется временной ключевым словом *TEMPORARY*. Ключевое слово *GLOBAL* задаёт удаление данных в конце сеанса пользователя, *LOCAL* - по завершении программы или модуля. ON COMMIT DELETE ROWS - удаление данных по завершению транзакции. Если задано ON COMMIT PRESERVE ROWS, то данные во временной таблице по завершении транзакции сохраняются.

Ограничения столбца имеют следующие значения:

- NOT NULL - в столбце не допускается отсутствие значения;
- UNIQUE - все значения в столбце должны быть уникальными, но допускается NULL;
- PRIMARY KEY - действует так же, как UNIQUE, но NULL не допускается;
- CHECK (*условие*) - проверка условия
- DEFAULT =*выражение*- значение по умолчанию;
- REFERENCES *имя таблицы [(имя столбца)]* - внешний ключ. Имя столбца указывается, когда столбец не является первичным ключом.

Ограничение может иметь имя, которое задаётся перед ограничением ключевым словом CONSTRAINT:

CONSTRAINT *имя ограничения*,

например,

CONSTRAINT *znach* NOT NULL.

Ограничения таблицы:

- UNIQUE (*список столбцов)*) - все комбинации значения столбцов должны быть уникальными, но допускается NULL;
- PRIMARY KEY (*список столбцов*) действует так же, как UNIQUE, но NULL не допускается;
- CHECK (*условие*) - проверка условия
- FOREIGN KEY (*список столбцов*) *имя таблицы* [(*список столбцов*)] - внешний ключ. Список столбцов связанной таблицы указывается, когда он не является первичным ключом.

Пример создание двух связанных таблиц.

Рассмотрим сущности *писатель* и *книга*. Для простоты будем считать, что у книги может быть только один автор. Тогда между сущностями *писатель* и *книга* существует связь типа *один ко многим*. Для отображения этой связи в реляционной модели создадим в таблице *pisatel* первичный ключ *Id_p*, а в таблицу *kniga* добавим внешний ключ *Id_pisatel*. Кроме того, зададим адрес

писателя по умолчанию и проверку числа страниц в книге. Соответствующие запросы будут выглядеть так:

```
CREATE TABLE pisatel
(Id_p SMALLINT PRIMARY KEY,
FIO CHAR (60) NOT NULL UNIQUE,
adr CHAR (100) DEFAULT='Москва'
);

CREATE TABLE kniga
( naim CHAR(100) NOT NULL UNIQUE,
kvo_stranic INTEGER CHECK ((kvo_stranic<10000) AND (kvo_stranic > 0))
god_izd SMALLINT,
Id_pisatel SMALLINT REFERENCES pisatel
);
```

В таблице *kniga* поле *naim* = потенциальный ключ.

Пример составного первичного ключа. В таблице *tovar* пара значений *наименование, изготовитель* должна быть уникальной.

```
CREATE TABLE tovar
(наименование CHAR (50),
изготовитель CHAR (50),
цена MONEY,
PRIMARY KEY (наименование, изготовитель)
);
```

Пример составного потенциального ключа. Поле *Id* - первичный ключ, пара полей *наименование, изготовитель* - потенциальный ключ.

```
CREATE TABLE товар1
(Id SMALLINT PRIMARY KEY,
наименование CHAR (50) NOT NULL,
изготовитель CHAR (50) NOT NULL,
цена MONEY,
UNIQUE (наименование, изготовитель)
);
```

Пример составного внешнего ключа. Пара полей *изделие, фирма* таблицы *заказ* - внешний ключ. Эта пара связана с потенциальным ключом *наименование, изготовитель* таблицы *товар1*

```
CREATE TABLE заказ
(N_заказа INTEGER NOT NULL UNIQUE,
изделие CHAR (50),
```

фирма CHAR (60),
к_во SMALLINT CHECK ((к_во> 0) AND, (к_во <= 100)),
FOREIGN KEY (изделие, фирма) товар1(наименование, изготовитель)
);

Оператор ALTER TABLE

Оператор ALTER TABLE служит для изменения структуры существующих таблиц. В реляционной базе данных существует множество часто неявных, скрытых связей, которые при изменении структуры таблиц могут быть нарушены. Поэтому применять оператор ALTER TABLE следует крайне осторожно.

Синтаксическая формула оператора ALTER TABLE:
ALTER TABLE *имя таблицы*
{ADD[COLUMN] {*имя столбца*} *тип*[(длина)] [*ограничение* ...]}
|{ALTER [COLUMN} {*имя столбца*} {изменяющее действие}
|{DROP [COLUMN] {*имя столбца*} RESTRICT | CASCADE}
|{ADD *ограничение таблицы*}
|{DROP CONSTRAINT *имя ограничения* RESTRICT | CASCADE};

изменяющее действие ::= {SET DEFAULT *значение по умолчанию*} | {DROP DEFAULT}

Пример добавления столбца. К таблице *заказ* добавляется столбец *цена*, имеющий тип *MONEY*.

ALTER TABLE заказ ADD цена MONEY;

Оператор DROP TABLE

Синтаксическая формула оператора DROP TABLE:
DROP TABLE *имя таблицы*;

Пример

DROP TABLE kniga;

Оператор CREATE INDEX

Синтаксическая формула оператора CREATE INDEX:
CREATE [UNIQUE] INDEX *имя индекса*
ON *имя твблицы*(имя столбца .,..);

Пример. Содаётся индекс *kniga_ind* для таблицы *kniga*.

CREATE INDEX kniga_ind
ON kniga(naim);

Оператор DROP INDEX

Синтаксическая формула оператора DROP INDEX:
DROP INDEX *имя индекса*;

Пример

DROP INDEX kniga_ind;

Оператор CREATE VIEW

В контексте реляционных баз данных термин VIEW переводится на русский язык как *представление*. Синтаксическая формула оператора CREATE VIEW:

CREATE VIEW *имя таблицы* [(*столбец* .,..)]

(AS *оператор запроса SELECT*

[WITH [CASCADED | LOCAL] CHECK OPTION]);

Оператор *CREATE VIEW* необычен тем, что содержит в себе оператор *SELECT*, принадлежащий языку манипулирования данными. Заметьте, что в синтактической формуле используется не *имя представления*, что было бы логично, а *имя таблицы*. Когда в каком-либо запросе языка манипулирования данными встречается *имя таблицы*, объявленное в операторе *CREATE VIEW*, то выполняется запрос *SELECT*, объявленный в том же *CREATE VIEW*. Результаты этого запроса рассматриваются как обыкновенная таблица!

Данные могут извлекаться в представление из одной или из нескольких настоящих таблиц. Источником формирования представления может быть другое представление. В том случае, когда представление связано только с одной таблицей, оно может использоваться для изменения (оператор UPDATE), удаления (оператор DELETE) и добавления (оператор INSERT) данных в породившую его таблицу.

Предложение CHECK OPTION служит для проверки нарушения целостности данных при использовании представления в операторах INSERT и UPDATE. CASCADED распространяет проверку на все уровни вложенности представления, а LOCAL ограничивает проверку только одним уровнем.

Пример. Создаётся представление, в которое отбираются только дешёвые товары из таблицы *товар*

CREATE VIEW дешёвый_товар

(AS SELECT * FROM товар

WHERE цена <100);

Оператор DROP VIEW

Синтаксическая формула оператора удаления представления DROP VIEW:

DROP VIEW *имя представления* RESTRICT | CASCADE

RESTRICT вызывает сообщение об ошибке при существовании ссылки на это представление.

При задании CASCADE удаляются все объекты, в которых есть ссылки на удаляемое представление.

Оператор GRANT

Оператор GRANT служит для назначения прав (привилегий) пользователям. Синтаксическая формула оператора GRANT:

GRANT *привилегия* .,.. ON *имя таблицы*

TO {*имя пользователя, на которого распространяется привилегия*}.,..

| PUBLIC

[WITH GRANT OPTION]

привилегия::= {ALL PRIVILEGIES}

| { SELECT

| DELETE

| INSERT [({*имя столбца*} .,..)]

| UPDATE [({*имя столбца*} .,..)]

| REFERENCES [({*имя столбца*} .,..)]

}

REFERENCES::= право создавать внешние ключи

GRANT OPTION даёт возможность передавать права другим пользователям

PUBLIC предоставляет указанные в операторе GRANT права всем пользователям.

Примеры

1. Пользователю Петрову предоставляется право добавлять данные в таблицу *книга*.

GRANT INSERT ON книга TO Петров

WITH GRANT OPTION;

2. Пользователям Lada и Genja предоставляется право просматриваь таблицу *книга* и добавлять в неё данные.

GRANT SELECT, INSERT ON книга TO Lada,Genja;

3. Пользователям Andre и Peter предоставляется право обновлять поле naim в таблице *книга*.

GRANT UPDATE (naim) ON книга TO Andre,Peter;

Оператор REVOKE

Оператор REVOKE служит для отмены привилегий. Синтаксическая формула оператора REVOKE:

REVOKE {ALL [PRIVILEGES]

| *привилегия* .,.. [ON *имя таблицы*]

FROM {PUBLIC | {*имя пользователя*}

DML - язык манипулирования данными

Создавать и изменять структуру базы даных приходится довольно редко. Кроме того, практически для каждой СУБД имеется программа-оболочка, позволяющая автоматизировать процесс работы со структурами баз данных. Пример такой оболочки - phpMyAdmin для СУБД MySQL. Можно сказать, что СУБД ACCESS целиком упрятана от пользователя в оболочку. Во много раз чаще производится работа с содержимым базы данных, т. е. с данными. При работе с базой данных в основном используются четыре оператора языка манипулирования данными:

- SELECT - оператор выборки данных;
- INSERT - оператор добавления данных;
- UPDATE - оператор обновления данных;
- DELETE - оператор удаления данных

Оператор SELECT - особый по частоте использования и многообразию возможных форм не только в SQL, но и вообще в программировании. В шутку SQL иногда называют языком одного оператора SELECT.

В состав оператора языка SQL могут входить другие операторы языка SQL, математические операторы (арифметические, логические, операторы отношения) и функции, строковые операторы и функции.

Арифметические

\+ - сложение;

\- - вычитание;

* - умножение;

/ - деление;

** - возведение в степень.

Логические

NOT - отрицание;

AND - логическое умножение (конъюнкция);

OR - логическое сложение (дизъюнкция).

Операторы отношения

= Равно

> Больше чем

<Меньше чем

>= Больше или равно

<= Меньше или равно

<> Не равно

Операция над данными строкового типа

+ конкатенация (объединение, слияние двух строк)

Конкатенация данных строкового типа и сложения числовых данных обозначаются одним значком "+", но дают разный результат, например,

2 + 3 = 5 - данные числового типа;

"2" + "3" = "23" - данные строкового типа.

Оператор SELECT

Общая синтаксическая формула оператора SELECT очень сложна и мало подходит для начинающего изучать язык SQL. Поэтому начнём с самых простых частных случаев.

Выборка всех строк

Запрос на выборку всей таблицы имеет самый простой вид:

SELECT * FROM *имя таблицы*;

* - все колонки

Пример. Из базы выбирается вся таблица *заказ*.

Таблица *заказ*			
N_заказа	**изделие**	**фирма**	**к_во**
1	Ноутбук	Альфа	2
2	Мышка	Бета	4
3	Принтер	Альфа	1
4	флешка	Гамма	5
5	Мышка	Бета	1

Запрос

SELECT * FROM заказ;

Результат запроса - исходная таблица.

Выборка списка столбцов. Часто требуется выбрать из таблицы только часть столбцов. В этом случае вместо символа * выводимые столбцы перечисляются в нужном порядке через запятую.

SELECT {имя столбца} .,.. FROM *имя таблицы*;

Пример.

SELECT изделие, фирма FROM заказ;

Результат запроса

изделие	**фирма**
Ноутбук	Альфа
Мышка	Бета
Принтер	Альфа
флешка	Гамма
Мышка	Бета

Из примера видно, что реальная таблица реляционной базы данных в отличие от теоретического отношения может содержать одинаковые строки. Чтобы исключить дублирование строк, нужно вставить DISTINCT после SELECT:

SELECT DISTINCT изделие, фирма FROM заказ.

Результат запроса

изделие	**фирма**
Ноутбук	Альфа
Мышка	Бета
Принтер	Альфа
флешка	Гамма

В СУБД режим DISTINCT может быть установлен по умолчанию. Чтобы выводились все строки в таком случае, нужно указать *SELECT ALL ...*

Псевдонимы. Названия столбцов часто неудобны при просмотре результатов запроса. Для замены имён столбцов синонимами используется следующий синтаксис:

SELECT {{имя столбца} [AS псевдоним]} .,.. FROM *имя таблицы*;

Пример.

SELECT N заказа AS 'Номер заказа', изделие, фирма, к_во AS 'количество'

FROM заказ;

Результат запроса

Номер заказа	**изделие**	**фирма**	**количество**
1	Ноутбук	Альфа	2
2	Мышка	Бета	4
3	Принтер	Альфа	1
4	флешка	Гамма	5
5	Мышка	Бета	1

Вычисляемые поля. Вместо имени поля можно использовать арифметическое или строковое выражение. Подсчитаем стоимость товаров в таблице *затраты*, изменив единицу измерения стоимости на *тыс. руб.*

Таблица *Затраты*		
Товар	**Цена**	**к_во**
Стол	12000	5

Стул	1700	20
Шкаф	18500	2

Запрос выглядит так:

SELECT Товар,цена,к_во AS Количество,str(цена*к_во/1000)+' тыс. руб.' AS Стоимость

В запросе использована функция str(числовое выражение), преобразующая выражение в строковый тип. Результат запроса

Товар	**Цена**	**Количество**	**Стоимость**
Стол	12000	5	60 тыс. руб.
Стул	1700	20	34 тыс. руб.
Шкаф	18500	2	37 тыс. руб.

Агрегатные функции служат для вычисления характеристик таблицы в целом. Всего агрегатных функций пять: count, sum, max, min и avg.

count(*) вычисляет количество строк в таблице.

count(имя поля) - количество строк с непустым полем(не NULL), указанным в аргументе;

выражение:: = {имя поля} | {арифметическое выражение, вычисляемое для каждой строки таблицы};

sum(выражение) вычисляет сумму значений выражения во всех строках таблицы;

max(выражение) возвращает максимальное значение выражения;

min(выражение) возвращает минимальное значение выражения;

avg(выражение) возвращает среднее значение выражения.

Пример.

SELECT count (*) AS 'К-во товаров',
sum(цена*к_во) AS 'Стоимость всех товаров'
max(цена) AS 'Макс. цена',
min(цена) AS 'Мин. цена',
avg(цена) AS 'Средняя цена',
FROM Затраты;

Результат запроса

К-во товаров	**Стоимость всех товаров**	**Макс. цена**	**Мин. цена**	**Средняя цена**
3	131000	18500	1700	10733

Выборка строк, удовлетворяющих условию

Для выборки из таблицы только тех строк, содержимое которых удовлетворяет заданному условию, дополним синтактическую формулу предложением WHERE:

SELECT {{имя столбца} [AS псевдоним]}.,..

FROM *имя таблицы*

WHERE *условие*;

условие::= предикат

предикат::= выражение, принимающее значение *истина* или *ложь*

Примеры.

1. *Использование операций сравнения.*

Выберем из таблицы *Затраты* строки, в которых цена больше 5000.

SELECT * FROM Затраты WHERE цена > 5000;

Результат запроса

Товар	Цена	Количество
Стол	12000	5
Шкаф	18500	2

В этом примере *цена* имеет числовой тип. Можно сравнивать и текстовую (строковую) информацию. Вся текстовая информация хранится в памяти ЭВМ в виде последовательности цифровых кодов символов. Таблицы кодирования составлены так, что следующая буква алфавита имеет больший код, чем предыдущая, то есть, А <Б <В < ... < Я. Чтобы выбрать из таблицы *клиенты* все строки с фамилиями, начинающимися на буквы с А по И включительно, нужно выполнить следующий запрос

SELECT * FROM клиенты WHERE ФИО < К.

Часто, особенно в экономических задачах, требуется выбирать данные за заданный диапазон дат. Даты для того, чтобы над ними легко производить арифметические действия, хранятся в памяти ЭВМ в виде числа секунд, прошедшего с некоторого фиксированного момента времени, например с 1900-го года. Когда говорят "тип DATE", то имеют в виду дату в виде числа секунд. В СУБД обычно имеется набор функций для работы с датами.

Рассмотрим два примера использования в СУБД ACCESS функций *CDATE* и *DatePart*.

Функция CDATE (*дата в текстовом виде*) преобразует дату в тип DATE. Выберем из таблицы *студенты* все строки с датами рождения позже 31.12.1994.

SELECT * FROM студенты WHERE дата_рождения > CDATE('31.12.1994');

Функция *DatePart(часть даты, дата)* возвращает часть даты. Параметр *часть даты* может принимать следующие значения:

yyyy год
q квартал
m месяц
y день от начала года
d день от начала месяца
w день недели
ww неделя от начала года
h часы
n минуты
s секунды

Пример выделения из даты года и месяца. Из таблицы *клиенты* выбираются *ФИО* и *Год рождения* клиентов, родившихся в мае.

Таблица *клиенты*	
ФИО	**Дата рождения**
Сергеева Е. И.	16.02.1994
Петров А. П.	16.05.1995
Петров В. П.	07.10.1989
Иванов С. В.	21.05.1956

Запрос

```
SELECT ФИО,DatePart('yyyy',дата_рождения) AS 'Год рождения'
FROM клиенты
WHERE DatePart('m',дата_рождения)=5;
```

Результаты запроса

ФИО	**Год рождения**
Петров А. П.	1995
Иванов С. В.	1956

2. *Применение логических операторов AND, OR, NOT*

Выберем из таблицы *клиенты* все клиентов, родившихся между 1990-м и 1999-м годами или родившихся не в мае.

Запрос

SELECT * FROM клиенты

WHERE DatePart('yyyy’, дата_рождения)>=1990

AND DatePart('yyyy’, дата_рождения)<=1999

OR NOT DatePart('m’, дата_рождения)=5;

Результаты запроса

ФИО	**Дата рождения**
Сергеева Е. И.	16.02.1994
Петров А. П.	16.05.1995
Петров В. П.	07.10.1989

3. *Применение специальных операторов сравнения IN, BETWEEN, LIKE, IS NULL*

Перечисленные операторы иначе называют предикатами. Предикаты IN и BETWEEN служат для сокращения записи условия выборки.

Предикат IN используется в тех случаях, когда в условиях выборки нужно задать не диапазон, а список значений. Выберем из таблицы *Заказы* заказы для городов БЕРН, Женева и Грасс.

SELECT * FROM Заказы

WHERE ГородПолучателя **IN ('БЕРН', 'Женева','Грасс')**;

Предикат BETWEEN, наоборот, используется для задания диапазона значений, включая границы диапазона. Выберем заказы с ценой от 200 до 1000.

SELECT * FROM Заказы

WHERE цена **BETWEEN 200 AND 1000**;

При задании диапазона дат необходимо преобразовать даты из текстового типа в тип DATE'.

SELECT * FROM Заказы

WHERE Дата Размещения BETWEEN CDATE ('01.03.1997') AND CDATE ('30.06.1997')

Можно задавать текстовый диапазон. В результате выполнения запроса

SELECT * FROM Заказы

WHERE СтранаПолучателя **BETWEEN 'А' AND 'К'**;

выберутся заказы для стран на буквы от А до И включительно.

Предикат LIKE служит для поиска текстовой информации по шаблону. Заменим в предыдущем операторе предикат BETWEEN на LIKE, используя шаблоны, принятые в СУБД ACCESS.

SELECT * FROM Заказы

WHERE СтранаПолучателя **LIKE '[А-И] *'**;

Шаблон [А-И] * означает, что нужно найти название страны, начинающееся на любую букву от А до И включительно. При поиске шаблон как бы накладывается на сравниваемый текст. В приведённом примере [А-И] накладывается на первую букву, а * означает что остальная часть текста может быть любой. Если текст подходит под шаблон, то результат сравнения принимает значение *истина*, иначе - *ложь*. В квадратных скобках можно указать любой перечень символов. Для поиска стран Россия, Белоруссия, Казахстан можно указать шаблон [БКР][аео]*[ня]. Задание точного шаблона, в общем случае, трудная задача. В примере кроме России, Белоруссии, Казахстана будет найдена и Болгария. Читателю представляется возможность уточнить этот шаблон.

В стандарте SQL есть только два символа для поиска по шаблону:

_ (подчёркивание) - один любой символ; ему соответствует
символ "?" в ACCESS;

% - любая последовательность любых символов;
ему соответствует символ "*" в ACCESS;

Примеры шаблонов в ACCESS

Шаблон	Соответствующий текст
?аша	Маша, Даша или Саша
л [! аи]па	лупа, но не лапа и не липа
п??	поп, пол, пел
п?л	пол, пел, пал, пил, пул

Если пользователи ничего не записали в какое-либо поле, то считается, что оно хранит признак пустоты *NULL*. Для того, чтобы найти строки, в которых заданное поле хранит NULL, нужно написать условие

WHERE *имя поля* IS NULL

Для задания обратного условия нужно написать

WHERE *имя поля* IS NOT NULL

Предложения GROUP BY и HAVING

Предложение GROUP BY служит для разбиения всех строк таблицы на группы и последующего применения к каждой группе агрегатных функций. Например, для фирмы, занимающейся доставкой товаров, представляет интерес не только общее количество заказов, но и распределение заказов по странам. Такие данные получаются в результате выполнения запроса

SELECT Страна Получателя, count (*) AS 'Количество заказов',

sum (Стоимость Доставки) AS 'Суммарная стоим доставки',

max (Стоимость Доставки) AS 'Макс стоим доставки'
FROM Заказы
GROUP BY Страна Получателя;

Результаты запроса

Страна получателя	Количество заказов	Сумм стоим доставки	Макс стоим доставки
Австрия	39	6 637,24р.	789,95р.
Аргентина	16	598,58р.	217,86р
Бельгия	20	1 341,89р.	424,30р.
.....	.....	.....	.....
Франция	80	4 276,20р.	487,38р.
Швейцария	18	1 368,53р.	232,42р.
Швеция	36	2 992,81р.	328,74р.

Предложение HAVING служит для отбора групп, удовлетворяющих условию. В условие могут входить агрегатные функции. Перепишем предыдущий запрос, включив в него условие отбора только тех стран, максимальная стоимость доставки в которые больше четырёхсот рублей.

SELECT СтранаПолучателя, count(*) AS 'Количество заказов',
sum(СтоимостьДоставки) AS 'Сумм стоим доставки',
max(СтоимостьДоставки) AS 'Макс стоим доставки'
FROM Заказы
GROUP BY Страна Получателя
HAVING max(Стоимость Доставки)>400 ;

Результаты запроса

Страна получателя	Количество заказов	Сумм стоимости доставки	Макс стоимость доставки
Австрия	39	6 637,24р.	789,95р.
Бельгия	20	1 341,89р.	424,30р.
Бразилия	83	4 880,19р.	890,78р.
Германия	117	11 341,09р.	1 007,64р.
Ирландия	19	2 755,24р.	603,54р.
США	122	13 771,29р.	830,75р.
Франция	80	4 276,20р.	487,38р.

Общая структура однотабличного запроса имеет вид:

SELECT *список вывода*
FROM *имя таблицы*

WHERE *условие для отбора строк из всей таблицы*

GROUP BY *выражение для разбиения на группы*

HAVING *условие для отбора групп*

ORDER BY *список имен или номеров полей для сортировки*

Ещё раз усложним запрос к таблице *Заказы*. Отберём из неё только те заказа, которые доставлялись почтой (код почты - 2), и отсортируем полученные результаты по количеству заказов в порядке убывания:

```
SELECT СтранаПолучателя, count(*) AS 'Количество заказов',
sum(СтоимостьДоставки )  AS 'Сумм стоим доставки',
max (СтоимостьДоставки) AS 'Макс стоим доставки'
FROM Заказы
WHERE Доставка = 2
GROUP BY СтранаПолучателя
HAVING max(СтоимостьДоставки)>400
ORDER BY  count(*) DESC
```

<="" pre="" style="margin-left: 24px; text-indent: 20px; font-size: 14pt; text-align: justify; margin-bottom: 5px; margin-top: 5px;">

Результаты запроса

Страна получателя	Количество заказов	Сумм стоим доставки	Макс стоим доставки
Германия	53	4 148,10р.	810,05р.
США	51	5 778,58р.	830,75р.
Бразилия	35	2 513,43р.	890,78р.
Австрия	14	2 311,57р.	789,95р.
Ирландия	9	1 890,41р.	603,54р.
Бельгия	9	793,16р.	424,30р.

Предложение ORDER BY

Предложение ORDER BY служит для указания способа сортировки результатов запроса. Оно должно быть последним в запросе. Сортировать можно по одному или нескольким полям. Можно указывать псевдонимы имён полей или номера полей по порядку в списке вывода. Сортировка по возрастанию обычно устанавливается в СУБД по умолчанию. Для явного задания способа сортировки служат два ключевых слова: ASC - по возрастанию и DESC - по убыванию. В последнем рассмотренном запросе возможны три варианта указания сортировки по убыванию:

```
ORDER BY count (*) DESC
```

ORDER BY 'Количество заказов' DESC

ORDER BY 2 DESC

Многотабличные запросы

На практике часто с помощью одного запроса данные собираются сразу из нескольких таблиц. Некоторые СУБД позволяют в одном запросе делать выборку из нескольких баз данных.

Рассмотрим простейший двухтабличный запрос. Необходимо выбрать из базы данных названия всех городов вместе с названиями стран, в которых они находятся. Для этого используем связанные между собой таблицы *Города* и *Страны*

Таблица *Города*

Название	Население	кодСтраны
Москва	10 000 000	1
Новгород	1 300 000	1
С.-Петербург	5 300 000	1
С.-Петербург	250 000	3
Турку	180 000	2
Хельсинки	600 000	2
Лондон	370 000	

Таблица *Страны*

кодСтраны	Название
1	Россия
2	Финляндия
3	США
4	Чили

В таблице *Города* специально не указана страна для Лондона, хотя по населению можно догадаться, что это не столица Англии. Стране *Чили* не соответствует ни один город.

Запрос

SELECT a.Название AS Город, a.Население,b.Название AS Страна

FROM Города a, Страны b

WHERE a.кодСтраны=b.кодСтраны

Результат

Город	Население	Страна
Москва	10 000 000	Россия
Новгород	1 300 000	Россия
С.-Петербург	5 300 000	Россия
С.-Петербург	250 000	США
Турку	180 000	Финляндия
Хельсинки	600 000	Финляндия

В запросе для сокращения записи использованы псевдонимы имён таблиц. Связь между таблицами задаётся в предложении *WHERE*. В одной строке таблицы результатов объединяется строка с внешним ключом (город)

со строкой с равным первичным ключом. Лондон не вошёл в таблицу результатов, так как у него не указан внешний ключ.

Тот же результат получается другим способом, который называется *INNER JOIN - внутреннее соединение*:

SELECT a.Название AS Город, a.Население,b.Название AS Страна

FROM (Города a) INNER JOIN (Страны b) ON a.кодСтраны=b.кодСтраны

Если в этом запросе заменить INNER на LEFT (левое внешее соединение), то будут выбраны все строки, обеих таблиц, удовлетворяющие условию соединения, и из левой таблицы строки, которым нет соответствия в правой таблице. В нашем примере к таблице результатов добавится одна строка.

Запрос с *LEFT JOIN*

SELECT a.Название AS Город, a.Население,b.Название AS Страна

FROM (Города a) LEFT JOIN (Страны b) ON a.кодСтраны=b.кодСтраны

Результат

Город	Население	Страна
Москва	10 000 000	Россия
Новгород	1 300 000	Россия
С.-Петербург	5 300 000	Россия
С.-Петербург	250 000	США
Турку	180 000	Финляндия
Хельсинки	600 000	Финляндия
Лондон	370 000	

Правое внешнее соединение (RIGHT JOIN) аналогично левому, только кроме строк, удовлетворяющих условию, выбираются все оставшиеся строки правой таблицы *Страны*.

Запрос с *RIGHT JOIN*

SELECT a.Название AS Город, a.Население,b.Название AS Страна

FROM (Города a) RIGHT JOIN (Страны b) ON a.кодСтраны=b.кодСтраны

Результат

Город	Население	Страна
Москва	10 000 000	Россия
Новгород	1 300 000	Россия
С.-Петербург	5 300 000	Россия
С.-Петербург	250 000	США

Турку	180 000	Финляндия
Хельсинки	600 000	Финляндия
		Чили

При большом количестве таблиц запрос получается очень сложным, в нём легко сделать ошибку, поэтому целесообразно из нескольких способов формирования запроса, дающих одинаковый результат, выбирать наиболее простой. Ниже приведены два запроса к пяти таблицам, дающие одинаковый результат.

Запрос, полученный с помощью конструктора СУБД ACCESS. Используется внутреннее соединение INNER JOIN.

```
SELECT Клиенты.Название, Клиенты.ОбращатьсяК, Товары.КодТовара,
Товары.Марка, Сотрудники.Фамилия
FROM (((Заказано INNER JOIN Заказы ON
Заказано.КодЗаказа=Заказы.КодЗаказа)
INNER JOIN Клиенты ON Заказы.КодКлиента=Клиенты.КодКлиента)
INNER JOIN Сотрудники ON
Заказы.КодСотрудника=Сотрудники.КодСотрудника)
INNER JOIN Товары ON Заказано.КодТовара=Товары.КодТовара
WHERE (((Клиенты.Название) Like 'F*')
    AND ((Сотрудники.Фамилия)='Кротов'));
```

Запрос, полученный "вручную". Связи между таблицами заданы в предложении WHERE.

```
SELECT a.Название,a.ОбращатьсяК,b.КодТовара,b.Марка,c.Фамилия
FROM Клиенты a,Товары b, Сотрудники c,Заказано d,Заказы e
WHERE  e.КодЗаказа=d.КодЗаказа
AND e.КодКлиента=a.КодКлиента
AND e.КодСотрудника=c.КодСотрудника
AND d.КодТовара=b.КодТовара
AND a.Название Like 'F*' AND c.Фамилия='Кротов';
```

Число связей между таблицами на единицу меньше числа таблиц в запросе. Для уменьшения вероятности ошибки при составлении запроса следует проверять это условие.

Объединение таблицы с самой собой применяется в тех случаях, когда в одной строке результатов запроса требуется выводить данные из разных строк исходной таблицы. В SQL-запросах можно объединять в одной строке данные только из разных таблиц. Поэтому для объединения данных из разных

строк исходной таблицы нужно представить в запросе исходную таблицу дважды, присвоив ей два псевдонима.

Пример. Требуется в таблице *Сотрудники* найти всех программистов и их начальников. Известно, что программисты работают почти во всех отделах.

Таблица *Сотрудники*		
ФИО	**должность**	**отдел**
Алесандрова О.Ю.	экономист	2
Андреев А.Б.	Программист	2
Алесеева К.А.	Менеджер	1
Борисова Г.П.	Программист	2
Котова Н.В.	Секретарь	2
Новикова А.Г.	Начальник	1
Смирнов П.С.	Программист	1
Петров К.Ю.	Начальник	2

Запрос

```
SELECT a.ФИО AS программист,b.ФИО AS начальник_отдела
FROM Сотрудники a,Сотрудники b
WHERE a.должность=’программист’ AND b.должность=’нач_отдела’
      AND a.отдел=b.отдел
```

Результат

Программист	**Начальник_отдела**
Андреев А. Б.	Петров К. Ю.
Борисова Г. П.	Петров К. Ю.
Смирнов П. С.	Новикова А. Г.

Подзапросы. С помощью SQL можно вкладывать один запросы в другой. Внутренний запрос называют подзапросом. Обычно, внутренний запрос генерирует значение, которое проверяется в предикате внешнего запроса, определяющего верно оно или нет. Например, в следующем запросе выбираются из таблицы *Товары* те товары, цена которых меньше средней цены всех товаров таблицы:

```
SELECT*
FROM Товары
WHERE Цена< (SELECT AVG(Цена) FROM Товары);
```

В подзапросе может использоваться любая таблица.

Предикат EXISTS имеет синтаксис

EXISTS подзапрос

и принимает значение ИСТИНА (TRUE), если подзапрос содержит хотя бы одну строку. В следующем запросе выбираются фамилии всех сотрудников, оформлявших заказы для клиента ANTON, при условии, что хотя бы один заказ для клиента ANTON был размещён в мае любого года.

```
SELECT DISTINCT b.Фамилия
FROM Заказы a, Сотрудники b
WHERE EXISTS (SELECT *
FROM Заказы
WHERE Код Клиента='ANTON' AND DatePart('m',ДатаРазмещения)=5)
AND a.Код Сотрудника = b.Код Сотрудника AND a.КодКлиента='ANTON';
```

Рассмотрим запрос, в котором в основном запросе и подзапросе используются разные таблицы. Нужно найти всех клиентов из Лондона, если есть хотя бы один заказ из Лондона.

```
SELECT * FROM Клиенты
WHERE EXISTS
(SELECT * FROM Заказы
WHERE Город Получателя = 'Лондон'
        )
AND Город = 'Лондон';
```

Предикаты количественного сравнения ANY, SOME и ALL имеют синтаксис

оператор сравнения {ANY | SOME | ALL} подзапрос.

ANY и SOME – синонимы.

Пример использования предиката ANY

```
SELECT Код Заказа
FROM Заказы
WHERE Стоимость Доставки <ANY (SELECT Стоимость Доставки
FROM Заказы
WHERE Город Получателя='Ванкувер');
```

Применение предиката ANY может дать неожиданный результат. Выберем из таблицы *детали* все детали, длина которых больше длины любой (ANY) детали, ширина которой равна семи.

Таблица *детали*		
шифр	**длина**	**ширина**
А2	10	2
А3	8	7
Б12	20	10
Б14	14	7
В17	3	2
В20	40	12
Г1	9	7

Запрос

SELECT *

FROM ANYдетали

WHERE длина>ANY (SELECT длина FROM детали WHERE ширина=7)

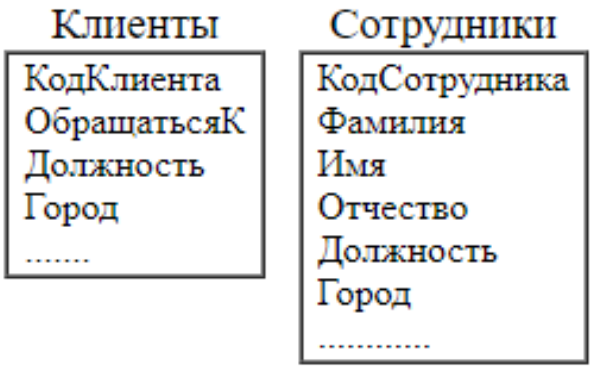
Результат запроса

шифр	**длина**	**ширина**
А2	10	2
Б12	20	10
Б14	14	7
В20	40	12
Г1	9	7

Детали с шириной, равной 7, имеют длины 8, 14 и 9. В результате запроса должна быть выбрана *любая* деталь, длинее 14, но выбраны детали с длиной, большей *наименьшей* из списка (8, 9, 14). Причина в том, что в контексте данного запроса ANY переводится как *какого-нибудь, хотя бы одного* вместо ожидаемого значения *каждого, всех*. Чтобы выбрать детали с длиной больше любой из списка, нужно вместо ANY применить предикат ALL.

Предложение UNION

Предложение UNION применяется для объединения результатов нескольких запросов в одной выводимой таблице. Количество столбцов во всех запросах должно быть одинаковым и типы соответствующих столбцов должны быть сравнимыми. В следующем примере объединяются данные из таблиц *Клиенты* и *Сотрудники*. Таблицы имеют следующие поля



Запрос

```
SELECT ОбращатьсяК AS 'Фамилия, имя',Должность,Город, 'Клиент' AS 'Кто'
 FROM Клиенты
 WHERE Должность LIKE 'Пр*'
UNION
SELECT Фамилия+' '+имя AS 'Фамилия, имя',Должность,Город, 'Сотрудник' AS 'Кто'
FROM Сотрудники
WHERE Город LIKE 'М*'
ORDER BY 1 ASC;
```

Результаты запроса

Фамилия, имя	Должность	Город	Кто
Андреева Юлия	Программист	Тверь	Клиент
Антонов Константин	Провизор	Новгород	Клиент
Петров Владимир.	Менеджер	Минск	Сотрудник
. . . . .	. . . . .	. . . . .	. . . . .

Оператор INSERT

Оператор INSERT служит для добавления данных к таблице. Синтаксис оператора INSERT:

```
INSERT INTO имя таблицы>
[(имя столбца.,..)]
{VALUES (значение .,..).,..}
|выражение запроса
|{DEFAULT VALUES};
```

Список столбцов можно не указывать, если задаются значения всех полей таблицы. Если поле не указано, то ему присваивается значение по умолчанию.

Пример. Добавим столицу Чили 'Сантьяго' и российскую Тверь в таблицу *Города*, рассмотренную в разделе *Многотабличные запросы*:

```
INSERT INTO Города (Название, Население, код Страны)
VALUES
 ('Сантьяго',5100000,4),
 ('Тверь',420000,1)
;
```

Так как вводятся значения во все поля таблицы, то список полей можно опустить:

```
INSERT INTO Города
VALUES ('Сантьяго',5100000,4),
      ('Тверь',420000,1);
```

Оператор UPDATE

Оператор UPDATE служит для изменения (редактирования) данных в таблице.

Синтаксис оператора UPDATE:

UPDATE *имя таблицы*
SET {*имя столбца*={*выражение для вычисления значения*
|NULL |DEFAULT}}.,..
[WHERE *предикат*];

Примеры.

1. Уточним население в Сантьяго в таблице *Города*:

```
UPDATE Города
SET население=5128000
WHERE Название='Сантьяго';
```

2. Увеличим цену всех товаров на 10%

```
UPDATE Товары SET цена = цена*1.1;
```

Оператор DELETE

Оператор DELETE служит для удаление строк из таблицы. Синтаксис оператора DELETE:

DELETE FROM *имя таблицы*
[WHERE *предикат*]

Пример. Удалим из таблицы *Города* все города, расположенные в Финляндии:

```
DELETE FROM Города WHERE кодСтраны=2;
```

Если условие *WHERE* не указано, то удалится ВСЯ ТАБЛИЦА (точнее, всё содержимое таблицы).

3. NoSQL

4 основных типа хранилищ:

1. Хранилище «ключ-значение» — в нём есть большая хеш-таблица, содержащая ключи и значения. Примеры: Riak, Amazon DynamoDB;
2. Документно-ориентированное хранилище — хранит документы, состоящие из тегированных элементов. Пример: CouchDB;
3. Колоночное хранилище — в каждом блоке хранятся данные только из одной колонки. Примеры: HBase, Cassandra;
4. Хранилище на основе графов — сетевая база данных, которая использует узлы и рёбра для отображения и хранения данных. Пример: Neo4J.

База данных типа «ключ-значение»

Отсутствие схемы в базах данных «ключ-значение», например, Riak, — это как раз то, что вам нужно для хранения данных. Ключ может быть синтетическим или автосгенерированным, а значение может быть представлено строкой, JSON, блобом (BLOB, Binary Large Object, большой двоичный объект) и т. д.

Такие базы данных как правило используют хеш-таблицу, в которой находится уникальный ключ и указатель на конкретный объект данных. Существует понятие блока (bucket) — логической группы ключей, которые не группируют данные физически. В разных блоках могут быть идентичные ключи.

Производительность сильно вырастает за счёт кеширующих механизмов, которые работают на основе маппингов. Чтобы прочитать значение, вам нужно знать как ключ, так и блок, поскольку на самом деле ключ является хешем (блок + ключ).

В модели «ключ-значение» нет ничего сложного, так как реализовать её проще простого. Не лучший способ, если вам нужно только обновить часть значения или сделать запрос к базе данных.

Если поразмыслить о теореме CAP, то становится довольно очевидно, что такие хранилища хороши в плане доступности (Availability) и устойчивости к разделению (Partition tolerance), но явно проигрывают в согласованности данных (Consistency).

Пример: посмотрим на набор данных, представленных таблицей ниже. Здесь ключ — это название страны, а значение — список адресов в этой стране:

Key	Value
“India”	{“B-25, Sector-58, Noida, India – 201301}”
“Romania”	{“IMPS Moara Business Center, Buftea No. 1, Cluj-Napoca, 400606",City Business Center, Coriolan Brediceanu No. 10, Building B, Timisoara, 300011”}
“US”	{“3975 Fair Ridge Drive. Suite 200 South, Fairfax, VA 22033”}

База данных такого типа позволяет читать и записывать значения с помощью ключа следующим образом:

- Get(key) возвращает значение, связанное с переданным ключом;
- Put(key, value) связывает значение с ключом;
- Multi-get(key1, key2, ..., keyN) возвращает список значений, связанных с переданным ключами;
- Delete(key) удаляет запись для ключа из хранилища.

И хотя базы данных типа «ключ-значение» могут пригодиться в определённых ситуациях, они не лишены недостатков. Первый заключается в том, что модель не предоставляет стандартные возможности баз данных вроде атомарности транзакций или согласованности данных при одновременном выполнении нескольких транзакций. Такие возможности должны предоставляться самим приложением.

Второй недостаток в том, что при увеличении объёмов данных, поддержание уникальных ключей может стать проблемой. Для её решения необходимо как-то усложнять процесс генерации строк, чтобы они оставались уникальными среди очень большого набора ключей.

Riak и Dynamo от Amazon — самые популярные СУБД данных такого типа.

Документоориентированная база данных

Данные, представленные парами ключ-значение, сжимаются как хранилище документов схожим с хранилищем «ключ-значение» образом, с той лишь разницей, что хранимые значения (документы) имеют определённую структуру и кодировку данных. XML, JSON и BSON — некоторые из стандартных распространённых кодировок.

В следующем примере можно увидеть данные в виде «документа», который отображает названия определённых магазинов. Обратите внимание, что, хотя все три примера содержат местоположение, они отображают его по-разному:

```
{officeName: "3Pillar Noida",
```

{Street: "B-25", City: "Noida",
State: "UP", Pincode: "201301"}
}
{officeName: "3Pillar Timisoara",
{Boulevard: "Coriolan Brediceanu No. 10",
Block: "B, Ist Floor",
City: "Timisoara", Pincode: "300011"}
}
{officeName: "3Pillar Cluj",
{Latitude: "40.748328",
Longitude: "-73.985560"}
}

Одним из ключевых различий между хранилищем «ключ-значение» и документоориентированным является то, что последний включает метаданные, связанные с хранимым содержимым, что даёт возможность делать запросы на основе содержимого. Например, в указанном примере можно попробовать найти все документы, в которых «City» равно «Noida», что вернёт все документы, связанные с магазинами в этом городе.

Apache CouchDB — пример документоориентированной СУБД. CouchDB использует JSON для хранения данных, JavaScript в качестве языка запросов с использованием MapReduce и HTTP для API. Данные и отношения не хранятся в таблицах так, как в традиционных реляционных базах данных, а по сути являются набором независимых документов.

Тот факт, что такие базы данных работают без схемы, делает простой задачей добавление полей в JSON-документы без необходимости сначала заявлять об изменениях.

Couchbase и MongoDB — самые популярные документно-ориентированные СУБД.

Колоночная база данных

В колоночных NoSQL базах данных данные хранятся в ячейках, сгруппированных в колонки, а не в строки данных. Колонки логически группируются в колоночные семейства. Колоночные семейства могут состоять из практически неограниченного количества колонок, которые могут создаваться во время работы программы или во время определения схемы. Чтение и запись происходит с использованием колонок, а не строк.

В сравнении с хранением данных в строках, как в большинстве реляционных баз данных, преимущества хранения в колонках заключаются в быстром поиске/доступе и агрегации данных. Реляционные базы данных

хранят каждую строку как непрерывную запись на диске. Разные строки хранятся в разных местах на диске, в то время как колоночные базы данных хранят все ячейки, относящиеся к колонке, как непрерывную запись, что делает операции поиска/доступа быстрее.

Пример: получение списка заголовков нескольких миллионов статей будет трудоёмкой задачей при использовании реляционных баз данных, так как для извлечения заголовков придётся проходить по каждой записи. А можно получить все заголовки с помощью только одной операции доступа к диску.

Модель данных

- Колоночное семейство — структура, которая может легко группировать колонки и суперколонки;
- Ключ — постоянное имя записи. У ключей может быть разное количество колонок, поэтому база данных может расширяться неравномерно;
- Пространство ключей — определяет самый внешний уровень организации, как правило, имя приложения/базы данных.
- Колонка — имеет упорядоченный список элементов — кортежей с именами и значениями.

Самыми известными примерами являются Google BigTable и HBase с Cassandra, вдохновлённые BigTable.

BigTable представляет собой высокопроизводительное, сжатое и проприетарное хранилище данных от Google. У него есть следующие атрибуты:

- Разреженность — некоторые ячейки могут быть пустыми;
- Распределённость — данные разделены между многими узлами;
- Постоянство — хранится на диске;
- Многомерность — более 1 измерения;
- Сопоставление — ключ и значение;
- Отсортированность — сопоставления обычно не сортируются, но этот случай — исключение.

Двумерная таблица, состоящая из строк и колонок, является частью реляционной системы баз данных.

City	Pincode	Strength	Project
Noida	201301	250	20
Cluj	400606	200	15
Timisoara	300011	150	10
Fairfax	VA 22033	100	5

Эту таблицу можно представить в виде BigTable-сопоставления следующим образом:

```
{
    3PillarNoida: {
        city: Noida
        pincode: 201301
    },
    details: {
        strength: 250
        projects: 20
    }
}
{
    3PillarCluj: {
        address: {
            city: Cluj
            pincode: 400606
        },
        details: {
            strength: 200
            projects: 15
        }
    },
{
    3PillarTimisoara: {
        address: {
            city: Timisoara
            pincode: 300011
        },
        details: {
            strength: 150
            projects: 10
        }
    }
{
    3PillarFairfax : {
        address: {
            city: Fairfax
            pincode: VA 22033
        },
        details: {
            strength: 100
            projects: 5
        }
}
```

- Внешние ключи «3PillarNoida», «3PillarCluj», «3PillarTimisoara» и «3PillarFairfax» являются аналогами строк.
- «address» и «details» — колоночные семейства.
- В колоночном семействе «address» есть колонки «city» и «pincode».
- В колоночном семействе «details» есть колонки «strength» и «projects».

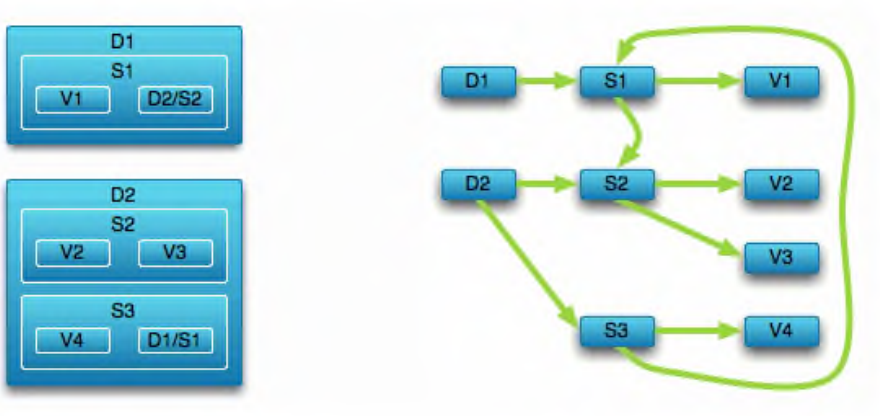
На колонки можно ссылаться с помощью колоночного семейства.

Графовая база данных

В графовой базе данных вы не найдёте строгого формата SQL или представления таблиц и колонок, вместо этого используется гибкое графическое представление, которое идеально подходит для решения проблем масштабируемости. Графовые структуры используются вместе с рёбрами, узлами и свойствами, что обеспечивает безиндексную смежность. При использовании графового хранилища данные могут быть легко преобразованы из одной модели в другую.

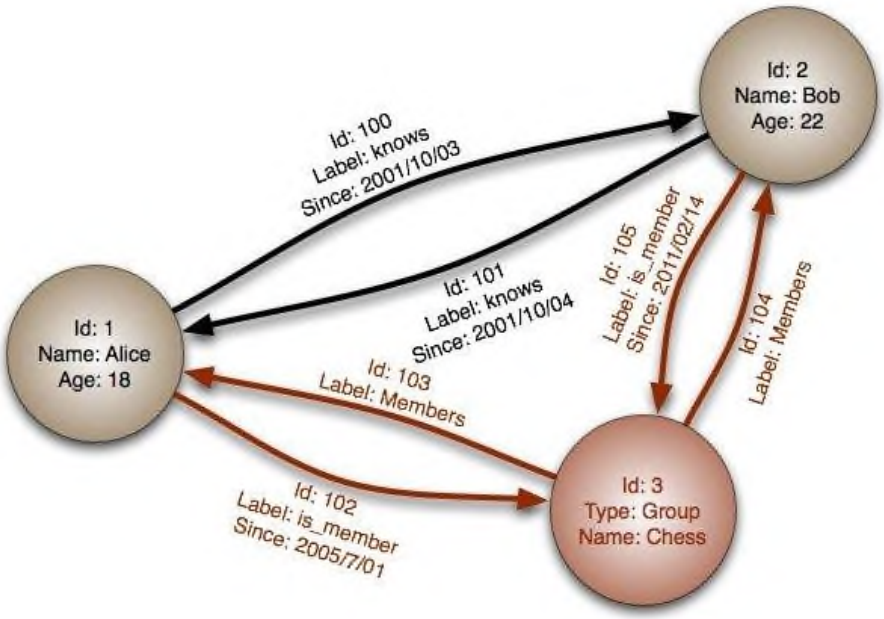


- Такие базы данных используют рёбра и узлы для представления данных.
- Узлы связаны между собой определённым отношениями, представленными рёбрами между ними.
- У узлов и отношений есть некоторые свойства.



Контейнерная иерархия документно-ориентированной базы данных содержит данные без схемы, которые можно представить в виде дерева, которое является графом. Если обращаться к документам или их элементам в этом дереве, можно получить более выразительное представление данных, в котором можно легко ориентироваться с помощью Neo4j.

Далее описаны некоторые особенности графовой базы данных на основе примера ниже:



Помеченный, направленный, атрибутированный мультиграф: граф содержит узлы, которые помечены определёнными свойствами и которые имеют связи друг с другом, что представлено направленными рёбрами. Например, связь «Элис знает Боба» выражена ребром с некоторыми свойствами.

Хотя реляционные базы данных могут скопировать поведение графовых, рёбрам потребуется соединение (JOIN), что дорого обойдётся.

Пример использования

Любой рейтинг «Рекомендовано вам», который можно увидеть на разных сайтах, зачастую составляется исходя из того, как другие пользователи оценили продукт. Графовые базы данных отлично подходят для такого случая.

InfoGrid и Infinite Graph — самые популярные графовые базы данных. InfoGrid позволяет соединять множество рёбер (Relationships) и узлов (MeshObjects), что упрощает представление набора информации со сложными взаимными ссылками.

InfoGrid предлагает два типа баз данных:

- MeshBase — подходящий вариант для автономного развёртывания;
- NetMeshBase — подойдёт для больших распределённых графов и имеет дополнительные возможности для взаимодействия с другими похожими NetMeshBase.

4. SQL

SQL — это структурированный язык запросов, созданный для того, чтобы получать из базы данных необходимую информацию. Если описать схему работы SQL простыми словами, то специалист формирует запрос и направляет его в базу. Та в свою очередь обрабатывает эту информацию, «понимает», что именно нужно специалисту, и отправляет ответ.

Данные хранятся в виде таблиц, они структурированы и разложены по строкам и столбцам, чтобы ими легче было оперировать. Такой способ хранения информации называют реляционными базами данных (от англ. *relation* — «отношения»). Название указывает на то, что объекты в такой базе связаны определенными отношениями.

Например, у маркетолога есть база, в которой собрана информация обо всех пиццериях в городе: названия, ассортимент, цены, график работы и прочее. Во время анализа конкурентов он решил выяснить, сколько пиццерий готовят пиццу с ананасами и оформляют доставку после 23:00. Для того чтобы получить такой список из базы, достаточно написать грамотный SQL-запрос.

Для чего нужен SQL

SQL — это не язык программирования, поэтому написать приложение или сайт с его помощью не получится, но при этом внутренняя работа сайта (backend) невозможна без запросов. Поиск информации в Google — это тоже

модель использования SQL. Пользователь задает параметры, которые его интересуют, и отправляет запрос на сервер; затем происходит магия и в поисковой выдаче появляются результаты, соответствующие именно этому запросу.

SQL используют разные виды специалистов:

- **Аналитики и продуктовые маркетологи.** Знание SQL помогает этим специалистам не зависеть от программистов, а самостоятельно получать и обрабатывать данные.
- **Разработчики и тестировщики.** С помощью SQL они могут самостоятельно проектировать базы для быстрой и надежной работы с данными, улучшать с их помощью сайты и приложения.
- **Руководители и менеджеры.** SQL позволит специалистам на руководящих постах самостоятельно обращаться к базам, контролировать работу компании и в реальном времени получать данные о положении дел.

Как работают запросы

Чтобы разобраться, как именно работает магия запроса, давайте представим его путь от пользователя до нужных ему данных:

Пользователь → Клиент → Запрос → Система управления → База данных → Таблица с базами данных

Данные для работы с SQL хранятся в таблицах. Как именно они устроены — разберемся ниже; пока же просто представим их. На пути от пользователя к таблице находится несколько посредников:

- **Клиент —** способ введения запроса. В случае с Google, например, клиентом будет поисковая строка браузера, в которую пользователь вводит сформулированный запрос.
- **Система управления базами данных (СУБД) —** комплекс программ, которые позволяют управлять данными. Эта система помогает таблицам понять, чего хочет пользователь, а пользователю — что ему отвечают таблицы.
- **База данных —** система хранения таблиц, в которой они связаны между собой. База данных сама по себе не умеет манипулировать информацией — это просто хранилище, где у каждого объекта есть свое место.

Что такое база данных в SQL

SQL-запросы обращаются к данным в виде таблиц, то есть к реляционным базам данных. Упрощенный вариант такой базы — это таблицы Excel, в которых информация также упорядочена в столбцы и строки.

Основные понятия реляционной модели:

1. **Отношение —** это сама таблица, она двумерная и состоит из столбцов и строк.

2. **Атрибут —** столбец в таблице, который содержит один конкретный параметр: название, тип, дату, стоимость или другую характеристику.

ID	Название	Доставка	Начало работы	Закрытие
1	Papa Jones	есть	8:00	22:00
2	Dominos	есть	8:00	23:00
3	Pizza Hut	нет	9:00	0:00
4	Dodo Pizza	есть	8:00	0:00
5	American Hut Pizza	нет	12:00	23:00

3. **Домен —** это допустимые значения для каждого атрибута. Например, в столбце «Имя» или «Название» значения должны представлять собой набор буквенных символов, но они не могут начинаться с «ь» или «ъ» и не могут быть записаны числами.

4. **Кортеж (строка или запись) —** это табличная строка с порядковым номером, в которой содержится информация об одном конкретном объекте.

ID	Название	Доставка	Начало работы	Закрытие
1	Papa Jones	есть	8:00	22:00
2	Dominos	есть	8:00	23:00
3	Pizza Hut	нет	9:00	0:00
4	Dodo Pizza	есть	8:00	0:00
5	American Hut Pizza	нет	12:00	23:00

5. **Значение —** элемент таблицы, который находится на пересечении столбцов и строк.

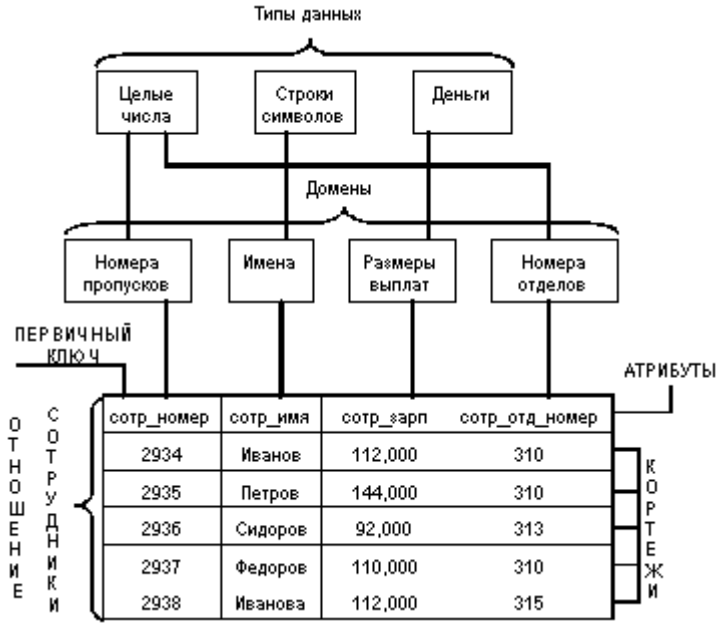
ID	Название	Доставка	Начало работы	Закрытие
1	Papa Jones	есть	8:00	22:00
2	Dominos	есть	8:00	23:00
3	Pizza Hut	нет	9:00	0:00
4	Dodo Pizza	есть	8:00	0:00
5	American Hut Pizza	нет	12:00	23:00

6. **Ключ —** это самый важный столбец в таблице, за счет этих значений и происходит взаимодействие в реляционной базе данных, он связывает таблицы между собой.

ID	Название	Доставка	Начало работы	Закрытие
1	Papa Jones	есть	8:00	22:00
2	Dominos	есть	8:00	23:00
3	Pizza Hut	нет	9:00	0:00
4	Dodo Pizza	есть	8:00	0:00
5	American Hut Pizza	нет	12:00	23:00

Ключи бывают нескольких видов:

- Первичный ключ — идентификатор, такой как индекс или артикул.
- Потенциальный ключ — другое уникальное значение, которое может служить идентификатором.
- Внешний ключ — столбец-ссылка, используется для объединения двух таблиц, каждое значение внешнего ключа обязательно соответствует первичному ключу в другой таблице.



Например, для решения задачи — выбрать все пиццерии, которые смогут доставить пиццу с ананасами после 23:00, — кроме основной таблицы с графиками работы понадобятся также таблицы с ассортиментом каждого заведения, а также таблицы с составом каждой пиццы (чтобы понять, есть ли в ней ананасы). Все эти таблицы будут связаны между собой с помощью ключей.

ID	Название	Доставка	Начало работы	Закрытие
1	Papa Jones	есть	8:00	22:00
2	Dominos	есть	8:00	23:00
3	Pizza Hut	нет	9:00	0:00
4	Dodo Pizza	есть	8:00	0:00
5	American Hut Pizza	нет	12:00	23:00

Список пиццерий в городе

ID заведения	Меню	Размер 1	Стоимость	Размер 2	Стоимость
1	Гавайская	30 см	370 руб.	35 см	450 руб.
1	Маргарита	30 см	320 руб.	35 см	390 руб.
1	4 сыра	30 см	390 руб.	35 см	470 руб.
1	Пепперони	30 см	410 руб.	35 см	490 руб.
...					

Ассортимент одной из пиццерий с ключом id — 1

SQL-операторы

Работать с данными помогают операторы — определенные слова или символы, которые используются для выполнения конкретной операции — например, для выбора из множества по конкретному параметру. Если нам нужно из всех видов пиццы отсортировать те, в которых есть пармезан, — нужно использовать оператор SELECT (выбор в соответствии с условием).

Операторы в SQL делятся на несколько групп в соответствии с задачами, которые они решают.

DDL (Data Definition Language) — операторы определения данных. Они работают с объектами, то есть с целыми таблицами. Если базу нужно дополнить таблицей с новыми данными или, наоборот, убрать одну из таблиц с ошибочными данными — используется этот набор операторов.

- CREATE — создание объекта в базе данных
- ALTER — изменение объекта
- DROP — удаление объекта

DML (Data Manipulation Language) — операторы манипуляции данными. Эти операторы уже работают с содержимым таблиц — строками, атрибутами и значениями. С их помощью можно вносить изменения в конкретное значение. Например, заменить поле в колонке «Фамилия» в строке с данными сотрудницы компании посте того, как она вышла замуж. Или удалить строку с данными уволенного сотрудника.

- SELECT — выбор данных в соответствии с условием
- INSERT — добавление новых данных
- UPDATE — изменение существующих данных
- DELETE — удаление данных

DCL (Data Control Language) — оператор определения доступа к данным. Он определяет, кто из пользователей может отправлять запросы к базе, менять объекты и значения. Например, можно отозвать доступ у сотрудника, перешедшего в другой отдел, а также открыть доступ к базе новому маркетологу или разработчику.

- GRANT — предоставление доступа к объекту
- REVOKE — отзыв ранее выданного разрешения
- DENY — запрет, который является приоритетным над разрешением

TCL (Transaction Control Language) — язык управления транзакциями. Транзакции — это набор команд, которые выполняются поочередно. Если все команды выполнены, транзакция считается успешной, а если где-то

произошла ошибка — транзакция откатывается назад, отменяя все выполненные команды. Наглядный пример такой транзакции — оплата онлайн, когда банк просит сначала ввести сумму и получателя, затем проверить и подтвердить операцию, а после ввести одноразовый код. На каждом из этих этапов оплату можно отменить, и транзакция откатится назад.

- BEGIN TRANSACTION — обозначение начала транзакции
- COMMIT TRANSACTION — изменение команд внутри транзакции
- ROLLBACK TRANSACTION — откат транзакции
- SAVE TRANSACTION — указание промежуточной точки сохранения внутри транзакции

Виды СУБД

Сами по себе таблицы или база данных не способны выполнять операции, а в СУБД можно создавать новые таблицы, удалять ненужные данные, настраивать ключи и обрабатывать запросы. Основные задачи СУБД:

- поддержка языков баз данных;
- непосредственное управление данными;
- управление буферами оперативной памяти;
- управление транзакциями;
- резервное копирование и восстановление после сбоев.

Существуют разные виды таких систем, которые разрабатывает и техногиганты, вроде Google, Microsoft и Amazon, и более нишевые студии. Разработчики стремятся сделать свой продукт лучше, чтобы их система быстрее и качественнее других обрабатывала данные. Из-за этого появились разные виды языка SQL — так называемые SQL-диалекты. У каждой СУБД диалект имеет что-то общее со всеми, а также свои особенности, которые не будут работать в другой системе.

СУБД могут быть коммерческими или иметь открытый код. Системы управления с открытым кодом можно бесплатно использовать в проектах, а также дополнять их документацию и совершенствовать процесс работы с системой. Коммерческие СУБД имеют платный доступ к полным версиям — как правило, такие используют крупные корпорации.

Система	Тип доступа	Написана на языке	Где используется
PostgreSQL	открытый	C	Skype, TripAdvisor, Яндекс.Почта, Avito
MySQL	открытый	C C++	GitHub, Facebook, Twitter, YouTube
SQLite	открытый	C	Adobe Photoshop Lightroom, Mozilla Firefox, Opera, Viber
Oracle	платный	Java C C++	МТС, Теле2, банк «Открытие», ВТБ
Google Cloud Spanner	общедоступный	C C++	PayPal, P&G, Lucille Games

- **PostgreSQL —** это объектно-ориентированная система, то есть она обрабатывает данные как абстрактные объекты. Каждый объект, в отличие от простых табличных значений, может иметь собственные характеристики и уникальные методы взаимодействия с другими объектами. Это позволяет PostgreSQL обрабатывать более сложные структуры данных и выполнять более сложные процедуры.
- **MySQL —** простая в изучении и функциональная система, которая работает с сайтами и веб-приложениями. Чаще всего используется в системах управления контентом сайтов (CMS), на сайтах с возможностью регистрации пользователей, в корпоративных системах CRM, в планировщиках, чатах и форумах. MySQL считается одним из самых безопасных и высокоскоростных решений, которое существует на рынке.
- **SQLite —** это облегченная встраиваемая версия СУБД. В ней нет возможности поделиться правами доступа, как во многих других системах, но благодаря своему устройству эта система быстрая и мощная. SQLite подходит для обработки запросов на сайтах с низким и средним трафиком, а также в однопользовательских мобильных приложениях и играх. Преимущество такой системы — файловая структура, то есть база в SQLite состоит из одного файла, поэтому ее очень легко переносить.
- **Oracle —** одна из первых СУБД, которая появилась еще в 1977 году и развивается до сих пор. Это кроссплатформенная система, которая может работать на Windows, Linux, MacOS, мобильных и других ОС. Система используется в крупных коммерческих проектах. Например, в России с Oracle сотрудничают операторы МТС и Теле2, банк «Открытие» и ВТБ.
- **Google Cloud Spanner —** это облачная система управления данными, которую Google разработал для управления собственными сервисами, например AdWords и Google Play. В 2017 году систему сделали общедоступной. Cloud Spanner относят к категории NewSQL — это системы,

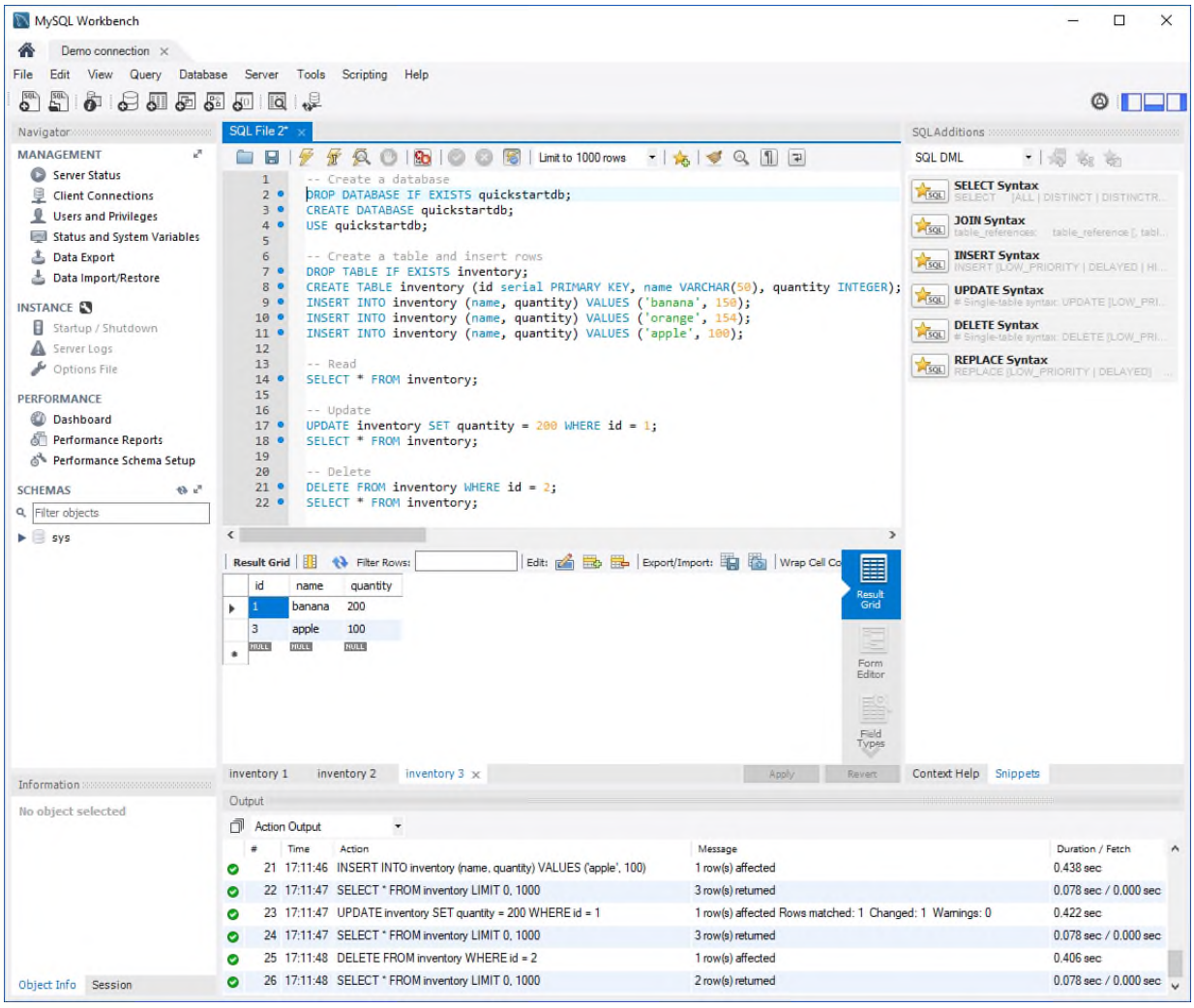
которые совмещают в себе преимущества реляционных и нереляционных СУБД.

Как начать работу с SQL

Для начала работы с SQL достаточно разбираться в основах Excel, чтобы понимать принцип работы запросов, а также иметь базовый уровень английского на уровне A1-A2. Эти навыки необходимы, чтобы понимать синтаксис SQL:

- SELECT — выбери данные
- FROM — вот отсюда
- JOIN — добавь еще эти таблицы
- WHERE — при таком условии
- GROUP BY — сгруппируй данные по этому признаку
- ORDER BY — отсортируй данные по этому признаку
- LIMIT — нужно такое количество результатов
- ; — конец предложения

Системы для работы с SQL имеют схожую структуру: есть редактор запросов, результат запросов и список таблиц, которые используются для обработки.



5. Индексы

Вы любите SQL и хотите улучшить свои навыки выполнения SQL-запросов? Вы знаете, что индексация — отличный инструмент для оптимизации запросов, но при этом не уверены, что она из себя представляет, с какой целью и как используется?

Представьте, что вы состоите в команде по аналитике электронной коммерции на Amazon и работаете с огромным объемом данных, включающих миллионы строк. Для наглядности воспользуемся условной таблицей с именем product, содержащей 12 миллионов товаров. **Кстати, именно такое количество и продается на Amazon, не считая книг, аудио и видеотехники, алкогольной продукции и услуг.**

product
product_id
product_name
product_subcategory
brand
category
price

Таблица ‘product’с 12 миллионами строк

product_id	product_name	product_subcategory	brand	category	price
1	A	headphone	Sony	electronics	$280
2	B	sneaker	Nike	shoes	$70
3	C	shirt	Levi’s	clothing	$50
4	D	baseball bat	Louisville Slugger	sports	$100

строки таблицы ‘product’

Начнем с простого запроса:

```
SELECT COUNT(*)
FROM product
WHERE category = ‘electronics’;
```

Для его выполнения база данных (БД) должна просканировать все 12 миллионов строк, чтобы проверить каждую запись на соответствие. Предположим, что время этой операции составляет **4 секунды**.

Можно ли быстрее? Конечно. А, как? С помощью индексации.

Индексация

Понятие индексации

Свое название индексация получила по образу и подобию книжного индекса. Если, читая книгу по статистике, вы ищите информацию о “линейной регрессии”, то, вряд ли, станете поочередно перелистывать сотни страниц, чтобы добраться до главы с интересующим вас материалом.

Вы просто откроете страницу индексов, найдете “линейную регрессию” и сразу перейдете на нужную страницу.

Индексация позволяет задействовать данный метод и в работе БД, которая с помощью созданного индекса быстро находит данные по запросу. А как именно это происходит, разберемся далее.

Создание индексов

Давайте создадим индекс для таблицы `product` и включим в него ‘category’:

```
Syntax:
CREATE INDEX [index_name]
ON [table_name] ([column_name]);

Query:
CREATE INDEX product_category_index
ON product (category);
```

В отличии от обычного запроса выполнение вышеуказанного займет гораздо больше времени. БД просканирует 12 миллионов строк и с нуля создаст индекс `category`. Допустим, на это уйдет **4 минуты.**

Теперь же задействуем индекс и протестируем выполнение самого первого нашего запроса:

```
SELECT COUNT(*)
FROM product
WHERE category = ‘electronics’;
```

Как видно, в этот раз он будет выполняться намного быстрее и, вероятно, займет **400 миллисекунд**.

Даже расширенные запросы, содержащие в качестве условия не только `category`, станут более эффективными благодаря созданному индексу. Рассмотрим пример:

```
SELECT COUNT(*)
FROM product
WHERE category = ‘electronics’
```

```
AND product_subcategory = ‘headphone’;
```

Выполнение этого запроса займет меньше времени, чем обычно — около **600 миллисекунд**. С помощью индекса БД быстро найдет все товары ‘electronics’ и из небольшого списка записей выберет ‘headphones’.

А сейчас изменим порядок условий в пункте WHERE.

```
SELECT COUNT(*)
FROM product
WHERE product_subcategory = ‘headphone’
AND category = ‘electronics’;
```

Несмотря на упоминание product_subcategory до category, БД тем не менее сначала выберет столбец с индексом, а именно category, после чего просканирует строки в поиске указанной product_subcategory из числа имеющихся записей.

Какова же внутренняя суть процесса?



Возможные планы выполнения запросов для оптимизатора

БД анализирует все возможные пути выполнения запроса, выбирая самый оптимальный из них.

Теперь пора познакомиться с некоторыми терминами БД. Каждый возможный путь называется **планом выполнения запроса.** По сути, это последовательность операций для получения результата SQL-запроса в **реляционной системе управления базами данных (СУРБД).**

А компонент СУРБД, определяющий наиболее эффективный способ выполнения запроса с учетом анализа всех возможных планов, называется **оптимизатором запросов**.

Индексация по нескольким столбцам

Теперь рассмотрим **индексацию по нескольким столбцам.**

Индекс можно создать более чем для одного столбца.

```
CREATE INDEX product_category_product_subcategory_index
ON product (category, product_subcategory);
```

Итак, теперь у нас есть индекс для обоих столбцов: category и product_subcategory. Обратите внимание, что здесь важна очередность — сначала сортируются данные в category, после чего в product_subcategory.

Данный тип индекса еще больше ускорит выполнение запроса, предположительно до **60 миллисекунд.**

Более того, БД может включать более одного индекса.

В каких случаях следует применять индексацию?

Индексы ускоряют работу БД, а по мере ее разрастания их эффективность становится очевиднее.

При этом важно помнить о том, что:

- Индексам необходимо место для хранения.
- При добавлении данных в БД сначала обновляется исходная таблица, а затем все ее индексы.

В связи с этим, лучше использовать индексы для БД в хранилищах данных, получающих плановые обновления, т. е. в часы наименьшей нагрузки, а не для производственных, которые обновляются постоянно. Это объясняется тем, что при постоянных обновлениях БД индексы обновляться не будут, а следовательно, станут бесполезны.

Типы индексов

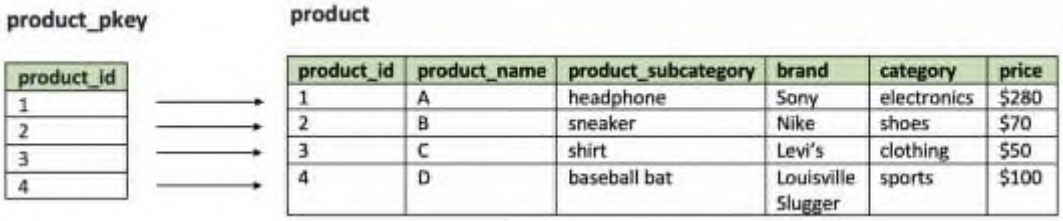
Здесь мы кратко рассмотрим 2 типа индексов БД для лучшего понимания темы:

1. Кластеризованные индексы
2. Декластеризованные индексы

Кластеризованные индексы

Кластеризованные называется особый индекс, который использует первичный ключ для структуризации данных в таблице. Он не требует явного объявления и создается по умолчанию при определении ключа. Отсортированный же в порядке возрастания первичный ключ по умолчанию применяется в качестве кластеризованного индекса.

Продемонстрируем вышесказанное на простом примере:



Кластеризованный индекс

Для таблицы product будет автоматически создан кластеризованный индекс product_pkey, сформированный вокруг первичного ключа product_id.

В этом случае при выполнении в таблице поискового запроса по product_id, как показано ниже, кластеризованный индекс поможет БД оптимально справиться с задачей и быстрее вернуть результат.

```
SELECT product_name, category, price
FROM product
WHERE product_id = 3;
```

Интересно, как же именно это происходит?

Индексы используют оптимальный метод поиска, известный как **двоичный поиск.**

Двоичный поиск — это эффективный алгоритм поиска записи в сортированном списке. Принцип его работы основан на повторяющемся делении данных пополам и определении того, находится ли искомая запись до или после записи в середине структуры данных. Если значение искомой записи меньше срединного, то поиск продолжается в первой половине, иначе — во второй. Эта процедура повторяется вплоть до нахождения значения. Благодаря данному методу уменьшается число требуемых поисков и, следовательно, ускоряется выполнение запросов.

Следующая таблица отражает соотношение записей данных и максимальное число поисков:

Number of data entries	Maximum no. of searches to find target (log_2n)
8	3
100	7
1,000	10
10,000	14
100,000	17
1,000,000	20

Сложность двоичного поиска

Аналогичным образом для нашего датасета с 12 миллионами строк понадобится не 12 миллионов, а всего лишь 24 поиска — и всё благодаря двоичному поиску. Думаю, теперь вы осознаете суперсилу индексов.

Некластеризованный индекс

Теперь узнаем, как применить преимущества индексации к столбцам, отличающимися от первичного ключа. Для этого существуют некластеризованные индексы.

Их примеры уже встречались в начальных разделах статьи во время написания оптимизированных запросов — это индексы, которые требуют явного определения.

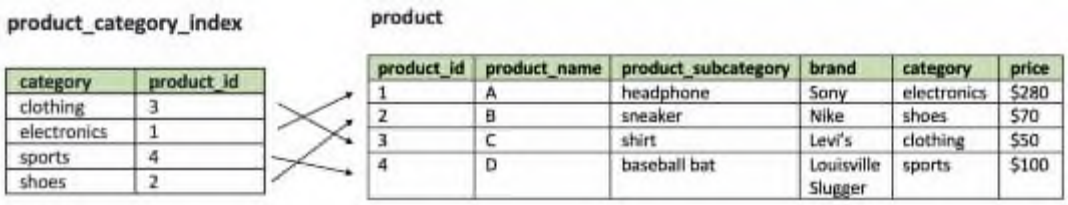
Некластеризованный индекс хранится в одном месте, а физические данные таблицы — в другом. Опять нам на ум приходит сравнение со страницей индексов, которая размещается отдельно от содержимого книги. Благодаря этой особенности для каждой таблицы можно создавать более одного некластеризованного индекса, как было показано ранее.

Как именно это происходит?

Предположим, вы уже создали некластеризованный индекс для столбца и теперь пишите запрос для поиска в нем записи. Этот индекс содержит следующее:

- записи столбца, для которых был создан индекс;
- адреса соответствующей строки (в основной таблице), в которой находится запись столбца.

Это наглядно отображено в таблице слева на рис.6:



Некластеризованный индекс

Давайте рассмотрим этот запрос более подробно:

```
CREATE INDEX product_category_index
ON product (category);

SELECT product_name, category, price
FROM product
WHERE category = 'electronics';
```

БД совершает 3 шага:

- Во-первых, она переходит по некластеризованному индексу (`product_category_index`) и методом двоичного поиска находит искомую запись столбца (`category = 'electronics'`).
- Во-вторых, в основной таблице она ищет адреса соответствующей строки, в которой находится запись столбца.
- В-третьих, она переходит к этой строке в основной таблице и выбирает другие значения столбца в соответствии с требованиями запроса (`product_name, price`).

Как видим, работа с некластеризованным индексом предполагает дополнительный шаг, включающий поиск адреса строки и переход к ней в основной таблице. Следовательно запрос с таким индексом выполняется медленнее в отличие от кластеризованного аналога.

6. Репликация и масштабирование

Масштабируемость базы данных

Масштабируемость базы данных — это способность базы данных обрабатывать изменяющиеся требования путем добавления / удаления ресурсов. Базы данных приняли множество методов, чтобы справиться.

Изначально масштабируемость базы данных заключалась в предоставлении услуг на все меньших компьютерах. Первые системы управления базами данных, такие как IMS, работали на мэйнфреймах. Второе поколение, включая Ingres, Informix, Sybase, RDB и Oracle, появилось на миникомпьютерах. Третье поколение, включая dBase и Oracle (снова), работало на персональных компьютерах.

В тот же период внимание было обращено на обработку большего количества данных и более требовательные рабочие нагрузки. Одним из ключевых нововведений в программном обеспечении конца 1980-х годов было уменьшение детализации блокировки обновлений с таблиц и дисковых блоков до отдельных строк. Это устранило критическое узкое место масштабируемости, поскольку более грубые блокировки могли задерживать доступ к строкам, даже если они не были напрямую вовлечены в транзакцию. Раньше системы были совершенно нечувствительны к увеличению ресурсов.

Когда ограничения программного обеспечения были устранены, внимание переключилось на оборудование. Инновации произошли во многих областях. Первая заключалась в поддержке многопроцессорных компьютеров. Это включало в себя возможность одновременной обработки запросов к базе данных нескольким процессорам, не блокируя друг друга. Это превратилось в поддержку многоядерных процессоров.

Гораздо более существенное изменение касалось того, чтобы распределенные транзакции могли влиять на данные, хранящиеся на отдельных компьютерах, с использованием протокола двухфазной фиксации, устанавливая архитектуру без совместного использования.

Еще позже Oracle представила архитектуру общего доступа, которая обеспечивала полную функциональность в многосерверных кластерах.

Еще одним нововведением было хранение копий таблиц на нескольких компьютерах (репликация базы данных), что повысило доступность (обработка копии могла продолжаться, даже если основная система была недоступна) и масштабируемость, особенно для запросов / анализа, поскольку запросы можно было направлять скопируйте, если основной достиг своей емкости.

В начале двадцать первого века для некоторых рабочих нагрузок системы NoSQL завоевали популярность перед реляционными базами данных. Мотивация заключалась в еще большей масштабируемости и поддержке документов и других «нереляционных» типов данных. Часто в жертву приносились протоколы строгой согласованности ACID, которые гарантировали идеальную согласованность в любое время в пользу конечной согласованности, которая гарантировала, что все узлы в конечном итоге вернут самые свежие данные. Некоторые даже позволяли время от времени терять транзакции, если система могла обрабатывать достаточно много запросов. Самой известной ранней системой была BigTable / MapReduce от Google, разработанная в 2004 году. Она обеспечила почти линейную масштабируемость на нескольких фермах серверов.за счет таких функций, как многострочные транзакции и объединения.

В 2007 году первая NewSQL система, H-Store, была разработана. Системы NewSQL пытаются объединить масштабируемость NoSQL с транзакциями ACID и интерфейсами SQL. [9]

Размеры

Масштабируемость базы данных имеет три основных аспекта: объем данных, объем запросов и размер запросов. Запросы бывают разных размеров: транзакции обычно влияют на небольшие объемы данных, но могут приближаться к тысячам в секунду; аналитических запросов обычно меньше, но они могут получить доступ к большему количеству данных. Смежное понятие - эластичность, способность системы прозрачно добавлять и уменьшать емкость для удовлетворения меняющихся рабочих нагрузок.

Вертикальный

Вертикальное масштабирование базы данных подразумевает, что система базы данных может полностью использовать максимально сконфигурированные системы, включая обычно многопроцессорные системы

с большой памятью и большой емкостью хранения. Такие системы относительно просты в администрировании, но могут иметь ограниченную доступность. Однако любой отдельный компьютер имеет максимальную конфигурацию. Если рабочие нагрузки превышают этот предел, можно выбрать либо миграцию в другую, еще более крупную систему, либо изменение архитектуры системы для достижения горизонтальной масштабируемости.

По горизонтали

Горизонтальное масштабирование базы данных включает добавление дополнительных серверов для работы с одной рабочей нагрузкой. Большинство горизонтально масштабируемых систем имеют недостатки в функциональности. Если приложению требуется больше функциональных возможностей, может быть предпочтительнее перейти на вертикально масштабируемую систему.

Оборудование

Базы данных работают на индивидуальном оборудовании различной мощности - от умных часов до суперкомпьютеров и нескольких прозрачно реконфигурируемых серверных ферм. Базы данных также масштабировались по вертикали для работы на 64-битных микропроцессорах, многоядерных процессорах и больших многопроцессорных системах SMP с использованием многопоточных реализаций.

Разногласия

Для полного использования аппаратной конфигурации требуются различные методы блокировки, от блокировки всей базы данных до целых таблиц, блоков диска и отдельных строк таблицы. Соответствующая степень детализации блокировки зависит от рабочей нагрузки. Чем меньше размер заблокированного объекта, тем меньше вероятность того, что запросы к базе данных заблокируют друг друга, пока оборудование простаивает. Обычно блокировки строк необходимы для поддержки приложений обработки больших объемов транзакций за счет накладных расходов на обработку для управления большим количеством блокировок.

Кроме того, некоторые системы гарантируют, что запрос видит согласованное по времени представление базы данных, блокируя данные, которые проверяет запрос, чтобы предотвратить их изменение обновлением, что замедляет работу. В качестве альтернативы некоторые базы данных используют многоверсионную согласованность чтения, чтобы избежать

(блокировать) блокировки чтения, при этом обеспечивая согласованные результаты запросов.

Еще одно потенциальное узкое место может возникнуть в некоторых системах, когда множество запросов пытается получить доступ к одним и тем же данным одновременно. Например, в системах OLTP многие транзакции могут пытаться одновременно вставить данные в одну и ту же таблицу. В системе без совместного использования ресурсов в любой момент все такие вставки обрабатываются одним сервером, который управляет этим разделом (осколком) таблицы, возможно, подавляя его, в то время как остальной системе мало что нужно делать. Во многих таких таблицах в качестве первичного ключа используется порядковый номер, который увеличивается с каждой новой вставленной строкой. Индекс для этого ключа также может испытывать конфликты (перегрев) при обработке этих вставок. Одно из решений для этого - перевернуть цифры первичного ключа. Это распределяет вставки как в таблицу, так и в ключ по нескольким частям базы данных.

Разбиение на разделы

Базовый метод - разбить большие таблицы на несколько разделов на основе диапазонов значений в ключевом поле. Например, данные за каждый год можно хранить на отдельном диске или на отдельном компьютере. Разделение снимает ограничения на размеры одной таблицы.

Репликация

Реплицированные базы данных поддерживают копии таблиц или баз данных на нескольких компьютерах. Этот метод масштабирования особенно удобен для редко или никогда не обновляемых данных, таких как история транзакций или налоговые таблицы.

Что такое репликация базы данных?

Репликация базы данных SQL Server — это особая технология копирования и распределения как самих данных, так объектов БД из одной базы в другую. Чтобы компоненты оставались целостными, нужна синхронизация. Репликацию можно сделать непрерывной или настроить на запуск через определенные промежутки времени.

А теперь то же самое, но простыми словами. Представьте, что у вас вместо одной БД две. Первая — главная. Именно там происходят все ключевые изменения. Чтобы не потерять важную информацию и настройки, при каждом изменении, состояние и наполнение таблиц копируется во вторую БД. Если одна из них по каким-то причинам «испортится», данные

можно будет быстро восстановить. Другими словами, репликация — это точный клон «главной» MS SQL.

Чем это отличается от обычного резервного копирования? Бэкапы — это просто информация. Например, резервная копия сервера — это данные о том, что на этом сервере было и как работало. Чтобы отладить процессы с помощью бэкапов, надо запустить восстановление. Репликация — это полностью готовый к работе клон главной базы данных. Если что-то идет не так, вы просто продолжаете работать, без каких-либо дополнительных действий. Когда главную БД починят, обновления тут же перенесутся с реплики. Чаще всего пользователи даже не замечают, что что-то произошло — процессы синхронизированы.

Компоненты репликации

Компоненты — это то, что можно настроить. Основными являются объекты, состояния, и стороны процесса. Проще говоря, это все, что отвечает за процесс синхронизации:

- что клонируется;
- кто клонирует;
- как это происходит;
- в каком состоянии это происходит;
- кто отвечает за изменения.

Ниже мы кратко рассмотрим все базовые компоненты.

Агенты

Репликация SQL Server использует заранее определенный набор автономных программ и событий, известных как агенты. По умолчанию, агенты запускаются как запланированные задания под управлением агента SQL Server.

Какими они бывают:

- снапшотов — моментальных снимков;
- чтения журналов;
- распространения;
- слияния.

Издатель

В издателе находится список объектов, обозначенных как «статьи». У него может быть одна или несколько публикаций. Именно издатель определяет механизм распространения данных, создавая несколько

хранимых процедур внутренней репликации баз данных. Этот компонент также отвечает за доступность информации.

Публикации

Публикация — это логический набор статей из базы данных. С помощью ее сущности можно определять и настраивать свойства статьи на более высоком уровне, чтобы свойства наследовались для всех статей в этой группе.

Статьи

Статья — это базовая единица клонирования. Компонент может состоять из:

- представлений;
- таблиц;
- процедур.

На основе одного объекта можно создать одну или более статей. Что можно делать со статьями во время настройки процесса:

- перемещаться по статье (с помощью Мастера создания публикаций);
- устанавливать свойства и доступы к нему;
- выводить список всех опубликованных статей;
- получать подробную информацию;
- запрашивать сведения об уровне столбца;
- перечислять столбцы, опубликованные в репликации транзакций в базе данных публикации.

Дистрибьютор

Дистрибьютор — это база данных, которая работает как хранилище для данных репликации, связанных с одним или несколькими издателями. Чаще всего он одновременно играет роль издателя и распространителя. В зависимости от места расположения бывает локальным или удаленным.

Базы данных дистрибьютора

У каждого дистрибьютора при репликации должна быть хотя бы одна собственная база данных, состоящая из:

метаданных репликации;

деталей статьи;

непосредственно данных.

Подписчик

Экземпляр базы данных, который использует сведения репликации SQL Server из публикации, называется подписчиком. Он может получать информацию от одного или нескольких издателей и публикаций, передавать обратно изменения или повторно публиковать данные (все зависит от модели репликации).

Подписки

Подписка — это запрос на доставку копии публикации подписчику. Она определяет, какие данные будут получены, где и когда.

Существует два типа подписок:

- Push — БД обновляются напрямую;
- Pull — БД обновляются по запросу.

Типы репликации

В зависимости от направления репликация MS SQL Server бывает:

- односторонней;
- двухсторонней;
- направление «одна-ко-многим»;
- направление «многие-к-одному».

Выделяют 4 типа репликации базы данных:

- моментальных снимков;
- транзакций;
- одноранговая;
- слиянием.

Моментальным снимком

Точное копирование данных в том виде, в котором они отображаются в момент создания снапшота БД.

Когда используется:

- если данные меняются не часто;
- значение реплики невелико;
- большое количество изменений происходит за малый промежуток времени.

Пример: ежедневное обновление цен на товары интернет-магазина, которые надо синхронизировать с главного сервера в полночь.

Транзакций

Синхронизация в режиме реального времени. Реплицируется не только конечное состояние БД, но и все выполненные транзакции (чтобы потом можно было отследить историю изменений).

Когда используется:

если реплика будет применяться при отказе основной БД;

головному офису нужны отчеты об операциях в филиалах;

частое изменение в SQL.

Одноранговая

Одновременная синхронизация для нескольких подписчиков.

Когда используется:

- у компании много SQL-серверов в разных географических точках;
- нужно, чтобы изменения можно было вносить на любом из серверов;
- необходимо эффективно масштабировать рабочее приложение с доступом к SQL.

Пример: крупная торговая сеть использует одну CRM, где каждый сотрудник может внести свои изменения.

Клонирование слиянием

Чаще всего используется в среде сервер-клиент, чтобы добиться синхронизации в тех случаях, когда отсутствует непрерывное подключение SQL Server.

Когда используется:

если серверы не всегда включены;

когда несколько вторичных хранилищ работают с одним общим.

Пример: магазин в Магадане заказывает со склада в Москве упаковку товара. Даже если сервер магазина в Москве выключен ночью, с утра сотрудники увидят обновленные данные об остатках на складе.

Репликация ACTIVE DIRECTORY

Репликация Active Directory гарантирует, что информация или данные между контроллерами домена остаются обновленными и согласованными.

Какие разделы клонируются в Active Directory:

- схемы;

- конфигурации;
- домены;
- приложения.

Настройка репликации

Мы рекомендуем обратиться к специалистам для настройки репликации MS SQL. Без IT-знаний и опыта это будет сложно сделать самостоятельно. В этой статье мы рассмотрим базовые принципы процесса и важные условия для клонирования.

О чем нельзя забывать

Прежде чем запустить репликацию БД, надо убедиться, что выполнены все важные условия:

для учетной записи назначена фиксированная роль db_owner;

есть разрешения на чтение и запись;

хотя бы в одной БД есть первичный ключ и статья;

на диске хватит места для хранения клонированного экземпляра.

Как настраивать

IT-специалисты используют специальный софт для того, чтобы упростить процесс настройки репликации MS SQL: SQL Server Management Studio.

Первым делом настраивают дистрибьютора, создают публикацию, папку для снапшотов и подписки. Все это можно сделать в графическом интерфейсе SQL Server Management Studio или из командной строки. Здесь же выбирают тип и настраивают компоненты. Чтобы выбрать их и настроить правильно, нужно четко понимать, какие процессы выполняет SQL Server, сколько у него дублеров и пользователей. Основное, что должен учитывать специалист — потребности пользователя, например бизнеса.

7. Резервное копирование

Что такое резервное копирование и восстановление данных?

Резервное копирование и восстановление — это создание и хранение копий данных, которые можно использовать для восстановления служб организации в случае сбоя первичных данных из-за отключения электричества, атаки программ-вымогателей и прочих нештатных ситуаций. Резервные копии позволяют восстановить состояние систем на более ранний

момент, предшествующий потере или повреждению данных, для восстановления служб.

Хотя резервные копии можно хранить на том же сервере, что и исходные данные, оптимально хранить их на другом сервере или в отдельной системе, например на защищенном облачном сервере, чтобы создать избыточность данных.

Рассматривая решения для резервного копирования и восстановления данных в вашей организации, важно определить целевое время восстановления (RTO) и целевую точку восстановления (RPO). Вам нужно знать, сколько времени займет восстановление из резервных копий и насколько актуальными будут данные.

Резервные копии и снэпшоты

Разница между этими двумя терминами заключается в деталях. Под резервной копией обычно понимается полная копия всех данных и файлов в системе. Снэпшот же копирует состояние системы в определенной точке во времени.

Что такое резервные копии?

Резервные копии — это полные копии ваших системных файлов. Процесс резервного копирования этих файлов может занять от нескольких минут до нескольких часов, в зависимости от объема данных и типа выполняемого резервного копирования.

Они предназначены для длительного хранения и в идеале должны храниться не на первоначальном сервере. Это может быть отдельный жесткий диск или другая серверная площадка.

Резервное копирование является стандартным компонентом планов обеспечения непрерывности бизнеса и аварийного восстановления (BCDR).

Что такое снэпшоты?

Снэпшоты сохраняют виртуальную картину файловой системы и настроек сервера в определенный момент времени. В отличие от резервного копирования, при котором создается полная копия данных, снэпшот копирует только параметры и метаданные, необходимые для восстановления данных в случае прерывания. Это важное различие, потому что вам все равно придется хранить исходные файлы снэпшотов в отдельном месте, чтобы иметь возможность их получить.

При восстановлении системы с помощью снэпшота она возвращается к тому состоянию, в котором находилась в момент создания снэпшота. Снэпшоты идеально подходят для краткосрочного хранения и часто используются для разработки и тестирования. Когда место для снэпшотов закончится, новые снэпшоты просто перезапишут старые.

Снэпшоты удобны, когда нужно «быстро сохранить» систему, например перед установкой значительного обновления. Если результаты обновления окажутся неудовлетворительными, можно откатиться к последнему снэпшоту, чтобы восстановить прежнее состояние системы.

Полное, инкрементное и дифференциальное резервное копирование и восстановление.

Существует три основных способа резервного копирования систем: полное, инкрементное и дифференциальное.

Каждый из них имеет свои преимущества и недостатки, поэтому для грамотного выбора важно знать принципы работы каждого способа. Возможно, вы даже решите использовать их сочетание.

Резервное копирование и восстановление полного образа. При использовании этого метода создается резервная копия всех данных в качестве точки восстановления, к которой можно легко откатить систему. Полное резервное копирование, возможно, обеспечивает наилучшую защиту данных, но для него также требуется больше всего времени. Организации, использующие резервное копирование полного образа, обычно не делают его ежедневно, поскольку это потребовало бы значительного объема дискового пространства.

Восстановление с помощью резервных копий полного образа происходит быстро, но процесс резервного копирования для создания таких копий занимает много времени. В результате данные системы, скорее всего, изменятся после выполнения последнего полного резервного копирования. Любые изменения, произошедшие после последнего полного резервного копирования, не будут восстановлены. Этот метод не идеален для организаций с более короткими RTO/RPO.

Инкрементное резервное копирование и восстановление. При использовании этого метода выполняется резервное копирование данных «по возрастанию». Процесс начинается с одного полного резервного копирования, а затем создаются резервные копии только тех данных, которые были изменены с момента последнего резервного копирования.

Недостатком инкрементных резервных копий является то, что их восстановление может занять некоторое время, поскольку восстановление также происходит постепенно. Для восстановления данных сначала необходимо развернуть последнюю полную резервную копию, а затем также применить каждую последующую инкрементную резервную копию к RPO.

Дифференциальное резервное копирование и восстановление. Как и инкрементное резервное копирование, дифференциальное резервное копирование также начинается с полного резервного копирования, а затем выполняется резервное копирование только измененных данных. При использовании этого метода восстановление обычно происходит быстрее. Требуется два файла: последняя резервная копия полного образа и последняя дифференциальная резервная копия.

8. База данных в облаке

Облачная база данных (Cloud Database) — это служба базы данных, созданная и доступная через облачную платформу. Он выполняет многие из тех же функций, что и традиционная база данных, с дополнительной гибкостью облачных вычислений. Пользователи устанавливают программное обеспечение в облачной инфраструктуре для реализации базы данных.

Ключевая особенность облачных баз данных

- Служба базы данных, создается и доступна через облачную платформу
- Позволяет корпоративным пользователям размещать базы данных без покупки специального оборудования
- Может управляться пользователем или предлагаться как услуга и управляться поставщиком
- Может поддерживать реляционные базы данных (включая MySQL и PostgreSQL) и базы данных NoSQL (включая MongoDB и Apache CouchDB)
- Доступ через веб-интерфейс или API, предоставляемый поставщиком.

Модели развертывания облачной базы

Существует две основные модели развертывания облачной базы данных.

- Традиционный, который очень похож на локальную базу данных, управляемую внутри компании, за исключением предоставления инфраструктуры. В этом случае организация приобретает пространство виртуальной машины у поставщика облачных услуг, а база данных развертывается в облаке. Разработчики организации используют модель DevOps или традиционный ИТ-персонал для управления базой данных. Организация отвечает за надзор и управление базой данных.
- База данных как услуга (DBaaS), в которой организация заключает договор с поставщиком облачных услуг через платную услугу подписки. Поставщик услуг предлагает конечному пользователю ряд задач по эксплуатации, обслуживанию, администрированию и управлению базами данных в реальном времени. База данных работает в инфраструктуре поставщика услуг. Эта модель использования обычно включает автоматизацию в областях предоставления, резервного копирования, масштабирования, высокой доступности, безопасности, установки исправлений и мониторинга работоспособности. Модель DBaaS предоставляет организациям наибольшую ценность, позволяя им использовать внешнее управление базами данных, оптимизированное за счет автоматизации программного обеспечения вместо того, чтобы нанимать собственных специалистов по базам данных и управлять ими.

Преимущества облачной базы данных

Облачные базы данных предлагают многие из тех же преимуществ, что и другие облачные сервисы, в том числе:

- Повышенная маневренность и инновации. Облачные базы данных можно очень быстро настроить и так же быстро списать, что упрощает и ускоряет тестирование, проверку и реализацию новых бизнес-идей. Если организация решает не реализовывать проект, она может просто отказаться от проекта (и его базы данных) и перейти к следующему нововведению.
- Более быстрое время выхода на рынок. При использовании облачной базы данных нет необходимости заказывать оборудование или тратить время на ожидание доставки, установки и настройки сети, когда новый продукт находится в очереди на разработку. Доступ к базе данных может быть доступен в течение нескольких минут.
- Сниженные риски. Облачные базы данных предлагают множество возможностей для снижения рисков во всем бизнесе, особенно для моделей DBaaS. Поставщики облачных услуг могут использовать автоматизацию для обеспечения соблюдения передовых методов и функций безопасности и для

снижения вероятности человеческой ошибки - основной причины простоя программного обеспечения. Автоматизированные функции обеспечения высокой доступности и соглашения об уровне обслуживания (SLA) могут уменьшить или устранить потерю дохода из-за простоев. Прогнозирование пропускной способности больше не является критической проблемой при реализации проектов, поскольку облако может представлять собой бесконечный пул оперативной инфраструктуры и услуг.

- Снижение затрат. Модели подписки с оплатой по мере использования и динамическое масштабирование позволяют конечным пользователям обеспечивать стабильное состояние, затем увеличивать масштаб для пиковых нагрузок в периоды занятости и затем уменьшать масштаб, когда спрос возвращается в устойчивое состояние. Это намного дешевле, чем поддержание этих возможностей внутри компании, когда организации должны приобретать физические серверы, способные справиться с пиковыми нагрузками, даже если им могут потребоваться пиковые возможности только пару дней в квартал. Предприятия могут сэкономить деньги, буквально отключая услуги, когда они не нужны. Они также могут снизить затраты за счет реализации глобальных инициатив с незначительными инвестициями в инфраструктуру. Во многих случаях автоматизация облачного программного обеспечения заменяет дорогих администраторов баз данных (DBA), тем самым сокращая операционные расходы за счет устранения необходимости в дорогостоящих внутренних ресурсах.

Варианты управления облачной базой данных

У предприятий есть выбор, как управлять своими облачными базами данных. Стили управления базами данных можно разделить на следующие четыре категории:

- **Самоуправляемые облачные базы данных (Self-managed).** В этой модели организация запускает свою базу данных в облачной инфраструктуре, но управляет самой базой данных, используя внутренние ресурсы, без какой-либо автоматизации, интегрированной поставщиком облачных вычислений. Эта модель предлагает некоторые из стандартных преимуществ размещения базы данных в облаке, включая повышенную гибкость и маневренность, но при этом организация сохраняет ответственность и контроль над управлением базами данных.
- **Автоматизированные облачные базы данных.** В этой модели организации используют программные интерфейсы приложений (API)

облачных служб баз данных для помощи в операциях жизненного цикла, но они поддерживают доступ к серверам баз данных и контролируют конфигурацию базы данных и операционные системы. Автоматизированные службы баз данных имеют ограниченные SLA и обычно исключают запланированные действия, такие как установка исправлений и обслуживание.

- **Управляемые облачные базы данных**. Эта модель аналогична автоматизированным облачным базам данных, но поставщик облачных услуг не разрешает потребителю доступ к серверам, на которых размещена база данных. Конфигурация ограничена конфигурациями, поддерживаемыми поставщиком облака, поскольку конечным пользователям не разрешается устанавливать собственное программное обеспечение.
- **Автономные облачные базы данных**. Это новая операционная модель без использования рук, в которой автоматизация и машинное обучение устраняют человеческий труд, связанный с управлением базами данных и настройкой производительности. Услуги включают SLA для критически важных для бизнеса приложений, таких как операции с нулевым временем простоя для незапланированных и запланированных операций с базами данных и жизненного цикла услуг.

Типы облачных баз данных - и переход к многомодельному

Существует множество типов облачных баз данных, все из которых предназначены для удовлетворения конкретных потребностей и обработки определенных типов рабочих нагрузок. Например, есть базы данных, специально разработанные для управления транзакциями, другие базы данных, предназначенные для запуска приложений интернет-масштаба, а другие служат хранилищами данных для аналитики. Применение конкретных моделей баз данных для удовлетворения потребностей конкретных приложений или рабочих нагрузок называется постоянством многоязычия.

Рабочие нагрузки OLTP поддерживаются моделями данных, которые отличаются от моделей, используемых в рабочих нагрузках OLAP. Документальные и мультимедийные данные основываются на таких форматах, как XML и JavaScript Object Notation (JSON). К другим типам баз данных относятся графовые базы данных, используемые для анализа связности, пространственные базы данных для географического анализа и хранилища «ключ-значение» для высокопроизводительного хранения и поиска простых типов данных.

Поскольку коммерческие корпоративные базы данных развивались с течением времени, они начали охватывать несколько моделей данных и методов доступа в рамках единой системы управления базами данных (СУБД). Сегодня в отрасли происходит переход к многомодельной базе данных, которая позволяет конечному пользователю работать с различными типами рабочих нагрузок из одной базовой базы данных.

Oracle называет эту концепцию "многомодельным многовариантным хранением". Эта новая возможность позволяет многим приложениям использовать одну и ту же систему управления базами данных, в то время как предприятие продолжает извлекать выгоду из уникальных моделей данных, необходимых для конкретного приложения. Эти новые архитектуры баз данных позволяют предприятиям значительно оптимизировать количество используемых баз данных и предотвращать создание разрозненных хранилищ данных, которые блокируют наиболее ценные активы (данные) организации от более широкого использования компанией.

Решения для облачных баз данных - что должно работать в облаке?

Практически любая отрасль, от финансовых услуг до здравоохранения, может получить выгоду от использования облачных решений для баз данных. Выбор не в том, использовать ли облачную базу данных. Выбор заключается в том, какая модель и тип лучше всего подходят для удовлетворения конкретных потребностей предприятия.

Многие организации предпочитают применять поэтапный подход к использованию облачных баз данных, сочетая традиционные модели облачных баз данных с моделями DBaaS. Для других, например, в сфере финансовых услуг, хранение критически важных приложений внутри компании могло бы оставаться приоритетом.

Однако все быстро меняется. По мере того, как модели DBaaS становятся более надежными и переход к автономным облачным базам данных набирает обороты, вполне вероятно, что предприятия обнаружат более широкие возможности и большие преимущества в полной миграции своих баз данных в облако.

База данных будущего: «Самоуправляемые» облачные базы данных

Новейший и наиболее инновационный тип облачной базы данных — это самоуправляемая (self-driving) облачная база данных (также известная как автономная (autonomous) база данных, о которой говорилось ранее). В то время как локальным базам данных требуется специальный администратор

баз данных для управления ими, такие же глубокие знания администраторов баз данных не требуются для управления автономной облачной базой данных. Этот тип базы данных использует облачные технологии и машинное обучение для автоматизации настройки базы данных, обеспечения безопасности, резервного копирования, обновления и других рутинных задач управления, которые традиционно выполнялись администраторами баз данных.

Автономные базы данных предназначены для автоматического противодействия аппаратным сбоям, в том числе сбоям на сайтах облачных платформ, и предлагают онлайн-исправления полного стека программного обеспечения, микропрограмм, виртуализации и кластеризации. Они легко масштабируют производительность и емкость по мере необходимости. Кроме того, они защищают данные как от внешних атак, так и от злонамеренных внутренних пользователей, и позволяют избежать многих проблем, связанных с простоями других моделей, включая плановое обслуживание.

Исследование IDC показывает, что до 75% общих затрат предприятия на управление данными могут быть связаны только с оплатой труда. Самостоятельная база данных потенциально может сэкономить среднему предприятию сотни или, возможно, тысячи рабочих часов сотрудников, занятых полный рабочий день, ежегодно для каждой из основных корпоративных баз данных. Кроме того, было подсчитано, что 72% корпоративных ИТ-бюджетов уходит на поддержку существующих систем, а на инновации остается лишь 25%.

Самостоятельные базы данных могут иметь большое значение для устранения этих высоких затрат и предоставления предприятиям возможности использовать своих администраторов баз данных для более важных задач, таких как моделирование данных, помощь программистам с архитектурой данных и планирование будущих мощностей.

Gartner прогнозирует наибольшего роста облачных баз данных

Gartner рекламирует облачные базы данных как один из самых быстрорастущих сегментов рынка общедоступных облачных сервисов и ожидает, что к 2021 году выручка базы данных как платформы как услуги (dbPaaS) достигнет почти 10 миллиардов долларов. Подробнее здесь.

На что обращать внимание при выборе облачной базы данных

Существует множество поставщиков и вариантов, доступных организациям, которые ищут решение облачной базы данных для своего

предприятия. Вам нужно выбрать модель, которая лучше всего подходит для вашего бизнеса. Ниже приведены некоторые ключевые функции, которые следует искать в любой облачной базе данных:

Производительность. Независимое онлайн-масштабирование вычислений и хранилищ, исправлений и обновлений - с непрерывной доступностью данных для приложений - обеспечит соответствие емкости вашей базы данных потребностям вашего предприятия при их изменении без прерывания работы. Автоматическая и онлайн-оптимизация производительности, такая как автоматическая индексация, является обязательной. Вам также понадобится горизонтально масштабируемая кластеризация для чтения и записи, чтобы обеспечить бесперебойную работу критически важных рабочих нагрузок в реальном времени.

Безопасность. Надежные функции безопасности имеют первостепенное значение. Любая модель базы данных, которую вы выберете, должна иметь возможность шифровать данные в состоянии покоя и во время полета и предоставлять автоматические обновления безопасности. Также важно обеспечить строгое разделение обязанностей, чтобы операторы не могли получить доступ к данным клиентов. Мощные возможности редактирования данных помогают обеспечить ограниченную и контролируемую видимость конфиденциальных данных. Обнаружение и предотвращение внешних атак на основе машинного обучения обеспечивает дополнительный уровень безопасности в реальном времени. Наконец, для ваших наиболее важных бизнес-приложений вам понадобится выделенная облачная инфраструктура, которая включает аппаратную изоляцию от других клиентов.

Остальное. Другие качества, на которые следует обратить внимание, включают читаемую резервную базу данных (в сочетании с отчетностью) для снижения затрат на высокую доступность и ведущие в отрасли технологии ретроспективного анализа, помогающие обеспечить защиту от ошибок пользователей. Наконец, ваша база данных должна иметь широкую совместимость со сторонними приложениями.

На заметку!

Исследования показывают, что среди администраторов баз данных (DBa):

- 39% управляют 50 или более базами данных
- 95% вручную создают или обновляют базы данных

- 78% испытают незапланированные простои из-за непроверенных изменений в базе данных во время своей карьеры.

При этом Автономные облачные базы данных:

Обеспечьте максимальное время безотказной работы и производительность базы данных

Обеспечьте максимальную безопасность базы данных, включая исправления и исправления

Устранение ручных, подверженных ошибкам задач управления за счет автоматизации

Позвольте администраторам баз данных применить свой опыт к функциям более высокого уровня

Перенос базы данных из локальной среды в облако

Перенос базы данных в облако может показаться сложной задачей, но это не обязательно. Ключ к успеху - заблаговременное планирование. Также важно помнить, что не все методы миграции применимы к каждому сценарию.

При выборе метода миграции необходимо учитывать несколько факторов, включая типы данных, операционные системы хоста и управление версиями базы данных. Вот несколько вещей, о которых следует подумать и к которым следует подготовиться, когда вы приближаетесь к миграции ваших баз данных в облако.

Совместимо ли программное обеспечение целевой облачной базы данных с тем, что у вас запущено в локальной среде? Совместимы ли версии ПО? Некоторые облачные провайдеры не предлагают службы баз данных, совместимые с локальными версиями. Кроме того, если ваша целевая облачная база данных поддерживает только более высокую версию программного обеспечения, которое вы используете, вы должны запланировать обновление.

- Каков размер и масштаб вашей базы данных и поддерживает ли целевое облако эту конфигурацию? Некоторые облачные провайдеры предлагают только меньшие конфигурации баз данных с точки зрения размера хранилища и количества ядер. Вам нужно заранее убедиться, что ваш провайдер может удовлетворить ваши потребности.
- Запускаете ли вы смежные сценарии на самих серверах баз данных? Если да, вам нужно будет заключить контракт на инфраструктуру как

услугу (IaaS) или автоматизированные услуги, а они могут быть недоступны через вашего облачного провайдера.

- Вам нужно выполнить миграцию с минимальным временем простоя или без него на существующее приложение? Ведущие поставщики облачных баз данных, такие как Amazon, Microsoft и Oracle, делают выбор и перенос баз данных проще, чем когда-либо. В зависимости от обстоятельств переход в облако может занять несколько минут.



МОДУЛЬ 6. АДМИНИСТРИРОВАНИЕ БАЗ ДАННЫХ

1. Типы и структура СУБД

2. Запросы, индексы и эксплейны

3. Администрирование MySQL

4. Администрирование PostgreSQL

5. Troubleshooting

1. Типы и структура СУБД

Основные функции и компоненты СУБД:

- поддержка целостности файлов
- восстановление согласованного состояния данных после сбоев
- обеспечение параллельной работы нескольких пользователей
- поддержка языка манипулирования данными

Эти функции традиционно поддерживаются СУБД.

Непосредственное управление данными во внешней памяти

Управление данными подразумевает организацию необходимых структур во внешней памяти:

- для хранения данных и метаданных, входящих в БД
- для служебных целей (для быстрого доступа к данным с помощью индексов)

В некоторых реализациях СУБД активно используются возможности существующих ФС, в других работа производится напрямую с устройствами внешней памяти.

Но подчеркнем, что в развитых СУБД пользователи в любом случае не обязаны знать, как организованы файлы. В частности, в СУБД обычно поддерживается собственная система именования объектов БД.

Управление буферами оперативной памяти

Управление буферами операционной памяти (ОП) необходимо для увеличения скорости работы с данными. СУБД обычно работают с БД значительного размера (для транзакционных систем в пределах 1 петабайта, для оптических систем возможны более 1 эксабайта).

Если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Единственным способом реального увеличения этой скорости является буферизация данных в ОП.

Обычно размеры БД больше объёма ОП, поэтому надо определить какую именно часть буферизовать. Даже если ОС производит общесистемную буферизацию данных, то СУБД намного лучше знает какую именно часть БД лучше буферизовать. Поэтому в развитых СУБД поддерживается собственный набор буферов ОП с использованием собственной дисциплиной замены буферов.

Управление транзакциями

Управление транзакциями нужно для обеспечения целостности данных и обеспечения многопользовательского доступа. Транзакция — это последовательность операций над БД, рассматриваемых СУБД как единое целое: либо все операции внутри транзакции выполняются, либо ни одна не выполняется. Поддержание механизма транзакций является обязательным в многопользовательских СУБД.

Рассмотри подробнее как транзакции поддерживают целостность БД. Вспомним наш пример ИС. Если мы хотим принять на работу нового служащего, то нам надо обязательно добавить запись в файл служащих и обновить численность его отдела. Таким образом нам надо добавить запись и изменить поле другой записи. Эти две операции нужно выполнить в одной транзакции. Тогда в случае, если после добавления нового служащего, мы ещё не успели обновить численность, и происходит сбой системы, можно без нарушения целостности БД, удалить нового служащего и написать сообщение об ошибке. Т. е. не будет ситуации, когда применена только часть изменений, которая нарушает целостность БД.

Таким образом важным свойством транзакции является сохранение целостности БД, т. к. если транзакция исполнена, то она гарантирует целостность данных в БД, а если не исполнена, то она гарантирует, что данные в БД не изменятся.

Однако сохранение целостности БД не единственное важное свойство транзакций. При правильном управлении параллельными транзакциями, каждый пользователь может ощущать себя единственным пользователем СУБД. С этим связано важное понятие сериализации транзакций.

Сериализация транзакций — это такой порядок выполнения транзакций, при котором суммарный эффект набора параллельных транзакций эквивалентен эффекту их некоторого последовательного выполнения. Другими словами, в многопользовательской СУБД при одновременной работе нескольких пользователей надо определить в каком порядке выполнять текущие транзакции так, чтобы результат их выполнения был таким, будто он выполняются не параллельно, а последовательно.

Если удается добиться действительно сериального выполнения набора транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. Наиболее распространены алгоритмы, основанные на синхронизационных блокировках объектов БД. При использовании любого алгоритма сериализации возможны конфликты между двумя или более транзакциями по доступу к объектам БД. В этом случае для поддержки сериализации необходимо выполнить откат (ликвидировать все изменения, произведённые в БД) одной или более транзакций. Это один из случаев, когда пользователь многопользовательской СУБД может реально (и достаточно неприятно) ощутить присутствие в системе транзакций других пользователей.

Журнализация

Журнализация необходима для восстановления БД в случае сбоев. Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя. Существует два вида аппаратных сбоев:

- мягкие сбои или внезапная остановка работы компьютера (например, аварийное выключение питания)
- жёсткие сбои, характеризуемые потерей информации на носителях внешней памяти

Примерами программных сбоев могут быть:

- аварийное завершение работы СУБД (по причине ошибки в программе или в результате некоторого аппаратного сбоя)
- аварийное завершение пользовательской программы, в результате чего некоторая транзакция остается незавершенной

В любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Другими словами, для обеспечения надежного хранения данных в БД требуется хранение избыточных данных, причем та часть данных, которая используется для восстановления БД, должна храниться особо надежно. Наиболее распространенным методом поддержания такой избыточной информации является ведение журнала изменений.

Журнал — это особая часть БД, недоступная пользователям СУБД и поддерживаемая с особой тщательностью, в которую поступают записи обо всех изменениях основной части БД. Записи в журнале могут соответствовать, например, некоторой логической операции изменения БД (удаления, вставки, изменения).

Принято использовать стратегию «упреждающей» записи в журнал, которая заключается в том, что запись об изменении любого объекта БД должна попасть в журнал раньше, чем изменённый объект попадет в основную часть БД. Такая стратегия используется в протоколе Write Ahead Log (WAL). Известно, что если в СУБД корректно соблюдается протокол WAL, то с помощью журнала можно решить все проблемы восстановления БД после любого сбоя.

Типовая организация современной СУБД

Организация типичной СУБД и состав её компонентов соответствует рассмотренному выше набору функций. Мы выделили следующие основные функции СУБД:

- управление данными во внешней памяти
- управление буферами оперативной памяти
- управление транзакциями
- журнализация и восстановление БД после сбоев
- поддержка языков БД

Логически в современной реляционной СУБД можно выделить:

- наиболее внутреннюю часть – ядро СУБД (database engine)
- компилятор языка БД (обычно SQL)
- подсистему поддержки времени выполнения
- набор утилит

В некоторых системах эти части выделяются явно, в других - нет, но логически такое разделение можно провести во всех СУБД.

Ядро СУБД отвечает за управление данными во внешней памяти, управление буферами ОП, управление транзакциями и журнализацию. Соответственно, можно выделить следующие компоненты ядра:

- менеджер данных
- менеджер буферов
- менеджер транзакций
- менеджер журнала

Функции этих компонентов сильно взаимосвязаны, и для обеспечения корректной работы СУБД все эти компоненты должны взаимодействовать по тщательно продуманным и проверенным протоколам.

Ядро СУБД обладает собственным интерфейсом, обычно не доступным пользователям напрямую и используемым в программах, производимых компилятором SQL и утилитах БД. Ядро СУБД является основной резидентной

частью СУБД. При использовании архитектуры «клиент-сервер» ядро является базовой составляющей серверной части системы.

Основной функцией компилятора языка БД является компиляция операторов языка БД в некоторую выполняемую программу. Основной проблемой реляционных СУБД является то, что языки этих систем (как правило, SQL) не процедурные, т.е. в операторе такого языка специфицируется некоторое действие над БД, но эта спецификация не является процедурой, а лишь описывает в некоторой форме условия совершения желаемого действия. Таким образом, компилятор должен решить, как именно исполнить оператор до начала выполнения программы.

Применяются достаточно сложные методы оптимизации операторов. Результатом компиляции является выполняемая программа, представляемая в некоторых системах в машинных кодах. Но более часто (в машинно-независимых СУБД) представляется в выполняемом внутреннем машинно-независимом коде. В этом случае реальное выполнение оператора производится интерпретатором этого внутреннего языка.

В отдельные утилиты БД выделяют такие процедуры, которые требуют особого

внимания:

- загрузка и выгрузка БД
- сбор статистики
- глобальная проверка целостности БД

Утилиты программируются с использованием интерфейса ядра СУБД, а иногда

даже с проникновением внутрь ядра.

Классификация СУБД

Классификация по модели данных

Прежде всего, СУБД различаются по модели данных, которая определяет архитектуру, структуры данных, методы работы с данными этой СУБД. Бывают:

- сетевые
- иерархические
- реляционные (и SQL-ориентированные)
- объектно-ориентированные
- XML-ориентированные и другие

Универсальные и специализированные СУБД

Существуют универсальные СУБД (Oracle, DB2, Microsoft SQL Server), которые направлены на многофункциональность, на удовлетворение всех потребностей ИС. Из-за своей всенаправленности, универсальные СУБД очень громоздкие и неповоротливые, часто они не успевают быстро адаптироваться к новым потребностям бизнеса, из-за чего бизнес теряет деньги. Считается, что именно бизнес является главной движущей силой развития СУБД (и IT продуктов в целом).

Альтернативой универсальным приходятся специализированные СУБД (Vertica, VoltDB и т. д.), ориентированные на эффективную поддержку одного класса ИС (например, транзакционных или аналитических систем). За счёт своей узконаправленности они могут быстрее адаптироваться к требованиям бизнеса.

Файл-серверные, клиент-серверные и встраиваемые СУБД

В файл-серверных СУБД (Informix SE, Microsoft Access и т.д.) БД хранится на специализированном файл-сервере, а СУБД запускается на каждом клиенте. Как правило такая архитектура используется внутри локальной сети. Так как клиенты ничего друг про друга не знают, то синхронизация многопользовательского доступа реализуется на файл-сервере. Понятно, что многопользовательская синхронизация в этом случае будет на уровне ФС, что влечёт блокировки файлов. Более того, сервер только хранит данные, и не занимается их обработкой, следовательно, ответ от такого файл-сервера будет в виде блока данных для каждого запроса. Из-за этого ощущается нагрузка на сеть при большом числе клиентов. Такая организация довольно старая и пригодна только при небольшом количестве клиентов (до нескольких десятков).

В клиент-серверных СУБД (Oracle, DB2, PostgreSQL и т.д.) все основные компоненты СУБД выполняются на отдельном сервере БД, и на нём же хранится БД. На клиенте находится интерфейсная (клиентская) часть СУБД и выполняется код приложения. Плюсом такой архитектуры является оптимизации на стороне клиента: всю работу с БД осуществляет сервер, по сети передаётся только обработанный

ответ небольшого размера. Недостатком такой архитектуры является зависимость клиентов от сервера БД. В случае, неисправности которого, ни один клиент не сможет работать с БД.

Также существенным недостатком клиент-серверных СУБД является необходимость установления прямого соединения между клиентским компьютером и сервером БД. Существует трёхзвенная клиент-серверная

архитектура, которая добавляет дополнительный вспомогательный сервер приложений - прослойку между клиентами и сервером БД, который выполняет код приложения. Это позволяет полностью изолировать клиентов от конкретной БД, и вся логика работы с БД ложится на плечи вспомогательного сервера. Трёхзвенная архитектура помимо повышения уровня безопасности трёхзвенная архитектура позволяет более гибко модернизировать приложения. При модернизации не нужно закачивать обновления в клиенты. Достаточно обновить только сервер приложений.

Стоит отметить, что клиент-серверная архитектура подразумевает работу с БД на слабых клиентских устройствах, но, в действительности, из года в год производительность компьютерной техники только увеличивается. Получается, что сеть загружена меньше, но вычислительные возможности клиентов не реализуют своих возможностей полностью.

Встраиваемые СУБД (BerkeleyDB, SQlite и т.д.) встраиваются в код приложений (информационные системы; клиенты) и полностью выполняется на том же компьютере и даже в том же процессе. Как правило это библиотека, которая подключается в код программы и позволяет использовать функции СУБД прямо внутри программы. Главная проблема здесь в незащищенности БД от клиента. Встраиваемые СУБД успешно применяются в Интернет- и мобильных приложениях.

Классификация по месту хранения БД

СУБД могут хранить данные во внешней памяти, и СУБД, сохраняющие данные в ОП (in-memory).

Исторически подавляющее большинство СУБД хранит БД во внешней памяти. В этом случае хранилище внешней памяти доступно СУБД через системные вызовы операционной системы и структурировано в виде блоков. Нужный блок из внешней памяти буферизуется в ОП, где с блоком происходит дальнейшая работа.

In-memory СУБД располагают всю БД в ОП и использует внешнюю память для обеспечения долговременности транзакций. За счёт расположения БД в ОП достигается очень большая скорость работы с БД (на 4 порядка выше по сравнению с магнитными дисками). Существует несколько подходов к организации in-memory СУБД:

- Внешняя память вообще не используется, а надёжность достигается за счёт
- хранения реплик БД в разных узлах кластерной системы

- БД хранится целиком в ОП, а журнал изменений во внешней памяти
- Самым популярным вариант, при котором БД хранится целиком и в ОП, и во внешней памяти, однако операция чтения происходит из ОП, а запись - в обе

Стоит добавить, что появление SSD накопителей качественно изменило ситуацию, и теперь разница в скорости доступа между внешней памятью на SSD и ОП составляет всего 2 порядка. Работы по увеличению скорости внешней памяти активно ведутся и сегодня, поэтому, может быть, со временем устройства внешней памяти сравняются по скорости с ОП (или ОП станет энергонезависимой). Это бы принесло существенное увеличение производительности, сделав СУБД одноуровневой (только ОП, или только внешняя память) и сохранив надежность и долговременность хранения данных.

Классификация по типу параллельности

По типу параллельности СУБД бывают:

- однопроцессорные
- параллельные с общей памятью (shared-everything)
- параллельные с общими дисками (shared-disks)
- параллельные без использования общих ресурсов (shared-nothing)

Однопроцессорные СУБД не используют аппаратные возможности параллелизма и выполняются на одном процессоре (в частности, на одном одноядерном процессоре). До появления многоядерных процессоров такие СУБД были распространены, поскольку многопроцессорные компьютеры с общей памятью были слишком дороги и малодоступны.

Параллельные СУБД с общей памятью (Oracle, DB2) поддерживались ведущими компаниями с 1980-х гг., но стали массово распространенными только после появления многоядерных процессоров. Параллельные СУБД этого класса обеспечивают межзапросный (inter-query) или внутризапросный (intra-query) параллелизм с использованием нескольких ядер (или аппаратных потоков). Наличие общей памяти позволяет избежать пересылок данных, но требует применения сложной синхронизации.

Один из подходов, направленный на упрощение архитектуры такой СУБД, и меньшением затрат на работу с общей памятью был реализован компанией Oracle. Допустим, имеется N ядер. Одно ядро объявляется управляющим, и оно отвечает за компиляцию запросов, а также планирует

выполнение уже скомпилированных запросов на других ядрах. Все оставшиеся ядра работают обработчиками запросов. Такой подход обеспечивает практически линейный прирост производительности при росте числа ядер. Однако здесь остаётся тонким вопрос синхронизации данных, требующий отдельного внимания.

В будущем параллельные СУБД получат качественный прирост в производительности за счёт энергонезависимой ОП и возникновения процессоров с очень большим числом ядер.

Параллельные СУБД с общими дисками (Oracle Real Application Cluster) работают на кластерах и имеют общую дисковую подсистему, в которой хранится БД. Т. е. все узлы кластера имеют собственную ОП, но общие диски внешней памяти. Эта архитектура сильно похожа на файл-серверную архитектуру, однако общие диски — это не файловые хранилища, а отдельные аппаратно-программные решения, которые работают с узлами не на уровне блоков, а выполняют части запросов. Таким образом обеспечивается межзапросный или внутризапросный параллелизм.

Параллельные СУБД без использования общих ресурсов (Greenplum, Vertica) работают на кластерах, но каждый узел со своим разделом БД. Для этого при компиляции запросов каждый запрос разбивается на части, адресуемые к соответствующему узлу кластера. Результат запроса получается объединением частичных результатов с узлов. На сегодняшний день считается, что именно shared-nothing архитектура обеспечивает наилучшее горизонтальное масштабирование при росте объема данных (для решения проблемы BigData в области аналитики).

Но в такой архитектуре возникает очень важная проблема разбиения БД по узлам кластера. Как разделить узлы, чтобы в среднем любой запрос выполнялся наиболее быстро?

Так как при обработке части запроса на конкретном узле, всё равно требуется обращение к другим узлам, то необходимо такое разбиение кластера, чтобы число таких связей в кластере было минимальным.

Известно, что задача такого разбиения NP-полная, т. е. решения, кроме полного перебора всех вариантов, не существует. Конечно, существуют эвристические подходы к организации shared-nothing систем, которые работают на конкретных запросах и конкретных данных, но никто не гарантирует надёжности стабильной работы системы при других данных и запросах.

Допустим, что задача разбиения N узлов по кластеру решена. Однако параллельные СУБД предполагают горизонтальное масштабирование. Другими словами, при добавлении нового N+1-го узла нужно получить новое разбиение кластера и перераспределить данные между узлами (согласно новому разбиению). Эта процедура перераспределения очень вычислительно сложная, и связанная с этим недоступность БД во время перераспределения просто недопустима для реального бизнеса.

2. Запросы, индексы и эксплейны

Запрос строится на основе одной или нескольких взаимосвязанных таблиц, позволяя комбинировать содержащуюся в них информацию. При этом могут использоваться как таблицы базы данных, так и сохраненные таблицы, полученные в результате выполнения других запросов. Кроме того, запрос может строиться непосредственно на другом запросе с использованием его временной таблицы с результатами. Запрос QBE содержит *схему данных*, включающую используемые таблицы и *бланк запроса*. При конструировании запроса достаточно выделить и перетащить с помощью мыши необходимые поля из таблиц, представленных в схеме данных запроса, в бланк запроса и ввести условия отбора записей.

Назначение и виды запросов

Запрос позволяет выбрать необходимые данные из одной или нескольких взаимосвязанных таблиц, произвести вычисления и получить результат в виде виртуальной таблицы. Полученная таблица может использоваться в качестве источника данных в следующих запросах, формах, отчетах, страницах доступа к данным. Через запрос можно производить обновление данных в таблицах, добавление и удаление записей.

С помощью запроса можно выполнить следующие виды обработки данных:

- выбрать записи, удовлетворяющие условиям отбора;
- включить в результирующую таблицу запроса заданные пользователем поля;
- произвести вычисления в каждой из полученных записей;
- сгруппировать записи с одинаковыми значениями в одном или нескольких полях в одну запись с одновременным выполнением над другими полями групповых функций;
- произвести обновление полей в выбранном подмножестве записей;
- создать новую таблицу базы данных, используя данные из существующих таблиц;
- удалить выбранное подмножество записей из таблицы базы данных; добавить выбранное подмножество записей в другую таблицу.

Многотабличный запрос позволяет сформировать записи результата путем объединения взаимосвязанных записей из таблиц БД и включения нужных полей из нескольких таблиц. В частности, при объединении двух

нормализованных связанных одно-многозначными отношениями таблиц результирующая запись образуется на основе записи подчиненной таблицы, в которую добавляются поля из связанной записи в главной таблице. Заметим, что подобное объединение формирует новую таблицу, которая не является нормализованной. Выбранный тип объединения таблиц задается при установлении связи между таблицами и определяет способ формирования записей запроса. По умолчанию связи устанавливаются с параметром объединения первого типа: объединение только тех записей, в которых значения связанных полей обеих таблиц совпадают.

Последовательное выполнение ряда запросов по образцу позволяет решать достаточно сложные задачи, не прибегая к программированию.

Предназначение индексов

Простейший метод решения задачи поиска записей в базе данных, удовлетворяющих определенному критерию, — полный перебор. Но с ростом количества записей производительность такого подхода будет заметно падать. Для повышения производительности поиска создаются вспомогательные структуры — *индексы*. Используя индексы, можно существенно поднять скорость поиска, потому что данные в индексе хранятся в форме, позволяющей нам в процессе поиска не рассматривать области, которые заведомо не могут содержать искомые элементы.

Если провести аналогию между базой данных и книгой, индексами можно считать оглавление книги и предметный указатель. Действительно, если бы у нас не было таких «индексов», для поиска конкретной главы или для поиска определения какого-то понятия пришлось бы листать и читать всю книгу целиком, пока не найдем то, что нужно. Имея оглавление и предметный указатель, нам нужно просмотреть существенно меньший объем данных, после чего мы точно узнаем номер страницы книги, на которой находится то, что мы ищем. Индексы в базах данных, по сути, устроены так же, как оглавление или как предметный указатель книги.

Важно, что использование индексов не только сокращает время поиска в абсолютном выражении, но и уменьшает алгоритмическую сложность процесса поиска. Это значит, что время, необходимое на поиск с помощью индексов, при росте объема базы данных будет расти существенно медленнее, чем при использовании полного перебора.

В качестве примера рассмотрим задачу поиска в списке чисел. Используя перебор элементов списка, в худшем случае, нам придется просмотреть список целиком. Алгоритмическая сложность такого метода —

O(n). Но если мы будем хранить наши числа особым образом — отсортированными по возрастанию или по убыванию — сможем использовать алгоритм бинарного поиска.

2 4 5 10 23 34 38 58 112 114 115 110 123 134 138 158 180

Допустим, необходимо определить, содержит ли этот отсортированный список число 158. Для этого:

- Смотрим на число в середине списка — 114. Наш список отсортирован по возрастанию, и мы ищем число 158> 114. Значит, левую половину списка до числа 114 мы можем отбросить: в ней гарантированно не может быть искомого элемента.
    - 2 4 5 10 23 34 38 58 112 **114** 115 110 123 134 138 158 180
- Теперь делаем то же самое для правой половины списка. В середине у нее число 134, значит, мы снова можем отбросить элементы левее.
    - 2 4 5 10 23 34 38 58 112 114 115 110 123 **134** 138 158 180
- Делаем то же самое для элементов правее 134. В середине у них число 158 — искомый элемент. Поиск закончен.

В итоге метод бинарного поиска дал нам результат всего за три шага. При полном переборе с начала списка нам потребовалось бы 16 шагов. Бинарный поиск имеет алгоритмическую сложность O(log(n)). Используя формулы алгоритмической сложности O(n) и O(log(n)), мы можем оценить, как будет меняться приблизительное количество операций при поиске разными способами с ростом объема данных:

Размер списка	**Поиск перебором**	**Бинарный поиск**
100	~100 операций	~7 операций
1`000	~1`000 операций	~10 операций
100`000	~100`000 операций	~16 операций
1`000`000`000	~1`000`000`000 операций	~30 операций

Результат впечатляет. Храня данные в отсортированном виде, мы не только снизили скорость поиска по ним, но и колоссально сократили скорость замедления поиска при росте объема данных.

Использование индексов в базе данных дает аналогичный результат. Принцип работы одного из важнейших индексов в базе данных (индекс на основе B-дерева) основан именно на рассмотренном нами выше принципе — возможности хранить данные в отсортированном виде.

Пользователь базы данных должен запускать разные типы запросов для разных целей. Но иногда многие запросы не возвращают ожидаемых результатов, и администратору базы данных необходимо выяснить причину. MySQL содержит полезный инструмент EXPLAIN для диагностики ошибки, при которой оператор запроса не работает должным образом. Он используется в начале любого оператора запроса, чтобы предоставить информацию о выполнении запроса. Ключевое слово EXPLAIN может использоваться с операторами SELECT, INSERT, UPDATE, DELETE и REPLACE.

Рассмотрим, как ключевое слово EXPLAIN работает с операторами SELECT для диагностики ошибок запроса или оптимизации запроса.

Вывод ключевого слова EXPLAIN для запроса SELECT:

Когда ключевое слово EXPLAIN выполняется с помощью оператора SELECT, вывод EXPLAIN будет возвращать следующие столбцы.

Столбец	Описание
id	В нем указывается идентификатор запроса. Он представляет собой порядковый номер запросов SELECT.
select_type	Он указывает тип запроса SELECT. Тип может быть SIMPLE, PRIMARY, SUBQUERY, UNION и т. д.
table	Он указывает имя таблицы, используемое в запросе.
partitions	Он указывает разделы исследуемой многораздельной таблицы.
type	Он указывает тип JOIN или тип доступа к таблицам.
possible_keys	Он указывает ключи, которые MySQL может использовать для поиска строк в таблице.
key	Он указывает индекс, используемый MySQL.
key_len	Он указывает длину индекса, который будет использовать оптимизатор запросов.
ref	Он указывает столбцы или константы, которые сравниваются с индексом, указанным в ключевом столбце.
rows	В нем указаны списки проверенных записей.
filtered	Он указывает предполагаемый процент строк таблицы, которые будут отфильтрованы по условию.

extra	Он указывает дополнительную информацию о плане выполнения запроса.

Предположим, у вас есть две связанные таблицы с именами клиенты и заказы в базе данных с именем company.

Ниже приведены необходимые операторы SQL для создания базы данных и таблиц с данными.

```
CREATE DATABASE company;

USE company;

CREATE TABLE customers (
id INT(5) AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(50) NOT NULL,
mobile_no VARCHAR(50) NOT NULL,
email VARCHAR(50) NOT NULL) ENGINE = INNODB;

CREATE TABLE orders (
id VARCHAR(20) PRIMARY KEY,
order_date date,
customer_id INT(5) NOT NULL,
delivery_address VARCHAR(50) NOT NULL,
amount INT(11),
FOREIGN KEY (customer_id) REFERENCES customers(id))
ENGINE = INNODB;

INSERT INTO customers values
(NULL,'AndreyEx', '18477366643' , 'andreyex@andreyex.ru'),
(NULL,'Destroyer', '17839394985','destroyer@andreyex.ru'),
(NULL,'Alex','14993774655' , 'alex@andreyex.ru');

INSERT INTO orders value
('3456', '2021-01-03', 1, 'Moskow', 30000),
('3457', '2021-01-13', 3, 'Krasnodar', 25000),
('2314', '2021-02-01', 1, 'Samara', 18000),
('8563', '2021-02-11' , 2 , 'Sankt Petergburg', 19000),
```

Выполните следующую инструкцию, чтобы просмотреть текущий список записей в таблице клиентов.

```
SELECT * FROM customers;
```

Выполните следующую инструкцию, чтобы увидеть текущий список записей таблицы заказов.

*SELECT * FROM orders;*

Использование простого оператора EXPLAIN:

Следующий оператор SQL вернет ключевое слово оператора EXPLAIN простого запроса SELECT, который извлекает все записи из таблицы клиентов.

*EXPLAIN SELECT * FROM customers \G;*

Следующий вывод появится после выполнения оператора. Это запрос с одной таблицей, и в нем не используются специальные предложения, такие как JOIN, UNION и т. д. Для этого значение select_type равно SIMPLE. Таблица клиентов содержит всего три записи, поэтому значение строк равно 3. Значение отфильтрованного равно 100%, поскольку извлекаются все записи таблицы.

Использование EXPLAIN в запросе SELECT с JOIN:

Следующая инструкция EXPLAIN применяется в запросе SELECT для двух таблиц с предложением JOIN и условием WHERE.

EXPLAIN SELECT customers.name, orders.order_date, orders.amount
FROM customers
JOIN orders ON (customers.id = orders.customer_id)
WHERE customers.name = 'AndreyEx' \G

Следующий вывод появится после выполнения оператора. Здесь select_type равно SIMPLE для обеих таблиц. Две таблицы связаны отношением «один ко многим». Первичный ключ из клиентов таблицы используется в качестве внешнего ключа из заказов таблицы. Для этого значение possible_keys для второй строки — customer_id. Отфильтрованное значение для таблицы клиентов составляет 33%, потому что «AndreyEx» является первой записью в этой таблице и больше искать не нужно. Отфильтрованное значение таблицы заказов равно 100% из-за всех значений заказов в таблице, необходимая для проверки для получения данных.

В выводе вышеуказанного оператора есть предупреждение. Следующая инструкция используется для просмотра запроса, который выполняется после внесения каких-либо изменений оптимизатором запросов, или проверки причины ошибки, если какая-либо ошибка возникает после выполнения запроса.

SHOW WARNINGS \G

В запросе нет ошибки. В выходных данных отображается измененный выполненный запрос.

Использование EXPLAIN для обнаружения ошибки запроса SELECT:

Запрос SELECT, используемый в следующем операторе EXPLAIN, содержит ошибку. MySQL поддерживает формат даты «YYYY-MM-DD». Но в условии WHERE этого запроса значение даты указано как ‘DD-MM-YYYY’, что неверно.

```
EXPLAIN SELECT customers.name, orders.order_date, orders.amount
FROM customers
JOIN orders ON (customers.id = orders.customer_id)
WHERE orders.order_date = '10-10-2020' \G
```

После выполнения оператора появится следующий вывод. Будет показано два предупреждения. Одно значение по умолчанию, которое объясняется в предыдущем примере, а другое — для ошибки даты, упомянутой ранее.

Запустите инструкцию, чтобы увидеть ошибку.

```
SHOW WARNINGS \G
```

Выходные данные ясно показывают ошибку с сообщением об ошибке и именем столбца.

Использование EXPLAIN в запросе SELECT с оператором UNION ALL:

Оператор UNION ALL используется в запросе SELECT для извлечения всех совпадающих значений столбцов с дубликатами из связанных таблиц. Следующий оператор отобразит выходные данные EXPLAIN применения оператора UNION ALL между таблицами клиентов и заказов.

```
EXPLAIN SELECT id as ID
FROM customers
UNION ALL
SELECT customer_id as ID
FROM orders \G
```

В выводе значение select_type — UNION для второй строки вывода, а значение Extra — это индекс.

Вывод:

Мы рассмотрели очень простое использование оператора EXPLAIN. Но этот оператор можно использовать для решения различных сложных проблем с базой данных и оптимизации базы данных для повышения производительности.

3. Администрирование MySQL

Популярная система управления базами данных **MySQL** широко применяется для различных нужд, в первую очередь как стандарт де-факто в области интернет хостинга. Не менее широко распространен пакет для управления данной СУБД - **phpMyAdmin**. Без сомнения, это хороший, удобный продукт, но очень часто случается так, что навыки работы с этим продуктом полностью заменяют навыки работы с самой СУБД. Поэтому в данном материале мы рассмотрим, как выполнять основные задачи администрирования MySQL из командной строки.

Управление базами данных и пользователями в MySQL

Работу с СУБД следует начинать с создания баз данных и пользователей этих баз. Задачи по сути своей простые и прекрасно, а главное - просто, решаются из консоли. Для работы с сервером MySQL предназначена одноименная утилита **mysql**, работа с которой происходит в интерактивном режиме, поэтому вначале подключимся к серверу: mysql -u root -p

Где ключ -u задает имя пользователя, а -p указывает на аутентификацию по паролю, синтаксис команды позволяет указать пароль, вписав его без пробелов сразу после ключа, но в этом случае он сохранится в истории команд, что не совсем хорошо, поэтому лучше ввести пароль интерактивно. Выполнив эту команду, мы окажемся в среде MySQL, на что указывает изменившееся приглашение командной строки.

Работа в данной среде имеет свои особенности: каждая команда должна завершаться символом «**;**» или **\g**, о чем, кстати, написано в первой строке приветствия. Выход из данного режима осуществляется командой: quit;

Сразу об одной очень распространенной ошибке: забыли поставить точку с запятой в конце команды. Что делать? Ничего страшного, просто добейте недостающие символы в следующей строке.

Для начала посмотрим список баз: show databases;

Как говорится, без лишних подробностей, но для большинства административных задач этого достаточно:

Узнать список пользователей немного сложнее, он хранится в виде таблицы служебной базы **mysql**, поэтому пишем небольшой запрос:

select user,host from mysql.user;

Команда **select** выбирает указанные колонки **user, host**, опция from указывает откуда мы их выбираем, а именно из таблицы user базы данных **mysql**.

Первая колонка указывает пользователя, вторая - хост, на котором данному пользователю разрешены подключения, **%** - обозначает любое значение. Обратите внимание, что если **ivanov** имеет в настройках **localhost**, то при подключении к **mysql** по адресу **127.0.0.1** он не сможет получить доступ, это нужно учитывать, когда вы указываете данные подключения к серверу СУБД в своих приложениях.

Посмотреть владельцев баз можно следующим запросом:

select host,db,user from mysql.db;

Особых комментариев здесь не требуется. Например, **ivanov** имеет права на базы с префиксом **ivanov_** при подключении через **localhost**, а andrey на базы с префиксом **andrey_** на любом хосте. Посмотреть привилегии пользователя можно командой:

show grants for 'ivanov'@'localhost';

Имя пользователя и хост оборачиваем одинарными кавычками.

Первая строка сообщает, что у указанного пользователя отсутствуют привилегии (**USAGE**) на любую таблицу любой базы, вторая строка говорит о всех основных привилегиях для всех таблиц всех баз с префиксом **ivanov_**.

Подробный разбор системы прав MySQL выходит далеко за пределы данной статьи, скажем только что **ALL PRIVELEGES** дает пользователь все права на свои базы, но не позволяет управлять правами доступа для других пользователей. Для этого используется набор прав **ALL PRIVELEGES WITH GRANT OPTION**, который имеет по умолчанию **root**. Для простого пользователя такой набор прав избыточен.

Попробуем создать нового пользователя:

create user 'petrov'@'localhost' identified by 'password';

Синтаксис команды прост, мы указываем имя пользователя и хост, а также идентификационные данные в виде пароля. Все передаваемые значения оборачиваются в одинарные кавычки. Создав пользователя, нужно задать ему права, это делается командой **GRANT**. Сначала явно лишим его привилегий на чужие базы:

*grant usage on *.* to 'petrov'@'localhost';*

Затем можем выставлять права по своему усмотрению, например, выдача полных прав на базы с шаблоном имени **petrov_**:

grant all privileges on `petrov\_%`. to 'petrov'@'localhost';*

Обратите внимание, что шаблон следует обернуть символами **грависа** (`), которые расположены на клавише с русской буквой Ё.

Выдать права на отдельную базу можно так:

grant all privileges on andrey_drupal8. to 'petrov'@'localhost';*

Для отбора прав служит команда REVOKE, которая имеет аналогичный синтаксис, только **to** (*кому*), заменяем на **from** (*у кого*). Например:

revoke all privileges on andrey_drupal8. from 'petrov'@'localhost';*

Для того чтобы MySQL сервер применил права следует заставить его перезагрузить кэш привилегий командой:

flush privileges;

Также вам может понадобиться сменить пароль пользователя:

set password for 'petrov'@'localhost' = password('newpassword');

Переименовать его, причем переименование не обязательно обозначает смену именно имени пользователя, можно изменить как имя, так и хост, в любых комбинациях:

rename user 'petrov'@'localhost' to 'petr'@'127.0.0.1';

Ну и наконец удалить учетную запись:

drop user 'petr'@'127.0.0.1';

Перейдем от пользователей к базам данных, в простейшем случае для создания новой базы достаточно команды:

create database petrov_newdb;

Это создаст базу данных с кодовой страницей и кодировкой сравнения по умолчанию. Если сервер СУБД отдельно не настраивался, то такой кодировкой скорее всего будет **latin1_swedish_ci**, что в ряде случаев может вызвать проблемы, если не сейчас, то в дальнейшем, поэтому правилом хорошего тона будет явно указывать кодировку при создании базы. Для UTF-8 это будет так:

create database petrov_newdb default character set utf8 collate utf8_general_ci;

Для Windows-1251:

create database petrov_newdb default character set cp1251 collate cp1251_general_ci;

Для удаления базы используйте:

drop database petrov_newdb;

Проверка, оптимизация, исправление ошибок баз данных MySQL

По мере активной работы MySQL базы могут фрагментироваться, а также содержать ошибки в данных таблиц. Мы не говорим сейчас о серьезных сбоях, такие ситуации следует рассматривать индивидуально, а о простейших ошибках, которые успешно устраняются средствами самой СУБД. Для проверки, ремонта и оптимизации удобно использовать утилиту mysqlcheck.

Для проверки базы данных выполните, где **andrey_drupal8** - имя базы:

mysqlcheck -u root -p --check andrey_drupal8

Сразу все базы можно проверить командой :

mysqlcheck -u root -p --check --all-databases

А так как весь вывод на экран скорее всего не поместится, то есть смысл перенаправить его команде **less**:

mysqlcheck -u root -p --check --all-databases | less

Обратите внимание, что **less** позволяет прокручивать вывод как вниз, так и вверх, используя стрелки, для выхода нажмите **q**.

Если в какой-либо из баз были обнаружены ошибки стоит попытаться их исправить, для этого укажите:

mysqlcheck -u root -p --auto-repair andrey_drupal8

Для оптимизации используйте ключ --**optimize**, оптимизировать можно отдельную базу или несколько, для этого перечислите их после ключа --**databases**:

mysqlcheck -u root -p --optimize --databases andrey_drupal8 petrov_newdb

а также сразу все:

mysqlcheck -u root -p --optimize --all-databases

Выгрузка и загрузка дампов БД MySQL

Еще одна распространенная задача при администрировании любой СУБД, дампы баз используются как для резервного копирования, так и для переноса или создания копий. Если резервное копирование - процесс автоматизированный, то создание копий для переноса на другой сервер или перед какими-то существенными вмешательствами в структуру базы приходится делать вручную.

А если дело касается больших баз, то тут **phpMyAdmin** плохой помощник, сказываются ограничения на время исполнения скриптов, размер загружаемого файла, доступную память и т.д. И если выгрузить большой дамп с его помощью еще можно, то вот загрузить обратно может и не получиться.

Начнем с создания дампов, для этих целей используется утилита **mysqldump**, синтаксис которой повторяет синтаксис **mysqlcheck**. Для выгрузки дампа используйте команду:

mysqldump -u root -p andrey_drupal8 > ~/drupal8.sql

Чтобы выгрузить сразу несколько баз используйте ключ **--databases** или **--all-databases** для создания дампа сразу всех баз. Вывод команды следует направить в файл и указать его расположение, в нашем случае это файл

drupal8.sql в домашней директории. Также можно передать вывод по конвейеру архиватору и получить сразу архив:

mysqldump -u root -p andrey_drupal8 | gzip > ~/drupal8.sql.gz

В целом мы не советуем использовать один дамп сразу для нескольких баз, лучшим вариантом будет свой дамп для каждой базы, в тоже время использование ключа **--all-databases** оправдано в случаях, когда вам надо быстро сделать резервную копию всего сервера, например, при обновлении или переустановке СУБД, чтобы можно было быстро восстановить информацию если что-то пойдет не так.

Для того чтобы восстановить базу нужно направить дамп на вход утилиты mysql, для одиночных дампов всегда следует указывать базу приемник, например:

mysql -u root -p andrey_drupal8 < ~/drupal8.sql

В качестве приемника вовсе не обязательно должна выступать база источник, но учтите, что если база уже существует, все ее содержимое будет заменено содержимым дампа.

Для дампов, содержащих более одной базы просто укажите:

mysql -u root -p < ~/all.sql

В этом случае каждая база будет загружена в свой источник, если база источник не существует, то она будет создана.

Как видим, создание и восстановление дампов при помощи командной строки происходит буквально в одну строку и гораздо проще, и быстрее, чем с использованием phpMyAdmin или подобных ему инструментов.

Восстановление забытого пароля root MySQL

Еще одна очень популярная задача. Скажем сразу, восстановить пароль суперпользователя MySQL, как и любого другого пользователя, нельзя, но можно его сменить. Для этого нужно обладать правами суперпользователя операционной системы. Сначала остановим службу СУБД:

service mysql stop

Затем запускаем ее в безопасном режиме с пропуском таблиц привилегий:

mysqld_safe --skip-grant-tables &

Обратите внимание, что после выполнения данной команды приглашение командной строки пропадет, останется один мигающий курсор. Пишем прямо туда:

mysql -u root

и попадаем в среду mysql с правами root без ввода пароля.

Нам кажется, что многие уже догадались, что делать дальше, однако выполнить **set password** в данном режиме не получится, поэтому надо идти другим путем. Вспоминаем, что информация о пользователях хранится в таблице **user** служебной БД **mysql**. После чего выполним следующий запрос:

update mysql.user set password = password ('newpassword') where user='root';

Важно! В новых версиях MySQL вместо колонки password таблицы user используется колонка *authentication_string*, поэтому вместо set password следует указывать set *authentication_string.*

Данный запрос установит новый пароль **newpassword** для всех строк в колонке **user** которых значится **root**.

Обновим кэш привилегий:

flush privileges;

Выйдем из режима:

quit;

Остановим службу и запустим в обычном режиме:

service mysql stop

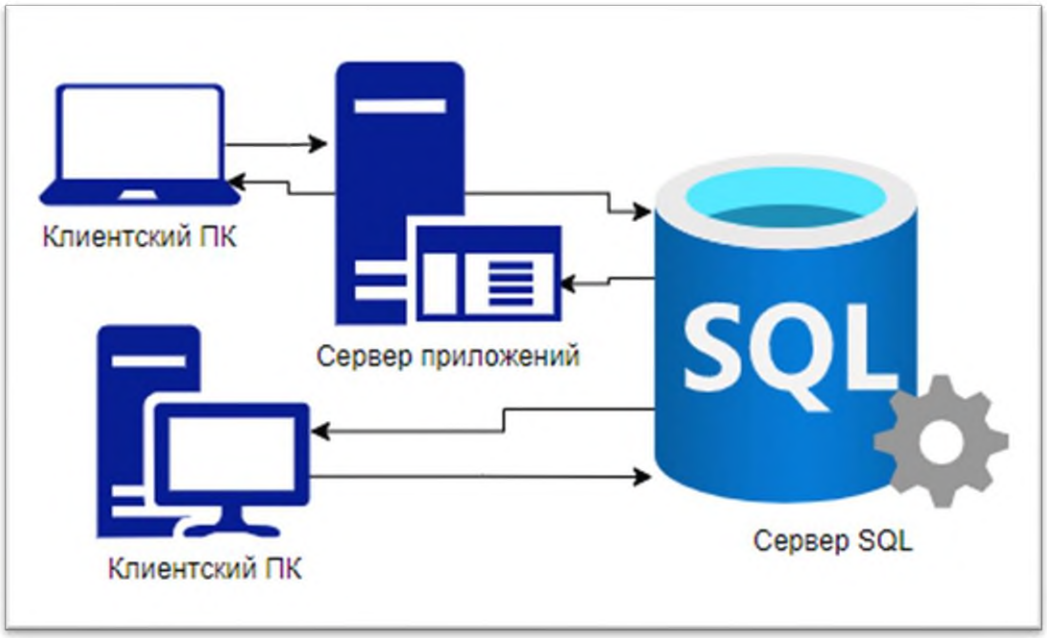
service mysql start

4. Администрирование PostgreSQL

4.1. Общее устройство PostgreSQL

Клиент серверная работа PostgreSQL

PostgreSQL эта сервер, который обслуживает базы данных SQl. К нему подключаются клиенты и с помощью SQL запросов работают с этими базами. Клиентские приложения могут быть расположены на сервере приложений или прямо на компьютере пользователя.



Клиент – это какое-то приложение, например, psql. Клиент с сервером общается по определённому протоколу. Протокол у PostgreSQL открытый, но для каждого приложения его не нужно реализовывать. Обычно используют стандартные библиотеки и драйверы. Основная библиотека это libpq, её использует psql и все штатные утилиты PostgreSQL. Для многих языков программирования есть свои библиотеки, которые основаны на libpq.

Алгоритм работы с сервером примерно такой:

- клиент подключается к серверу, а сервер выполняет его аутентификацию;
- клиент формирует запросы, а сервер их выполняет и возвращает результаты;
- дополнительно клиент управляет транзакциями, а сервер обеспечивает их поддержку.

Транзакции SQL

Транзакция – это группа последовательных операций с базой данных, которые выполняются как одна операция.

Транзакция имеет определённые требования:

- Атомарность – это означает что транзакция всегда выполняется полностью, или не выполняется вообще. Это гарантирует что все операции в транзакции будут успешно завершены. А если не будут, то транзакция, откатив все предыдущие операции и база вернется в исходное состояние. Другими словами, нельзя выполнить транзакцию наполовину.
- Согласованность – предполагается, что база до выполнения транзакции находится в одном согласованном состоянии. Согласованность гарантирует что после выполнения транзакции база должна перейти в другое согласованное состояние, или вернуться в прежнее.
- Изолированность – это свойство позволяет транзакциям работать независимо друг от друга. Две параллельные транзакции должны отработать, как будто выполнялись последовательно.
- Долговечность – гарантирует, что результат совершенной транзакции сохранится в случае сбоя системы.

За выполнение атомарности, согласованности и изолированности отвечает много-версионность. За долговечностью следит другой механизм – журнал.

Выполнение запроса

Запрос в процессе своего выполнения проходит 4 стадии:

- Разбор. Система определяет корректность запроса. Он должен быть синтактически правильным и все объекты, к которым он обращается, должны существовать. Также к этим объектам должны быть права доступа. Для разбора PostgreSQL хранит системные каталоги – это такое место, где СУБД хранит информацию о таблицах и столбцах и служебные сведения. Системные каталоги — это обычные таблицы. Поэтому вы можете удалить и пересоздать их, добавить столбцы, изменить и добавить строки, то есть разными способами вмешаться в работу системы.
- Трансформация. В PostgreSQL есть механизм правил, с помощью которых можно немного изменить запрос. Например, если вы ссылаетесь на какое-нибудь представление, то имя представления с помощью правил трансформируется в запрос, который стоит за этим представлением. После трансформации получается более низкоуровневый запрос.
- Планирование. Когда мы пишем запрос на SQL, мы говорим, что хотим получить, а не как это сделать. За то “как это сделать” отвечает планировщик, основываясь на статистике. Статистика включает в себя: сколько у нас таблиц, сколько в них строчек, сколько они занимают страниц, как распределены данные в столбцах и тому подобное. Планировщик решает с каких таблиц начать выполнение запроса, какие условия и в каком порядке применять и так далее. Планировщик подготавливает план выполнения.

- Выполнение. И наконец, когда построен план выполнения, начинается работа с данными.

После выполнения запроса его результат отдаётся клиенту. Обычно результат отдается целиком, но можно использовать так называемые курсоры, чтобы отдавать результат построчно. Курсор вначале нужно открыть, затем с помощью команды FETCH мы построчно получаем результат запроса и в конце закрываем курсор.

Еще в PostgreSQL есть механизм подготовки оператора. Мы заранее можем провести разбор и трансформацию и сохранить такой разобранный запрос. Дальше, для подготовленного оператора вы задаёте фактические параметры и запрос минуя разбор и трансформацию сразу начинает планироваться с этими параметрами. Если у запроса нет параметров, то и план запроса сохраняется и сразу начинается выполнение.

Процессы и память

Процессов на сервере PostgreSQL много, но выделяют три вида:

- Основной процесс, раньше назывался Postmaster, сейчас просто Postgres. Он слушает назначенный порт и по необходимости запускает другие процессы: фоновые или обслуживающие. Когда клиентское приложение хочет подключиться, то основной процесс запускает обслуживающий процесс и подключает приложение к нему.
- Обслуживающие процессы. Работает с клиентом (отвечает за обработку его запросов). У этого процесса есть локальная память для хранения подготовленных операторов (разобранных запросов), курсоров и другого. Сколько клиентов подключаются к серверу, столько и обслуживающих процессов создается.
- Фоновые процессы. Выполняют различные служебные операции, например, очищают базу от уже неактуальных данных.
- Память, выделяемая для работы PostgreSQL тоже бывает разных типов:
- Локальная память процесса. Необходима обслуживающему процессу для хранения подготовленных операторов, курсоров, различных переменных окружения и тому подобное. В эту память входит и так называемая WorkMem память, которая используется для внутренних операций.
- Общая память. В ней хранятся фактические данные, то есть таблицы базы данных. Все процессы взаимодействуют с базой данных и друг с другом через эту память. Так эта память общая для всех процессов, то к ней применяют механизм блокировок. Если 1 процесс работает с какой-то структурой данных он должен её заблокировать от других процессов.

Пулы соединений

Чтобы уменьшить потребление памяти, иногда используют пулы соединений. Между клиентом и сервером работает менеджер пула. Он открывает несколько соединений с базой данных и держит их. А все клиентские соединения работают с базой данных через этот менеджер пула. Так получается уменьшить количество обслуживающих процессов и соединений, а значит экономится оперативная память.

Встроенного пула в PostgreSQL нет, но клиент сам может его реализовать.

Двойная буферизация PostgreSQL

Напрямую к дискам PostgreSQL не обращается, а обращается, используя ОС. В Linux все что читается или пишется на диск всегда проходит дисковый кэш. Например, прочитали вы файлик, он попал в кэш (в оперативную память), следующий раз при обращении к этому файлу он будет читаться из кэша, а не с диска.

Получается, выполнили вы запрос, некоторые таблички были прочитаны с диска и попали в дисковый кэш. Из кэша PostgreSQL забрал из в общую память, чтобы поработать с табличками. Получается данные дублируются в оперативной памяти. Это не очень хорошо, но нужно знать про эту особенность работы PostgreSQL.

Надежность PostgreSQL при сбоях

Чтобы гарантировать восстановление после сбоя PostgreSQL использует журнал предварительной записи – WAL. Этот журнал позволяет после сбоя восстановить согласованность данных. Подробнее разберу его позже, а пока просто запомните, что такой механизм присутствует.

Расширяемость PostgreSQL

Так как PostgreSQL это база с открытым исходным кодом, её можно дописать под себя. Но PostgreSQL позволяет многое в себя добавить, не меняя код ядра с помощью расширений. Расширения позволяют создавать новые типы данных, новые типы индексов, новые функции, операторы и другое. Также можно создавать свои фоновые процессы.

Дополнительно к вышесказанному, к базе данных можно подключать внешние источники данных. Например, подключить к базе данных файл и работать с ним как с таблицей.

4.2. Задачи администрирования

Статистика работы PostgreSQL

PostgreSQL собирает статистику с помощью фонового процесса “stats collector” (коллектор статистики). Эта статистика может понадобиться для анализа работы сервера PostgreSQL. В этой статье разберем системные представления, с помощью которых можно смотреть на эту статистику.

Статистика PostgreSQL включается конфигурационными параметрами в файле **postgresql.conf**, например:

- **track_counts** – обращения к таблицам и индексам. Этот параметр включен по умолчанию.
- **track_io_timing** – статистика операций ввода/вывода. Этот параметр выключен по умолчанию, в результате нужно включать. Но следует учитывать, что включение этого параметра может замедлить сервер;
- **track_functions** – статистика вызовов функций и времени их выполнения. Этот параметр, тоже выключен по умолчанию. Если мы захотим включить его, то можем указать следующие значения:
- **pl** – включает отслеживание функций только на процедурном языке,
- **all** – включает отслеживание функций на всех языках, например, SQL и C.

Каждый **backend** процесс собирает статистику в процессе своей работы. Затем эта статистика отправляется процессу **stats collector**, который собирает статистику со всех **backend** процессов. Раз в полсекунды, статистика сбрасывается в каталог **$PGDATA/pg_stat_tmp**, этот период (500мс) можно изменить только при компиляции. В результате эту статистику можно посмотреть с помощью представлений и функций.

При остановке сервера PostgreSQL, статистика сбрасывается в другой каталог – **$PGDATA/pg_stat**.

Статистика ведется с момента первого запуска сервера, а с помощью функции **pg_stat_reset()** её можно сбросить. Другими словами, обнулить все счетчики. Но это обнулит не все счетчики, а только в текущей базе данных.

На уровне всего кластера обнулить счетчики можно с помощью функции **pg_stat_reset_shared ()**. Аргумент может принимать значения **bgwriter** и **archiver**, с которыми обнуляются все счётчики в представлении **pg_stat_bgwriter** или **pg_stat_archiver**.

Статистику можем смотреть в следующих представлениях:

- **pg_stat_all_tables** – в разрезе строк и страниц для определённой базы данных;
- **pg_statio_all_tables** – в разрезе 8 КВ страниц для определённой базы данных;
- **pg_stat_all_indexes** – по индексам для определённой базы в разрезе строк;
- **pg_statio_all_indexes** – по индексам для определённой базы в разрезе страниц;

- **pg_stat_database** – глобальная статистика по определённой базе данных;
- **pg_stat_bgwriter** – статистика для анализа фоновой записи.

Утилита pgbench

В PostgreSQL есть специальная утилита **pgbench**. С помощью которой можно произвести нагрузочное тестирование. Команда *pgbench -i <база данных>* создаст 4 таблицы **pgbench_accounts, pgbench_branches, pgbench_history и pgbench_tellers.** Предварительно уничтожит существующие таблицы с этими именами.

Таким образом можем запустить нагрузочное тестирование на 10 секунд pgbench *-T 10 <имя базы данных>.*

Практика

Подготовим базу данных для нагрузочного тестирования и анализа.

Во-первых, включим сбор статистики ввода/вывода и выполнения функций:

```
postgres@s-pg13: ~$ psql
Timing is on.
psql (13.3)
Type "help" for help.
postgres@postgres=# ALTER SYSTEM SET track_io_timing=on;
ALTER SYSTEM
Time: 0,733 ms
postgres@postgres=# ALTER SYSTEM SET track_functions='all';
ALTER SYSTEM
Time: 0,731 ms
postgres@postgres=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
Time: 1,664 ms
```

Дальше создадим базу данных admin_monitoring и с помощью специальной утилиты pgbench заполним её:

```
postgres@postgres=# CREATE DATABASE admin_monitoring;
CREATE DATABASE
Time: 101,196 ms
postgres@postgres=# \q
postgres@s-pg13:~$ pgbench -i admin_monitoring
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
```

NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.25 s (drop tables 0.00 s, create tables 0.00 s, client-side generate 0.16 s, vacuum 0.05 s, primary keys 0.04 s).

Следующим шагом сбросим все накопленные ранее счетчики статистики. Во-первых, статистику для текущей базы сбросим с помощью функции pg_stat_reset(). А во-вторых, с помощью функции pg_stat_reset_shared() очистим глобальную статистику записи на диск:

postgres@s-pg13:~$ psql -d admin_monitoring
Timing is on.
psql (13.3)
Type "help" for help.
postgres@admin_monitoring=# SELECT pg_stat_reset();
pg_stat_reset

(1 row)
Time: 1,033 ms
postgres@admin_monitoring=# SELECT pg_stat_reset_shared('bgwriter');
pg_stat_reset_shared

(1 row)
Time: 0,296 ms

Затем запустим на 10 секунд pgbench для нагрузочного тестирования базы admin_monitoring:

postgres@admin_monitoring=# \q
postgres@s-pg13:~$ pgbench -T 10 admin_monitoring
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 10 s
number of transactions actually processed: 18432
latency average = 0.543 ms
tps = 1843.192532 (including connections establishing)

tps = 1843.508640 (excluding connections establishing)

В результате мы видим, что pgbench работал с базой данных на скорости 1843 транзакций в секунду в течение 10 секунд.

Далее очистим pgbench_accounts, чтобы это действие тоже попало в статистику:

```
postgres@s-pg13: ~$ psql -d admin_monitoring
Timing is on.
psql (13.3)
Type "help" for help.
postgres@admin_monitoring=# VACUUM pgbench_accounts;
VACUUM
Time: 32,538 ms
```

Просмотр статистики для одной таблицы

Посмотрим статистику по таблице pgbench_accounts в разрезе строк. Другими словами, сколько наша система прочитала, записала или изменила строк. Для этого заглянем в представление pg_stat_all_tables:

```
postgres@admin_monitoring=# SELECT * FROM pg_stat_all_tables WHERE relid='pgbench_accounts'::regclass \gx
-[ RECORD 1 ]-------+------------------------------
relid               | 16514
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 0
seq_tup_read        | 0
idx_scan            | 36864
idx_tup_fetch       | 36864
n_tup_ins           | 0
n_tup_upd           | 18432
n_tup_del           | 0
n_tup_hot_upd       | 16766
n_live_tup          | 100000
n_dead_tup          | 0
n_mod_since_analyze | 0
n_ins_since_vacuum  | 0
last_vacuum         | 2021-06-28 15:46:12.080963+03
last_autovacuum     |
last_analyze        |
last_autoanalyze    | 2021-06-28 15:45:43.07853+03
vacuum_count        | 1
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 1
```

Time: 13,532 ms

В результате увидим следующую информацию:

- relid – идентификатор базы;
- schemaname – имя схемы;
- relname – имя таблицы;
- seq_scan – сколько раз выполнялось последовательное чтение всей таблицы;
- seq_tup_read – количество строк, прочитанных при последовательных чтениях;
- idx_scan – количество сканирований по индексу;
- idx_tup_fetch – количество строк, отобранных при сканированиях по индексу;
- n_tup_ins – количество вставленных строк;
- n_tup_upd – количество обновлённых строк (UPDATE);
- n_tup_del – количество удалённых строк;
- n_tup_hot_upd – количество строк, обновлённых в режиме HOT (без отдельного изменения индекса);
- n_live_tup – оценочное количество строк;
- n_dead_tup – оценочное количество “мёртвых” строк;
- n_mod_since_analyze – оценочное число строк, изменённых в этой таблице, с момента последнего сбора статистики;
- n_ins_since_vacuum – примерное число строк, вставленных в эту таблицу с момента последнего сбора статистики;
- last_vacuum – когда последний раз работал VACUUM;
- last_autovacuum – когда последний раз работал AUTOVACUUM;
- last_analyze – когда последний раз VACUUM собирал статистику;
- last_autoanalyze – когда последний раз AUTOVACUUM собирал статистику;
- vacuum_count – сколько раз VACUUM выполнялся;
- autovacuum_count – сколько раз AUTOVACUUM выполнялся;
- analyze_count – сколько раз вручную собирали статистику;
- autoanalyze_count – сколько раз AUTOVACUUM собирал статистику.

Короче говоря, команда pgbench -T 10 admin_monitoring просто выполняла UPDATE в таблице.

Из представления pg_statio_all_tables посмотрим статистику в разрезе страниц, то есть сколько страниц было прочитано, изменено и так далее:

*postgres@admin_monitoring=# SELECT * FROM pg_statio_all_tables WHERE relid='pgbench_accounts'::regclass \gx*

-[RECORD 1]---+-----------------
relid | 16514
schemaname | public
relname | pgbench_accounts
heap_blks_read | 27
heap_blks_hit | 68962
idx_blks_read | 275
idx_blks_hit | 77326
toast_blks_read |
toast_blks_hit |
tidx_blks_read |
tidx_blks_hit |
Time: 11,974 ms

В результате мы видим:

- heap_blks_read – сколько страничек было прочитано с диска;
- heap_blks_hit – сколько страничек было прочитано из буферного кэша;
- idx_blks_read – сколько индексов было считано с диска;
- idx_blks_hit – сколько индексов было считано из буферного кэша.

Вдобавок можем посмотреть статистику по индексам. Во-первых, в разрезе строк (pg_stat_all_indexes), во-вторых, в разрезе страниц (pg_statio_all_indexes):

*postgres@admin_monitoring=# SELECT * FROM pg_stat_all_indexes WHERE relid='pgbench_accounts'::regclass \gx*
-[RECORD 1]-+---------------------
relid | 16514
indexrelid | 16528
schemaname | public
relname | pgbench_accounts
indexrelname | pgbench_accounts_pkey
idx_scan | 36864
idx_tup_read | 38794
idx_tup_fetch | 36864
Time: 11,698 ms
*postgres@admin_monitoring=# SELECT * FROM pg_statio_all_indexes WHERE relid='pgbench_accounts'::regclass \gx*
-[RECORD 1]-+---------------------
relid | 16514
indexrelid | 16528
schemaname | public
relname | pgbench_accounts
indexrelname | pgbench_accounts_pkey

idx_blks_read | 275
idx_blks_hit | 77326
Time: 11,315 ms

Просмотр статистики всей базе данных

В конце концов можем посмотреть глобальную статистику по всей базе данных из представления pg_stat_database:

*postgres@admin_monitoring=# SELECT * FROM pg_stat_database WHERE datname='admin_monitoring' \gx*
-[RECORD 1]---------+------------------------------
datid | 16507
datname | admin_monitoring
numbackends | 1
xact_commit | 18515
xact_rollback | 0
blks_read | 510
blks_hit | 243592
tup_returned | 277836
tup_fetched | 38458
tup_inserted | 18438
tup_updated | 55308
tup_deleted | 0
conflicts | 0
temp_files | 0
temp_bytes | 0
deadlocks | 0
checksum_failures | 0
checksum_last_failure |
blk_read_time | 10.429
blk_write_time | 0
stats_reset | 2021-06-28 15:43:53.618542+03
Time: 11,633 ms

В результате видим:

- tup_inserted – сколько строк было вставлено;
- tup_updated – сколько строк было изменено;
- blks_read – сколько страниц было прочитано с диска;
- blks_hit – сколько страниц было прочитано из буферного кэша;
- numbackends – сколько сейчас клиентов подключено к базе данных;
- xact_commit – сколько транзакций было успешно завершено;
- xact_rollback – сколько транзакций было откачено.

Просмотр статистики фоновой записи

Помимо всего прочего PostgreSQL ведёт статистику по фоновой записи. Для того чтобы её посмотреть, нужно заглянуть в представление pg_stat_bgwriter:

```
postgres@admin_monitoring=# CHECKPOINT;
CHECKPOINT
Time: 1,232 ms
postgres@admin_monitoring=# SELECT * FROM pg_stat_bgwriter \gx
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 5
checkpoints_req       | 2
checkpoint_write_time | 149479
checkpoint_sync_time  | 3
buffers_checkpoint    | 2095
buffers_clean         | 0
maxwritten_clean      | 0
buffers_backend       | 154
buffers_backend_fsync | 0
buffers_alloc         | 508
stats_reset           | 2021-06-28 15:44:14.892968+03
Time: 10,707 ms
```

В результате видно:

- buffers_clean – сколько страниц записал процесс фоновой записи;
- buffers_checkpoint – сколько страниц записал процесс checkpoint;
- buffers_backend – сколько страниц записывали фоновые процессы;
- checkpoints_timed – сколько раз процесс checkpoint срабатывал по таймеру;
- checkpoints_req – сколько раз процесс checkpoint срабатывал в связи с активной работой с таблицей.

4.3. Мониторинг индексов в PostgreSQL и их перестроение

Поиск неиспользуемых индексов

Иногда случается, что в базе данных некоторые индексы не нужны. То есть при создании базы сделали лишние или пересекающиеся индексы, а они на самом деле не используются. Лишние индексы занимают место, а также добавляют накладные расходы. Как только вы меняете таблицу, вам нужно обновить все индексы в ней. Поэтому мониторинг индексов необходим в PostgreSQL.

В представлении pg_stat_all_indexes в поле idx.scan вы можете увидеть количество обращений к индексу. Если там ноль, значит индекс не

используется и возможно его следует удалить. Но следует точно понимать, что вы делаете, и перепроверить все перед удалением таких индексов.

Посмотреть представление pg_stat_all_indexes для какой-нибудь таблицы можно так:

```
postgres@s-pg13:~$ psql -d data_lowlevel
Timing is on.
psql (13.3)
Type "help" for help.
postgres@data_lowlevel=# SELECT * FROM pg_stat_all_indexes WHERE relname='t';
 relid | indexrelid | schemaname | relname | indexrelname | idx_scan | idx_tup_read | idx_tup_fetch
-------+------------+------------+---------+--------------+----------+--------------+---------------
 16497 |      16504 | public     | t       | t_pkey       |        0 |            0 |             0
(1 row)
Time: 10,941 ms
```

Перестроение индексов

Помимо лишних индексов существует другая проблема – разрастание индексов. Например, у вас выросла таблица и индексы в ней тоже разрослись. Они перестали помещаться в одной странице и заняли ещё одну. Затем вы почистили табличку с помощью VACUUM, а страницы индексов всё равно останется две, просто в них образуются пустоты. Страницы могут только делиться, а склеиваться назад уже не могут (если не использовать VACUUM FULL).

Существует команда REINDEX которая занимается перестроением индексов. Можно перестроить как отдельный индекс, так и все индексы в таблице, в базе данных или во всей системе:

- *REINDEX INDEX индекс;*
- *REINDEX TABLE таблица;*
- *REINDEX DATABASE база;*
- *REINDEX SYSTEM;*

Команда VACUUM FULL перестраивает и таблицы, и индексы, а команда REINDEX только индексы. REINDEX также устанавливает эксклюзивную блокировку на таблицу.

Пересоздание индексов

Можно пересоздать индексы без эксклюзивной блокировки с помощью ключевого слова CONCURRENTLY:

```
postgres=# CREATE INDEX новый ON ... CONCURRENTLY;
postgres=# DROP INDEX старый;
```

С помощью такого перестроения индексов новый индекс будет строится дольше, но это не приведёт к блокировке таблицы. Также такое создание нового индекса может завершиться с ошибкой, тогда этот новый индекс нужно удалить и попробовать сделать новый индекс ещё раз.

Индекс может служить для поддержки ограничения целостности, например, для первичного ключа. В этом случае меняем индексы таким способом:

postgres=# CREATE UNIQUE INDEX новый ON ... CONCURRENTLY;

postgres=# ALTER TABLE ...

DROP CONSTRAINT старое,

ADD CONSTRAINT новое [UNIQUE|PRIMARI KEY] USING INDEX новый;

5. Troubleshooting

Устранение неполадок сбоев базы данных и проблем с подключением (Troubleshooting)

Вступление

Предполагаемый сбой базы данных может быть стрессовым событием для всех. Пользователи разочаровываются и беспокоятся, что их данные могут быть потеряны, разработчики пытаются найти причину, а заинтересованные стороны компании считают каждую минуту, которая может понести потерю дохода.

Восстановление базы данных в работоспособном состоянии в случае сбоя является приоритетом. Тем не менее, может быть трудно даже рассуждать о том, в чем может быть проблема в разгар такого стрессового события. Эта статья призвана помочь вам разобраться в некоторых из наиболее распространенных причин, по которым ваша база данных может быть неработоспособна, и что вы можете сделать, чтобы исправить это.

В статье рассматриваются различные области, на которые следует обратить внимание при работе с перебоями в работе баз данных, в том числе:

Проблема

Потенциальные проблемы

Действия

База данных, похоже, не принимает подключения

Проблемы с учетными данными пользователя базы данных, проблемы со строкой подключения, достигнутые ограничения подключения

Проверка журналов сервера базы данных для определения причины

Сетевые коммуникации кажутся нарушенными

Проблемы с VPC и брандмауэром, задержки и тайм-ауты между приложением и базой данных

Проверка правил брандмауэра, поиск задержек и тайм-аутов, проверка исчерпанных лимитов подключений

Сервер базы данных или сервер приложений аварийно завершает работу.

Потенциальные проблемы с объемом данных, приводящие к тому, что сервер базы данных и сервер приложений достигают пределов ресурсов

Поиск больших объемов возвращаемых данных, оптимизация запросов, добавление индексов

Сервер базы данных работает, но приложение не отображает данные

Потенциальные недавние изменения в коде, включая изменения схемы, непримененные миграции, искаженные запросы

Проверка журнала системы управления версиями для поиска последних изменений и возврата при необходимости

Проблемы с подключением к базе данных

Подсказки из журналов приложений

Если база данных не работает, журналы приложений могут дать представление о любых проблемах, связанных с приемом запросов или подключением к базе данных. Поскольку сервер приложений обрабатывает как клиентские, так и запросы к базе данных, он обычно регистрирует ошибки при возникновении проблем с базой данных.

Успешно ли сервер приложений обрабатывает подключения?

Сервер приложений, предоставляющий API и обслуживающий запрос путем запроса базы данных, имеет два типа подключений:

Пользовательские запросы к API, предоставляемые сервером приложений

Запросы сервера приложений к базе данных

При отладке сбоя имейте в виду, что проблемы с обоими типами подключений могут привести к сбою.

Платформы сервера приложений обычно поставляются со встроенным регистратором. Обычно регистрируется, что сервер может принимать подключения после запуска сервера.

Например, взгляните на эту строку журнала:

{"level":30,"time":1617808854673,"pid":96741,"hostname":"do-server-1","msg":"Server listening at http://0.0.0.0:8000"}

Журнал показывает, что сервер запущен и принимает подключения через порт 8000. Но недостаточно определить, произошла ли ошибка с момента запуска сервера.

Следующим шагом будет просмотр самых последних журналов и проверка кода состояния HTTP самых последних запросов к серверу. Обычно журналы запросов выглядят следующим образом:

{"level":30,"time":1617809865718,"pid":97326,"hostname":"do-server-1","reqId":5,"req":{"method":"POST","url":"/graphql","hostname":"0.0.0.0:3000","remoteAddress":"127.0.0.1","remotePort":53540},"msg":"incoming request"}

{"level":30,"time":1617809865719,"pid":97326,"hostname":"do-server-1","reqId":5,"res":{"statusCode":200},"responseTime":1.1810400485992432,"msg":"request completed"}

Каждый журнал входящих запросов содержит метку времени, URL-адрес и другие сведения о запросе. Первый журнал указывает на поступление запроса, а второй показывает, что запрос выполнен успешно с кодом состояния HTTP 200.

Были ли какие-либо ошибки базы данных в последнее время?

Поскольку вы пытаетесь отладить проблему, связанную с базой данных, вы должны специально искать неудачные запросы. Вы можете найти их, отфильтровав запросы с кодом состояния ответа 5xx, например, 500, что указывает на ошибку сервера.

Если вы заметили несколько запросов с кодом состояния 500, найдите точку, в которой они начались, и проверьте наличие других зарегистрированных ошибок.

Если база данных не работает или учетные данные базы данных неверны, вы можете увидеть это в журнале. Например:

Can't reach database server at `db-postgresql-563564.b.db.ondigitalocean.com`:`5432`

Приведенный выше журнал указывает, что сервер приложений не может получить доступ к базе данных. В таких случаях проверьте URL-адрес базы данных на сервере приложений.

Устранение неполадок с базами данных

Теперь, когда вы знаете, что сервер приложений запущен, но не может получить доступ к базе данных, следующим шагом будет определение причины. Лучше всего подойти к этому, устранив потенциальные источники проблемы.

Если вы используете управляемую службу для своей базы данных, сначала проверьте страницу состояния поставщика облачных услуг. Большинство облачных провайдеров предоставляют способ проверить,

работает ли их инфраструктура, например Digital Ocean, Google Cloud Platform и AWS.

После того, как вы исключили какие-либо проблемы с облачной платформой, следующее, что нужно проверить, это журналы или состояние экземпляра вашей базы данных.

Журналы управляемой базы данных

Если используемый поставщик управляемой базы данных работает, следующим шагом будет проверка журналов для конкретного экземпляра базы данных.

Большинство облачных платформ предоставляют способ просмотра журналов базы данных в дополнение к некоторым метрикам, таким как пропускная способность и статистика запросов. Макет и уровень детализации журналов отличаются от одного поставщика облака к другому. Тем не менее, большинство облачных провайдеров предлагают достаточный уровень детализации, чтобы помочь вам в поиске проблем.

Каждый поставщик облачных услуг также имеет свой способ доступа к журналам базы данных. Большинство из них делают журналы легко доступными со страницы управления развертыванием.

Например, DigitalOcean предоставляет вкладку под названием «Журналы и запросы», доступную непосредственно из меню управления развертыванием.

Этот раздел содержит несколько различных видов данных журнала, включая общий раздел «Последние журналы». Вы можете найти признаки проблем с подключением к базе данных или других проблем, которые могут возникнуть.

Проблемы с сетью

Проблемы, связанные с сетью, могут привести к тому, что база данных будет недоступна или будет казаться неработоспособной. В трехуровневом приложении, состоящем из клиентского уровня, уровня приложений (серверная часть) и уровня данных (база данных), могут возникать проблемы с сетью между тремя уровнями.

К числу вопросов, связанных с сетевым взаимодействием, относятся:

Проблемы с политикой VPC и брандмауэра

Задержка и тайм-ауты между приложением и базой данных

Если база данных не работает или не отвечает, и вы подозреваете, что это может быть связано с сетевыми проблемами, лучше сначала определить, заблокирован ли трафик для политик брандмауэра или есть реальные проблемы с сетью.

Примечание: Приведенные ниже рекомендации основаны на предположениях, которые могут не относиться к вашей архитектуре. Поэтому рекомендуется понимать режимы сбоев и их причины при отладке проблем с базой данных.

VPC

При выделении облачных ресурсов, таких как базы данных на облачных платформах, они изолированы в виртуальном частном облаке (VPC). На практике VPC служит частной сетью для ресурсов приложения и изолирован от общедоступного Интернета. Если база данных и приложение не находятся в одном облаке VPC (или в одном облаке), обычно необходимо настроить правила брандмауэра VPC, чтобы разрешить IP-доступ приложения к базе данных.

Правила брандмауэра

При развертывании приложения и базы данных на разных облачных платформах или разных VPC необходимо настроить брандмауэр для разрешения доступа приложения к базе данных.

В то время как некоторые политики брандмауэра используют политики черной дыры для молчаливого сброса трафика, который не разрешен, другие отправляют отклоненный ответ, который может помочь вам определить, несет ли брандмауэр ответственность за нарушение соединения. Аналогичным образом, если у вас есть ошибка типа «нет маршрута к хосту», это может означать, что сегмент сети не работает или логика маршрутизации неверна.

Кроме того, если IP-адрес вашего приложения является динамическим, он может измениться в любое время. При изменении IP-адреса приложения может не удается подключиться к базе данных до тех пор, пока брандмауэр не будет обновлен для разрешения нового IP-доступа.

Средства защиты правил брандмауэра

Как правило, лучше развертывать приложение и базу данных в одном и том же VPC и одном регионе, чтобы они взаимодействовали по частной сети. Это также предотвращает узкие места, которые могут

возникнуть в общедоступных сетях. Однако, если это невозможно, есть несколько возможных исправлений.

Первый подход заключается в использовании выделенного IP-адреса для вашего приложения и добавлении правила брандмауэра, разрешающего доступ с IP-адреса. Такой подход гарантирует, что вам не нужно обновлять правила брандмауэра, поскольку IP-адрес остается прежним.

Если использование выделенного IP-адреса невозможно из-за ограничений облачной платформы или подхода к развертыванию, например, бессерверного, рассмотрите возможность предоставления доступа к базе данных через общедоступные сети. Хотя такой подход снижает безопасность базы данных, он гарантирует, что приложение не будет испытывать внезапного простоя до тех пор, пока вы не обновите правила брандмауэра при изменении IP-адреса приложения. Кроме того, механизм аутентификации вашей базы данных остается основной линией защиты.

Обратите внимание, что открывать подключения к базе данных из общедоступного Интернета не рекомендуется. Там, где это возможно, мы рекомендуем ограничить подключения определенным IP-адресом или использовать такие методы, как пиринг VPC.

Задержка и тайм-ауты

Когда географическое и сетевое расстояние между приложением и базой данных является значительным, проблема может быть затруднена с увеличением задержки запросов и ошибок тайм-аута. В таких сценариях может потребоваться увеличение времени ожидания подключения к базе данных в приложении, чтобы избежать ошибок тайм-аута.

ORM и построители запросов содержат пул подключений к базе данных. Эти подключения обычно имеют конфигурацию времени ожидания подключения, которая определяет, как долго ждать до истечения времени ожидания при установлении соединения.

Рекомендуется задать начальное базовое значение времени ожидания и нагрузочно протестировать приложение, чтобы увидеть, как оно работает. В зависимости от количества неудачных запросов из нагрузочного теста может потребоваться настроить время ожидания и повторить попытку, пока не убедитесь, что тайм-ауты не являются причиной проблемы. Если конфигурация времени ожидания подключения

слишком мала, существует риск того, что эти тайм-ауты приведут к сбою запросов.

Лимит исчерпанного соединения

Еще одна распространенная проблема с базами данных на основе подключений, такими как MySQL и PostgreSQL, заключается в том, что вы можете быстро исчерпать лимит подключения базы данных. Базы данных, ориентированные на подключения, накладывают ограничение на количество открытых подключений к базе данных.

При развертывании приложения с использованием традиционной модели длительных процессов можно мультиплексировать множество входящих запросов на меньшее количество подключений к БД в пуле. Например, экземпляр одного сервера может обрабатывать 100 одновременных HTTP-запросов с пулом из 20 подключений БД.

Однако в бессерверных функциях каждый экземпляр функции может обрабатывать только 1 HTTP-запрос одновременно. Поскольку для каждого входящего запроса требуется по крайней мере одно подключение к базе данных, мультиплексирование подключений к базе данных невозможно.

Учитывая это, рекомендуется использовать ограничение подключений к базе данных, а затем распределить их между экземплярами сервера, которые подключаются к ней, таким образом, чтобы не исчерпать предел подключения базы данных. Например, предположим, что база данных имеет ограничение на количество подключений 20. При наличии двух экземпляров сервера приложений необходимо установить пул подключений каждого экземпляра сервера не более чем на 10 подключений.

Для бессерверных развертываний рассмотрите возможность запуска внешнего пула подключений, такого как PgBouncer — дополнительного компонента инфраструктуры, который поддерживает подключения к базе данных и мультиплексирует входящие запросы к базе данных из бессерверных функций.

Проблемы с объемом данных

По мере роста приложения объем данных для этого приложения, скорее всего, также будет расти. Ключевым фактором как для производительности, так и для времени безотказной работы базы данных является объем данных, обрабатываемых для удовлетворения заданного запроса. Неэффективные запросы, написанные для приложения, когда

общий объем данных невелик, могут превратиться в узкие места, которые могут снизить производительность и вызвать сбои при росте этих данных.

Большие объемы данных могут влиять на три местоположения:

- Сервер базы данных
- Сервер приложений
- Клиент
- Сервер базы данных.

Когда запрос к базе данных выполняется с сервера приложений, сервер базы данных отвечает за извлечение запрошенных данных и их отправку обратно на сервер приложений. При этом сервер базы данных должен сначала отсканировать полученные данные в собственную память. Когда объемы данных невелики, процесс сканирования данных в память и пересылки их на сервер приложений тривиален. Однако если из запроса возвращается очень большой объем данных, недостаточно подготовленный сервер базы данных может разрушиться под нагрузкой.

Этот сценарий часто возникает, когда запросы, которые изначально были достаточными с небольшим общим размером базы данных, недостаточно оптимизированы по мере увеличения размера данных.

Рассмотрим сценарий, в котором приложению необходимо отобразить список контактов для пользователя. Типичный запрос к базе данных для предоставления этих данных, вероятно, выбирает данные, ограниченные для этого пользователя.

SELECT * from contacts WHERE userId = 123.

В зависимости от потребностей приложения серверу приложений может нечего делать, кроме как переслать результаты клиенту. Этот запрос вряд ли вызовет проблемы с объемом данных.

Однако подумайте, что произойдет, если в приведенном выше запросе не будет пункта. WHERE

SELECT * from contacts;

Если вместо этого запрос запрашивал все контакты в базе данных, а сервер приложений отвечал за фильтрацию списка на основе пересылки результатов клиенту, разница в объеме данных могла быть астрономической. userId

При относительно небольших общих объемах данных этот неадекватный запрос может остаться незамеченным. По мере роста объемов данных снижение производительности может стать заметным, но не полностью нарушить работу приложения. Однако по мере этого

сервер базы данных будет загружаться сверх того, для чего он может быть подготовлен. Поскольку объемы данных продолжают расти еще больше, может быть достигнут переломный момент, когда сервер базы данных больше не сможет справиться с нагрузкой.

Сервер приложений

Подобно тому, как сервер базы данных не имеет неограниченной емкости для обработки больших объемов данных, то же самое верно и для сервера приложений. Возможно, у вас больше контроля над емкостью и размером сервера приложений, чем для сервера баз данных, но увеличение емкости на сервере приложений может быть ненужным.

Для обработки большого объема данных, возвращаемых с сервера базы данных, требуется время и ресурсы. Если сервер приложений отвечает на запрос клиентского приложения, длительное ожидание на сервере приложений может быть проблематичным. Многие провайдеры облачного хостинга применяют тайм-аут запроса по прошествии нескольких минут. Веб-браузеры будут автоматически повторять запросы через несколько минут после того, как запрос находится в состоянии «ожидания», создавая дополнительную нагрузку на систему.

Клиент

Клиентские приложения могут быть наиболее подвержены узким местам, вызванным большими объемами данных. В отличие от сервера приложений и сервера баз данных, где у вас может быть возможность увеличить емкость, клиентские приложения, которые выполняются в браузере или на мобильных устройствах, подвержены ограничениям браузера, операционной системы или того и другого.

Отправка чрезмерно большого объема данных в клиентское приложение приведет к задержкам в обработке и может серьезно ухудшить скорость отклика. Как только заданный размер данных будет достигнут, браузер может полностью перестать отвечать и в конечном итоге аварийно завершить работу.

Если очень большие объемы данных передаются из базы данных в клиент, снижение производительности будет ощущаться на каждом этапе на этом пути. Эта трифекта приведет к длительному времени загрузки и, возможно, к предполагаемому отключению в любой из этих точек. Усугубляя проблему, устранение неполадок там, где существует узкое место, может стать затруднительным.

Средства защиты от размера данных

Исправление снижения производительности и простоев, вызванных проблемами с объемом данных, почти всегда заключается в ограничении объема данных, возвращаемых сервером базы данных. Это облегчит проблемы на сервере базы данных, сервере приложений и клиенте.

Область действия данных с предложениями WHERE

В приведенном выше примере вымышленного запроса был запрошен весь список контактов для базы данных. Это означает, что сервер базы данных должен сканировать всю таблицу в память перед фильтрацией до желаемых результатов, что может создать огромную нагрузку на сервер базы данных и сервер приложений.

Начните с поиска запросов, которые приводят к слишком широкой сети. Скорее всего, обработка этих запросов на сервере базы данных занимает много времени. Просмотр журналов для длительных запросов может дать представление о том, где это происходит. Вместо того, чтобы возвращать много результатов, а затем фильтровать их с помощью кода приложения, создайте свои запросы, чтобы вернуть подмножество результатов, которые действительно необходимы. В SQL используйте это предложение, чтобы получить конкретную информацию о фактических данных, которые вам действительно нужны. WHERE

Разбиение данных на страницы с помощью и LIMITOFFSET

Если для клиентских приложений действительно требуются большие объемы данных, рекомендуется ввести разбивку на страницы.

Разбиение на страницы — это шаблон проектирования, который ограничивает общее количество записей, запрашиваемых и возвращаемых в данный момент времени. Если пользователь хочет видеть больше записей, он должен запросить их явно.

Этот шаблон проектирования чаще всего реализуется с помощью предложений and. Выбрав пропуск определенного количества записей, а также ограничив количество возвращаемых результатов, серверы клиентов и приложений могут обеспечить разбиение на страницы. LIMITOFFSET

Запрос для первой страницы может выглядеть следующим образом:

SELECT * from contacts WHERE userId = 123 LIMIT 25 OFFSET 0;

Затем запрос для второй страницы просто установит значение значения, чтобы получить следующую «страницу». LIMITOFFSET

SELECT * from contacts WHERE userId = 123 LIMIT 25 OFFSET 25;

Устранение неполадок с томами данных

Большинство баз данных предлагают средства, которые могут помочь в устранении неполадок с объемом данных. В частности, многие базы данных предлагают команду, которая отображает план выполнения запроса и дает представление о любых связанных с этими проблемами. EXPLAIN

Например, в PostgreSQL команда показывает план запроса для рассматриваемых инструкций и отображает ориентировочную стоимость выполнения. Это оценка того, сколько времени потребуется для выполнения запроса. EXPLAIN

Эти сведения могут быть полезны для сужения конкретных таблиц или инструкций запроса, которые вызывают узкие места.

Помимо вызова команды вручную, многие поставщики облачных баз данных предлагают способ автоматической визуализации запросов, вызывающих проблемы. Эта информация часто встречается в представлении «Медленные запросы» для поставщиков облачных услуг. Проверьте раздел Медленные запросы поставщика базы данных, чтобы узнать, какие запросы могут негативно повлиять на приложение. EXPLAIN

Добавление индексов

Проблемы, связанные с большими объемами данных, часто можно устранить с помощью индексов.

Индексы для таблицы базы данных можно концептуализировать так же, как индекс в книге. Если вы ищете информацию по определенной теме в книге, вам потребуется много времени, чтобы прочитать всю книгу, чтобы найти ее. Вместо этого вы можете обратиться к индексу, чтобы найти конкретную тему, представляющую интерес. Если он существует, он укажет вам на конкретные страницы, на которых обсуждается тема в книге.

Та же концепция применяется к индексам баз данных. Добавление индексов в таблицу базы данных на основе общих шаблонов доступа позволяет выполнять быстрый поиск.

Например, если у вас есть таблица базы данных пользователей, содержащая столбец для, и эта таблица запрашивается для поиска пользователей на основе их электронной почты, это помогает повысить производительность, чтобы создать индекс для этой таблицы на основе столбца. Без индекса вся таблица будет просканирована, чтобы найти пользователя по его адресу электронной почты. Если размер таблицы

очень большой, это может создать значительные проблемы с производительностью. Однако, если для этого столбца создается индекс, поиск будет быстрым, поскольку поиск определенного сообщения электронной почты сразу же укажет на точную строку, которая необходима .emailemail

Добавление индексов в базу данных требует планирования и рассмотрения и должно быть выполнено до того, как база данных вызовет сбои в работе приложения. Однако, если вы можете найти конкретный запрос или запросы, ответственные за сбой, может быть полезно добавить индекс, чтобы немедленно облегчить нагрузку.

Взлом изменений кода

Предполагаемый сбой базы данных может быть прослежен до недавних критических изменений, внесенных в клиентский или серверный код. В этих случаях, скорее всего, не сама база данных испытывает сбой, а код, используемый для извлечения данных, их обработки и возврата клиенту, может сломаться.

Изменения кода, которые могут привести к прерываниям, обычно делятся на три категории:

Запросы к базе данных

Серверный код

Код клиента

Запросы к базе данных

Изменения в запросах к базе данных, даже если они небольшие, могут повлиять на приложение. При использовании необработанного SQL инструкция запроса может стать недействительной при недавнем изменении кода. Если сам запрос по-прежнему действителен, возможно, он был изменен для возврата результатов, которые больше не обрабатываются другими частями приложения.

Проверьте изменения системы управления версиями, чтобы найти последние изменения в доступе к базе данных (инструкции SQL или использование ORM). Попытайтесь изолировать запросы, связанные с определенными областями приложения, которые затронуты.

Если нет признаков изменений в доступе к базе данных, другая возможность заключается в том, что схема базы данных изменилась, но миграция не была выполнена. Найдите все недавние миграции, которые, возможно, все еще необходимо применить к рабочей базе данных.

Клиентский и серверный код

Изменения кода приложения на клиенте или сервере, даже если они небольшие, могут привести к сбою базы данных. Есть много конкретных проблем, которые могут быть в корне проблемы, но некоторые примеры включают в себя:

Возможно, клиент вызывает конечную точку сервера, которая больше не существует

Конечная точка сервера изменила правила проверки полезных данных или параметров запроса

Сервер пытается получить доступ к свойствам возвращенных данных, которые стали недействительными в результате недавнего изменения схемы

Заключение

Перебои в работе базы данных, независимо от основной причины, могут быть очень стрессовыми событиями. В спешке, чтобы попытаться восстановить и запустить приложение, может быть трудно правильно рассуждать о том, откуда могут возникнуть проблемы с базой данных.

Каждый сбой базы данных уникален, и при его возникновении не существует универсального решения. Тем не менее, хорошее знание возможных проблем и связанных с ними решений может помочь вам более эффективно определить причину проблемы и сократить время, необходимое для восстановления и работы.

Как и в случае с большинством проблем, лучше всего ознакомиться с потенциальными решениями проблем до того, как они произойдут. Мы надеемся, что это руководство поможет вам на пути к достижению именно этого.



МОДУЛЬ 7. МАСШТАБИРУЕМОСТЬ И ОТКАЗОУСТОЙЧИВОСТЬ

1. Кластеризация
2. Pacemaker
3. Резервное копирование, типы, виды, сравнение популярных решений
4. Балансировка нагрузки. HAProxy/Nginx
5. Disaster recovery
6. Keepalived/vrrp
7. Отказоустойчивость в облаке

1. Кластеризация

Кластеризация — это разбиение множества объектов на подмножества (кластеры) по заданному критерию. Каждый кластер включает максимально схожие между собой объекты. Представим переезд: нужно разложить по коробкам вещи по категориям (кластерам) — например одежда, посуда, декор, канцелярия, книги. Так удобнее перевозить и раскладывать предметы в новом жилье. Процесс сбора вещей по коробкам и будет кластеризацией.

Критерии кластеризации определяет человек, а не алгоритм, — этим она отличается от классификации. Этот метод машинного обучения (Machine Learning) часто применяют в различных неструктурированных данных — например если нужно автоматически разбить коллекцию изображений на мини-группы по цветам.

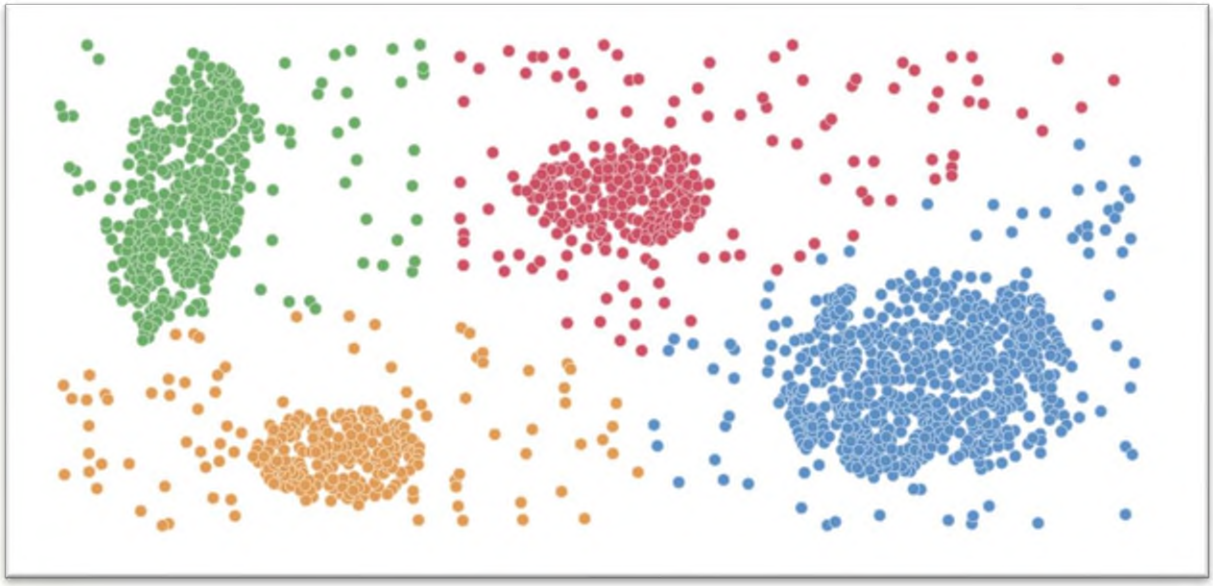


Рисунок 1. Визуализация кластеризации

Типы входных данных

- **Признаковое описание объектов**

Объект описывается при помощи набора характеристик. Признаки бывают числовые и категориальные. Например, можно кластеризовать группу покупателей на основе их покупок в интернет-магазине. В качестве входных данных будут средний чек, возраст, количество покупок в месяц, любимая категория покупок и другие критерии.

- **Матрица расстояний между выделенными объектами**

Это симметричная таблица, где по строкам и столбцам расположены объекты, а на пересечении — расстояние между ними: например, таблица с

расстояниями между отелями в разных городах. Такой способ может помочь выделить кластеры отелей, которые сгруппированы в одной и той же локации.

Цели кластеризации

- **Сжатие данных**

Кластеризация актуальна, если исходная выборка слишком большая. В результате от каждого кластера остается по одному типичному представителю. Количество кластеров может быть любым — здесь важно обеспечить максимальное сходство объектов внутри каждой группы.

- **Поиск паттернов внутри данных**

Разбиение объектов на кластеры позволяет добавить дополнительный признак каждому объекту. Так, если в результате кластерного анализа выявилось, что определенный покупатель относится к первому кластеру, и мы знаем, что первый кластер — это кластер людей, которые тратят большое количество денег на покупки по средам, то можно сказать, что это покупатель приобретает продукты в основном по средам.

- **Поиск аномалий**

В этом случае выделяют нетипичные объекты, не подходящие ни к одному сформированному кластеру. Интересны отдельные объекты, которые не вписываются ни в одну из сформированных групп.

Методы кластеризации

Общепринятой классификации методов нет, но есть несколько групп подходов.

1. Вероятностный подход. В рамках него предполагается, что каждый из объектов относится к одному из классов.

- EM-алгоритм применяется для нахождения оценок максимального правдоподобия параметров вероятностных моделей (если есть зависимость от скрытых переменных).
- K-средних — алгоритм минимизирует суммарное квадратичное отклонение точек кластеров от их центров.
- Алгоритмы семейства FOREL основаны на идее объединения объектов в один кластер в областях их максимальной концентрации.

2. Подходы с учетом систем искусственного интеллекта. Большая условная группа методов, разнится с методической точки зрения.

- Метод нечеткой кластеризации C-средних предполагает разбиение имеющегося множества элементов на определенное число нечетких множеств. Метод является усовершенствованным вариантом K-средних.

- Нейронная сеть Кохонена — класс нейронных сетей со слоем Кохонена, состоящим из линейных формальных нейронов.
- Генетический алгоритм — алгоритм поиска, применяемый для решения задач оптимизации и моделирования с помощью случайного подбора, вариации и комбинирования искомых параметров. Используются механизмы, аналогичные естественному отбору в природе.

3. Иерархический подход.

Предполагает наличие вложенных групп — кластеров разного порядка. Выделяются агломеративные и дивизионные (объединительные и разделяющие) алгоритмы. В зависимости от количества признаков могут выделяться политетические (используют при сравнении нескольких признаков одновременно) и монотетические (используют при применении одного признака) методы классификации.

Как описать кластеризацию формально?

В кластеризации имеют дело с множеством объектов (X) и множеством номеров кластеров (Y). Задана функция расстояния между объектами (ρ). Нужно разбить обучающую выборку на кластеры, так чтобы каждый кластер состоял из объектов, близких по метрике ρ, а объекты разных кластеров существенно отличались. При этом каждому объекту приписывается номер кластера y(i).

Алгоритм кластеризации — это функция, которая любому объекту X ставит в соответствие номер кластера Y.

Множество Y в некоторых случаях известно заранее, однако чаще ставится задача определить оптимальное число кластеров, с точки зрения того или иного критерия качества кластеризации.

Кластеризация (обучение без учителя) отличается от классификации (обучения с учителем) тем, что метки объектов из обучающей выборки yi изначально не заданы, и даже может быть неизвестно само множество Y.

Решение задачи кластеризации объективно неоднозначно по ряду причин:

- Не существует однозначного критерия качества кластеризации. Известен ряд алгоритмов, осуществляющих разумную кластеризацию "по построению", однако все они могут давать разные результаты. Следовательно, для определения качества кластеризации и оценки выделенных кластеров необходим эксперт предметной области;
- Число кластеров, как правило, заранее не известно и выбирается по субъективным критериям. Даже если алгоритм не требует изначального знания

о числе классов, конкретные реализации зачастую требуют указать этот параметр;

- Результат кластеризации существенно зависит от метрики.

Число кластеров фактически является гиперпараметром для алгоритмов кластеризации.

2. Pacemaker

Pacemaker — менеджер ресурсов кластера (Cluster Resource Manager), задачей которого является достижение максимальной доступности управляемых им ресурсов и защита их от сбоев как на уровне самих ресурсов, так и на уровне целых узлов кластера.

Ключевые особенности Pacemaker:

- обнаружение и восстановление сбоев на уровне узлов и сервисов;
- возможность гарантировать целостность данных путем ограждения неисправных узлов;
- поддержка одного или нескольких узлов на кластер;
- поддержка нескольких стандартов интерфейса ресурсов (все, что может быть написано сценарием, может быть кластеризовано);
- независимость от подсистемы хранения — общий диск не требуется;
- поддержка и кворумных и ресурсозависимых кластеров;
- автоматически реплицируемая конфигурация, которую можно обновлять с любого узла;
- возможность задания порядка запуска ресурсов, а также их совместимости на одном узле;
- поддерживает расширенные типы ресурсов: клоны (когда ресурс запущен на множестве узлов) и дополнительные состояния (master/slave и подобное);
- единые инструменты управления кластером с поддержкой сценариев.

Архитектура Pacemaker представляет собой три уровня:

- кластеронезависимый уровень — на этом уровне располагаются ресурсы и их скрипты, которыми они управляются и локальный демон, который скрывает от других уровней различия в стандартах, использованных в скриптах;
- менеджер ресурсов (Pacemaker) — реагирует на события, происходящие в кластере: отказ или присоединение узлов, ресурсов, переход узлов в сервисный режим и другие административные действия. Pacemaker, исходя из сложившейся ситуации, делает расчет наиболее оптимального состояния кластера и дает команды на выполнение

действий для достижения этого состояния (остановка/перенос ресурсов или узлов);

- информационный уровень (Corosync) — на этом уровне осуществляется сетевое взаимодействие узлов, т.е. передача сервисных команд (запуск/остановка ресурсов, узлов и т.д.), обмен информацией о полноте состава кластера (quorum) и т.д.

Узел (node) кластера представляет собой физический сервер или виртуальную машину с установленным Pacemaker. Узлы, предназначенные для предоставления одинаковых сервисов, должны иметь одинаковую конфигурацию.

Ресурсы, с точки зрения кластера, это все используемые сущности — сервисы, службы, точки монтирования, тома и разделы. При создании ресурса потребуется задать его класс, тип, провайдера и собственно имя с дополнительными параметрами. Ресурсы поддерживают множество дополнительных параметров: привязку к узлу (resource-stickiness), роли по умолчанию (started, stoped, master) и т.д. Есть возможности по созданию групп ресурсов, клонов (работающих на нескольких узлах) и т.п.

Связи определяют привязку ресурсов к узлу (location), порядок запуска ресурсов (ordering) и совместное их проживание на узле (colocation).

Настройка узлов кластера

Для функционирования отказоустойчивого кластера необходимо, чтобы выполнялись следующие требования:

- дата и время между узлами в кластере должны быть синхронизированы;
- должно быть обеспечено разрешение имён узлов в кластере;
- сетевые подключения должны быть стабильными;
- у узлов кластера для организации изоляции узла (fencing) должны присутствовать функции управления питанием/перезагрузкой с помощью IPMI(ILO);
- следующие порты могут использоваться различными компонентами кластеризации: TCP-порты 2224, 3121 и 21064 и UDP-порт 5405 и должны быть открыты/доступны.

В примере используется следующая конфигурация:

node01 — первый узел кластера (IP 192.168.0.113/24);

node02 — второй узел кластера (IP 192.168.0.145/24);

node03 — третий узел кластера (IP 192.168.0.132/24);

192.168.0.251 — виртуальный IP по которому будет отвечать один из узлов.

Дальнейшие действия следует выполнить на всех узлах кластера.

Рекомендуется использовать короткие имена узлов. Для изменения имени хоста без перезагрузки, можно воспользоваться утилитой hostnamctl:

hostnamectl set-hostname node01

Настройка разрешений имён узлов

Следует обеспечить взаимно-однозначное прямое и обратное преобразование имён для всех узлов кластера. Желательно использовать DNS, в крайнем случае, можно обойтись соответствующими записями в локальных файлах /etc/hosts на каждом узле:

 # echo "192.168.0.113 node01" >> /etc/hosts

 # echo "192.168.0.145 node02" >> /etc/hosts

 # echo "192.168.0.132 node03" >> /etc/hosts

Проверка правильности разрешения имён:

 # ping node01

PING node01 (192.168.0.113) 56(84) bytes of data.

64 bytes from node01 (192.168.0.113): icmp_seq=1 ttl=64 time=0.352 ms

 # ping node02

PING node02 (192.168.0.145) 56(84) bytes of data.

64 bytes from node02 (192.168.0.145): icmp_seq=1 ttl=64 time=0.635 ms

Настройка ssh-подключения между узлами

- При настройке ssh-подключения для root по ключу необходимо убрать комментарии в файле /etc/openssh/sshd_config для строк:

PermitRootLogin without-password

PubkeyAuthentication yes

AuthorizedKeysFile /etc/openssh/authorized_keys/%u /etc/openssh/authorized_keys2/%u .ssh/authorized_keys .ssh/authorized_keys2

PasswordAuthentication yes

- Кроме того, полезно добавить в /etc/openssh/sshd_config директиву:

AllowGroups sshusers

- создать группу sshusers:

 # groupadd sshusers

- и добавить туда пользователей, которым разрешено подключаться по ssh:

 # gpasswd -a <username> sshusers

- Создать новый ключ SSH без пароля (параметр -N):

 # ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519 -N ""

Незащищенные ключи SSH (без пароля) не рекомендуются для серверов, открытых для внешнего мира.

- Скопировать публичную часть SSH-ключа на другие узелы кластера:

 # ssh-copy-id -i ~/.ssh/id_ed25519.pub user@node02

 # ssh-copy-id -i ~/.ssh/id_ed25519.pub user@node03

В результате получаем возможность работы с домашними каталогами пользователя user удалённого узла — копировать к себе и от себя, удалять, редактировать и т.д.

- Скопировать публичную часть SSH-ключа на все узлы кластера для администратора. Для этого подключиться к каждому узлу и под root скопировать публичную часть ключа:

 # ssh user@node02

user@node02 $ su -

node02 # cat /home/user/.ssh/authorized_keys >> /root/.ssh/authorized_keys

node02 # exit

user@node02 $ exit

Каталог /root/.ssh при этом должен существовать.

- Убедиться, что теперь можно запускать команды удалённо, без пароля:

 # ssh node02 -- uname -n

node02

Установка кластерного ПО и создание кластера

Для управления кластером Pacemaker можно использовать утилиты pcs или crm (пакет crmsh).

- Установить на всех узлах необходимые пакеты:

apt-get install corosync resource-agents pacemaker pcs

Данные пакеты не входят в состав ISO-образа дистрибутива, их можно установить из репозитория p9. О добавлении репозиториев с использованием графических приложений вы можете почитать в разделе Добавление репозиториев.

Пакет resource-agent — содержит агенты ресурсов (набор скриптов) кластера, соответствующие спецификации Open Cluster Framework (OCF), используемые для взаимодействия с различными службами в среде высокой доступности, управляемой менеджером ресурсов Pacemaker. Если есть

необходимость управлять дополнительными ресурсами, следует установить недостающий пакет resource-agents-*:

*$ apt-cache search resource-agents**

Пакет pcs (pacemaker/corosync configuration system) — утилита для управления, настройки и мониторинга кластера. Управляется как через командную строку, так и через веб-интерфейс.

При установке Pacemaker автоматически будет создан пользователь hacluster. Для использования **pcs**, а также для доступа в веб-интерфейс нужно задать пароль пользователю hacluster (одинаковый на всех узлах):

 # passwd hacluster

- Запустить и добавить в автозагрузку службу pcsd:

 # systemctl enable --now pcsd

- Настроить аутентификацию (на одном узле):

 # pcs host auth node01 node02 node03 -u hacluster

Password:

node02: Authorized

node01: Authorized

node03: Authorized

После этого кластером можно управлять с одного узла.

- Создать кластер:

 # pcs cluster setup newcluster node01 node02 node03

Destroying cluster on hosts: 'node01', 'node02', 'node03'...

node03: Successfully destroyed cluster

node01: Successfully destroyed cluster

node02: Successfully destroyed cluster

Requesting remove 'pcsd settings' from 'node01', 'node02', 'node03'

node01: successful removal of the file 'pcsd settings'

node03: successful removal of the file 'pcsd settings'

node02: successful removal of the file 'pcsd settings'

Sending 'corosync authkey', 'pacemaker authkey' to 'node01', 'node02', 'node03'

node01: successful distribution of the file 'corosync authkey'

node01: successful distribution of the file 'pacemaker authkey'

node03: successful distribution of the file 'corosync authkey'

node03: successful distribution of the file 'pacemaker authkey'

node02: successful distribution of the file 'corosync authkey'

node02: successful distribution of the file 'pacemaker authkey'

Sending 'corosync.conf' to 'node01', 'node02', 'node03'
node01: successful distribution of the file 'corosync.conf'
node02: successful distribution of the file 'corosync.conf'
node03: successful distribution of the file 'corosync.conf'
Cluster has been successfully set up.

- Запустить кластер:

 # pcs cluster start --all
node02: Starting Cluster...
node03: Starting Cluster...
node01: Starting Cluster...

- Настройка автоматического включения кластера при загрузке:

 # pcs cluster enable --all
node01: Cluster Enabled
node02: Cluster Enabled
node03: Cluster Enabled

- Проверка состояния кластера:

 # pcs status cluster
Cluster Status:
Cluster Summary:
** Stack: corosync*
** Current DC: node02 (version 2.0.3-alt2-4b1f869f0) - partition with quorum*
** Last updated: Thu Jan 28 13:26:38 2021*
** Last change: Thu Jan 28 13:27:05 2021 by hacluster via crmd on node02*
** 3 nodes configured*
** 0 resource instances configured*
Node List:
** Online: [node01 node02 node03]*
PCSD Status:
node01: Online
node02: Online
node03: Online

- Проверка синхронизации узлов кластера:

 # corosync-cmapctl | grep members
runtime.members.1.config_version (u64) = 0
runtime.members.1.ip (str) = r(0) ip(192.168.0.113)
runtime.members.1.join_count (u32) = 1
runtime.members.1.status (str) = joined

runtime.members.2.config_version (u64) = 0
runtime.members.2.ip (str) = r(0) ip(192.168.0.145)
runtime.members.2.join_count (u32) = 1
runtime.members.2.status (str) = joined
runtime.members.3.config_version (u64) = 0
runtime.members.3.ip (str) = r(0) ip(192.168.0.132)
runtime.members.3.join_count (u32) = 1
runtime.members.3.status (str) = joined

- Веб-интерфейс управления кластером по адресу https://<имя-компьютера>:2224 (в качестве имени компьютера можно использовать имя или IP-адрес одного из узлов в кластере). Потребуется пройти аутентификацию (логин и пароль учётной записи hacluster).
- После входа в систему на главной странице отображается страница «Управление кластерами». На этой странице перечислены кластеры, которые в настоящее время находятся под управлением веб-интерфейса. При выборе кластера отображается информация о кластере.
- Чтобы добавить существующий кластер в веб-интерфейс, необходимо нажать кнопку Add Existing и в открывшемся окне ввести имя или IP-адрес любого узла в кластере.

Настройка параметров кластера

Настройки кластера можно просмотреть, выполнив команду:

 # pcs property
Cluster Properties:
cluster-infrastructure: corosync
cluster-name: newcluster
dc-version: 2.0.3-alt2-4b1f869f0
have-watchdog: false
stonith-enabled: false

Кворум

Кворум определяет минимальное число работающих узлов в кластере, при котором кластер считается работоспособным. По умолчанию, кворум считается неработоспособным, если число работающих узлов меньше половины от общего числа узлов.

Отключить эту политику, например, если узла всего два, можно выполнив команду:

 # pcs property set no-quorum-policy=ignore

Настройка ресурсов

Настроим ресурс, который будет управлять виртуальным IP-адресом. Этот адрес будет мигрировать между узлами, предоставляя одну точку входа к ресурсам, заставляя работать несколько узлов как одно целое устройство для сервисов.

Команда создания ресурса виртуального IP-адреса с именем ClusterIP с использованием алгоритма ресурсов ocf (каждые 20 секунд производить мониторинг работы, в случае выхода из строя узла необходимо виртуальный IP переключить на другой узел):

 # pcs resource create ClusterIP ocf:heartbeat:IPaddr2 ip=192.168.0.251 cidr_netmask=24 op monitor interval=20s

Для того чтобы добавить ресурс в веб-интерфейсе, необходимо перейти на вкладку RESOURCES, нажать кнопку Add и задать параметры ресурс.

- Список доступных стандартов ресурсов:

 # pcs resource standards
lsb
ocf
service
systemd

- Список доступных поставщиков сценариев ресурсов OCF:

 # pcs resource providers
heartbeat
pacemaker
redhat

- Список всех агентов ресурсов, доступных для определённого поставщика OCF:

 # pcs resource agents ocf:heartbeat
aliyun-vpc-move-ip
anything
AoEtarget
apache
asterisk
…
Xinetd
zabbixserver
ZFS

- Статус кластера, с добавленным ресурсом:

 # pcs status

Cluster name: newcluster

Cluster Summary:

** Stack: corosync*

** Current DC: node02 (version 2.0.3-alt2-4b1f869f0) - partition with quorum*

** Last updated: Thu Jan 28 13:47:39 2021*

** Last change: Thu Jan 28 13:47:22 2021 by root via cibadmin on node01*

** 3 nodes configured*

** 1 resource instance configured*

Node List:

** Online: [node01 node02 node03]*

Full List of Resources:

** ClusterIP   (ocf::heartbeat:IPaddr2):   Started node01*

Daemon Status:

corosync: active/enabled

pacemaker: active/enabled

pcsd: active/enabled

- Если остановить кластер на узле node01:

 # pcs cluster stop node01

node01: Stopping Cluster (pacemaker)...

node01: Stopping Cluster (corosync)...

- ClusterIP начнёт работать на node02 (переключение произойдёт автоматически). Проверка статуса на узле node02:

 # pcs status

Cluster Summary:

** Stack: corosync*

** Current DC: node02 (version 2.0.3-alt2-4b1f869f0) - partition with quorum*

** Last updated: Thu Jan 28 15:02:02 2021*

** Last change: Thu Jan 28 13:48:12 2021 by root via cibadmin on node01*

** 3 nodes configured*

** 1 resource instance configured*

Node List:

** Online: [node02 node03]*

** OFFLINE: [node01]*

Full List of Resources:

** ClusterIP   (ocf::heartbeat:IPaddr2):   Started node02*

Daemon Status:

corosync: active/enabled
pacemaker: active/enabled
pcsd: active/enabled

3. Резервное копирование, типы, виды, сравнение популярных решений

Что такое резервное копирование?

Резервное копирование — один из способов сохранения информации на случай вредоносных атак, сбоев в системе, отказа жесткого диска и других проблем.

Резервное копирование (РК, Backup copy, бэкап) — создание копий личных или корпоративных файлов, папок или систем на дополнительных (локальных или облачных) носителях информации.

Резервное копирование необходимо для формирования архива данных, защищенного от изменений и повреждений, а также восстановления в случае повреждения или сбоев в первоисточнике.

Виды резервного копирования

Не существует какого-то одного универсального вида резервного копирования, который можно было бы использовать повсеместно и любых условиях. Скорость работы, стоимость, объём занимаемого бэкапом места, нагрузка на оборудование — технологии имеют свои достоинства и недостатки, которые нужно учитывать. Поэтому компании используют разные механизмы создания бэкапов.

Существуют следующие виды резервного копирования данных:

- **Полное резервное копирование**

Охватывает всю систему и существующие в ней файлы, так как создаёт полную копию всех данных. С точки зрения управления бэкапами и скорости восстановления данных — это наиболее простой и надёжный способ. Но достоинства полного резервного копирования оборачиваются двумя важными недостатками. Во-первых, для создания полного backup системы требуется очень много времени. Во-вторых, процесс создания дубликата системы серьёзно увеличивает нагрузку на сеть.

В связи с этим существует рекомендация уменьшить частоту создания полных backup-ов, делая их не чаще одного-двух раз в неделю. А в остальные дни создавать инкрементальные или дифференциальные резервные копии. Это разумно, поскольку такой частоты достаточно большинстве компаний. Безусловно, существуют ситуации, когда по каким-то причинам скриптам и ПО для создания бэкапов доверять нельзя. Тогда нужно позаботиться о более частом бэкапировании. Но если таких проблем нет, стоит ли лишний раз

перегружать сетевую инфраструктуру и занимать лишнее место на диске объёмными backup-ми всей системы?

Плюсы:

- Управлять полной резервной копией просто, так как все хранится в одной версии.
- Полная резервная копия предлагает самое быстрое время восстановления, поскольку все необходимые данные хранятся в одном и том же наборе резервных копий.
- Поиск файлов прост

Минусы:

- Для полной резервной копии вам потребуется больше места для хранения по сравнению с инкрементными и дифференциальными резервными копиями.
- Самый рискованный в случае потери данных, поскольку все ваши данные хранятся или резервируются в одном месте
- В некоторых случаях полное резервное копирование также может оказаться дорогостоящим, и поэтому вам, возможно, придется использовать комбинацию двух других форм резервного копирования, а не только полное резервное копирование для резервного копирования данных.
- Вероятность дубликатов высока
- Время, затрачиваемое на резервное копирование данных, велико
- Высокое потребление пропускной способности сети

Дифференциальное резервное копирование

При использовании этого метода копируются все файлы, которые изменялись с момента последнего полного резервного копирования. Восстановление системы производится путём загрузки последнего полного бэкапа вместе с актуальной дифференциальной копией.

Этот метод позволяет быстрее осуществлять копирование системы, так как полный backup выполняется реже. Но для восстановления нужно больше времени, поскольку приходится поочерёдно загружать сначала последний полный backup, и лишь затем — дифференциальный блок.

Плюсы:

- Меньше места, чем полные резервные копии
- Восстановление быстрее, чем инкрементное резервное копирование

- Меньшее потребление полосы пропускания, чем при полном резервном копировании

Минусы:

- Занимает больше места, чем инкрементное резервное копирование
- Восстановление медленнее, чем полное резервное копирование
- Большее потребление полосы пропускания, чем при инкрементном резервном копировании.

Инкрементальное резервное копирование

Метод, который считается наиболее технологичным. Регулярно создаются резервные копии с файлами, которые были изменены с момента последнего backup-а. Остальная часть системы не затрагивается. То есть в каждом новом бэкапе хранятся только новые изменения.

Это довольно быстрый метод резервного копирования, поскольку копируется меньшее количество данных. Да и нагрузка на сеть незначительна. Но с точки зрения восстановления он сложнее, ведь сначала нужно восстановить данные из последнего полного бэкапа, а после этого шаг за шагом накатывать каждую последующую инкрементальную копию. Если одна из них повреждена или удалена, могут возникнуть проблемы.

Плюсы:

- Меньше места для хранения потребляется
- Скорость, с которой делается резервная копия
- Нет дублирования файлов

Минусы:

- Хитрая реставрация
- Полное восстановление невозможно в случае отсутствия одной из инкрементных резервных копий.

Синтетическое резервное копирование

Метод, который часто называют гибридным или смешанным, так как он объединяет усилия трёх способов создания копий: полного, инкрементального и дифференциального. В жизни это выглядит следующим образом: к полной резервной копии регулярно добавляются дифференциальные или инкрементальные блоки. В какой-то момент все эти блоки «склеиваются» с полный копией, образуя новый файл — актуальный полный backup. Он же становится исходным для бэкапов, которые будут созданы позднее.

Плюсы:

- Резервное копирование и восстановление выполняются быстрее
- Проще управлять данными
- Снижается нагрузка на сеть

Минусы:

- Повышенная нагрузка на backup-сервер
- Может обходиться дороже

Где и как хранить резервные копии?

На сегодняшний день существует целый ряд популярных технологий бэкапов, к которым относят следующие:

- **FTP-бэкапы**

Этот метод предназначен для резервного копирования сайтов (или любых других данных) и предлагается многими хостинг-провайдерами в виде дополнительной услуги. Его суть заключается в том, что провайдер выделяет необходимый объем дискового пространства на отдельном FTP-сервере и сохраняет туда данные сайтов своих клиентов.

FTP-backup может выполняться пользователем вручную или же в автоматическом режиме по расписанию – через панель управления, планировщик задач или скрипты. Пространство на таком сервере используется для бекапов большого количества сайтов, каждый пользователь выкупает определенный объем в зависимости от своих потребностей.

- **HDD-бэкапы**

Этот метод представляет собой процесс резервного копирования данных на жесткий диск HDD (hard disk drive). При этом устройство может быть как стационарным, так и съемным (внешним). Выполнять данную операцию можно как в ручном режиме, так и автоматически при помощи специальных программ. Например, весьма популярным ПО для этих целей считается Handy Backup, он дает возможность архивировать данные на жестком диске по расписанию, сохраняя их в виде файла-архива.

- **CDP-бэкапы**

Технология CDP (continuous data protection – непрерывная защита данных) позволяет автоматически сохранять данные при каждом их изменении в виде резервных копий. Для этих целей сначала на сервер устанавливается специальная программа (CDP-агент), которая разделяет его на логические блоки и поблочно передает данные в хранилище бэкапов (CDP-сервер).

Далее программа отправляет на сервер только те блоки данных, которые как-либо изменялись. Технология CDP-бекапов заметно ускоряет процесс резервного копирования, а также помогает оптимизировать объем используемого для копий пространства на сервере.

- **Облачные бэкапы**

Облачный back-up – это один из самых удобных способов резервного копирования данных, что делает его весьма популярным. Он предоставляется многими провайдерами облачных услуг, которые позволяют размещать в своих хранилищах данные любых видов и объемов: от отдельных файлов до операционных систем и физических или виртуальных серверов целиком с возможностью их полного восстановления на другое оборудование или в виртуальной среде.

Также провайдеры cloud backup часто предоставляют пользователям ряд дополнительных услуг, в том числе шифрование данных до отправки бекапа в облако, возможность выбора места хранения данных, гранулярное восстановление приложений и т.д. Наиболее известными поставщиками услуг облачных бэкапов считаются сервисы Google Cloud, Amazon Web Services, DropBox, Comodo Online Backup и т.

- **Snapshot-бэкап**

Еще одной альтернативной технологией является snapshot – мгновенная копия данных сервера или другой файловой системы, сделанная в определенный период времени. Важно, что снэпшот не считается полноценным бэкапом, однако оптимально подходит для быстрого резервного копирования наиболее важных данных.

Главное преимущество snapshot-бэкапа – это высокая скорость передачи файлов, при нем работа компьютера или другого оборудования приостанавливается буквально на секунду. Это выгодно отличает снэпшот от иных видов бекапа, которые (как правило) требуют немало времени и серьезно нагружают систему. Кроме того, файлы со «снимками» весят немного и их легко можно объединить в серию из нескольких десятков снимков.

Как сделать бекап?

Точная последовательность действий при создании back-up copy напрямую зависит от ряда факторов. А именно, выбранного вами вида резервного копирования, формата, количества и объема копируемых файлов, используемого при этом ПО, способа копирования (ручное или автоматическое) и т.д.

Говоря в общем, процедура бекапа включает в себя такие операции:

- Подключение внешнего носителя данных (это может быть отдельный жесткий диск, флешка, облачное хранилище, отдельный FTP или CDP-сервер и т.д.)
- Копирование необходимых данных на внешний носитель (ручное или автоматическое)
- Проверка копии на предмет объема, целостности, актуальности данных и прочих важных факторов
- Сохранение backup-копии данных, а также безопасное отключение внешнего накопителя (при необходимости).

Правила резервного копирования

1. Специалисты рекомендуют делать бекапы с регулярной периодичностью, так как чем чаще вы копируете данные на внешний носитель, тем больше актуальной информации сможете сохранить в случае сбоя оборудования или падения/взлома системы. Например, делать бэкапы сайта нужно хотя бы раз в месяц, а лучше несколько раз.
2. Перед началом резервного копирования обязательно определите, какую информацию нужно скопировать в первую очередь. Это значит, что следует выделить группу наиболее важных данных и бекапить ее с максимальной частотой. Например, это может быть ПО и настройки серверов, базы персональных данных клиентов, объекты интеллектуальной собственности, юридическая документация, финансовые и налоговые отчеты, данные, от которых зависит работа сервисов и т.д.
3. Далее нужно определить два ключевых параметра восстановления данных: точку восстановления (recovery point objective) и время восстановления (recovery time objective). Первый из них обозначает тот объем данных, который допускается к утере с момента последнего бекапа. Чем меньше этот объем, тем чаще нужно делать back-up. Второй параметр обозначает максимально допустимый отрезок времени от момента утери данных до их восстановления из backup copy. Сократить его можно путем усиления вычислительных мощностей, используемых для восстановления данных.
4. При создании системы резервного копирования следует обратить внимание на объем аппаратных ресурсов, предназначенных для создания бекапов и восстановления данных из них. Сюда относится мощность центрального процессора, пропускная способность сети, объем оперативной памяти, объем пространства на сервере/облаке/жестком диске и т.д. Убедитесь, что мощности ваших ресурсов хватает для обеспечения оптимального быстродействия при архивировании и загрузке данных на внешний накопитель.

5. Не менее важно придерживаться принятого в IT-сфере правила бекапов под названием «3-2-1». Согласно нему, для создания эффективной системы резервного копирования и восстановления нужно создать минимум 3 копии необходимых данных, загрузить их минимум на 2 типа носителей (например, в облако и на CDP-сервер), а также хранить одну дополнительную копию на изолированном накопителе, не подключенном к основной IT-инфраструктуре (например, на флешке или внешнем жестком диске).

6. При наличии большого объема регулярно обновляемых данных рекомендуется автоматизировать процесс их резервного копирования и восстановления. Особенно актуально это будет в том случае, если у вашей компании или проекта есть несколько хранилищ для разных типов данных. Автоматизация позволит оперативно бекапить информацию при каждом ее изменении, сохраняя максимальный объем актуальных данных с возможностью восстановить их в кратчайшие сроки.

7. Наконец, после завершения процедуры бекапа данных обязательно стоит проверить созданные копии. Прежде всего, на предмет их полноты, целостности и читаемости. Нередко возникают ситуации, когда файлы загружаются в backup copy в неполном объеме или поврежденном виде, что делает их восстановление невозможным или бессмысленным.

4. Балансировка нагрузки. HAProxy/Nginx

Что такое HAProxy?

HAProxy (High Availability Proxy) – это популярный открытый балансировщик нагрузки TCP/HTTP и прокси-сервер для систем Linux, Solaris и FreeBSD. Он позволяет улучшить производительность и ошибкоустойчивость серверного окружения путём распределения рабочей нагрузки между несколькими серверами (например, между веб-сервером, сервером баз данных и сервером приложений). HAProxy используют многие проекты с высокой нагрузкой: GitHub, Imgur, Instagram и Twitter.

Терминология HAProxy

В балансировке нагрузки и проксировании существует много терминов и понятий. Рассмотрим самые основные из них.

- **Списки управления доступом (ACL)**

В балансировке нагрузки списки управления доступом используются для проверки определённых условий и выполнения действий (например, для выбора сервера, блокировки запроса и т.п.) согласно результату этой проверки. Списки контроля доступа обеспечивают гибкую переадресацию сетевого трафика на основе различных факторов (например, с учётом количества подключений к бэкэнду).

ACL выглядит так:

acl url_blog path_beg /blog

Такой ACL выполняется, если путь запроса пользователя начинается с /blog (например, http://yourdomain.com/blog/blog-entry-1).

- **Бэкэнд**

Бэкэнд – это набор серверов, которые обрабатывают запросы. Настройки этих серверов находятся в разделе backend конфигурационного файла HAProxy. В основном бэкэнд определяется

алгоритмом балансировки нагрузки;

и списком серверов и портов.

Бэкэнд может состоять из одного или нескольких серверов. В целом, чем больше на бэкэнде серверов, тем он надёжнее и тем выше потенциальная мощность нагрузки. За счет распределения нагрузки между несколькими серверами растёт производительность и отказоустойчивость приложения.

Так выглядит настройка двух бэкэндов (web-backend и blog-backend) с двумя серверами в каждом.

backend web-backend
balance roundrobin
server web1 web1.yourdomain.com:80 check
server web2 web2.yourdomain.com:80 check
backend blog-backend
balance roundrobin
mode http
server blog1 blog1.yourdomain.com:80 check
server blog1 blog1.yourdomain.com:80 check

Строка *balance roundrobin* определяет алгоритм балансировки нагрузки.

Строка *mode http* определяет уровень балансировки (в данном случае это уровень 7).

Опция *check* в конце директив server настраивает проверку состояния серверов **бэкэнда**.

- **Фронтэнд**

Фронтэнд определяет, каким образом запросы должны направляться на бэкэнд. Настройки **фронтэнда** определены в разделе frontend конфигурации HAProxy и состоят из следующих компонентов:

- Наборы IP-адресов и портов (например, 10.1.1.7:80, *:443);
- Списки управления доступом.
- Правила use_backend, которые определяют, какой бэкэнд использовать в зависимости от условий ACL.
- Правила default_backend которые обрабатывают случаи, не отвечающие предыдущим правилам.

Фронтэнд можно настроить для поддержки различных типов сетевого трафика, о чём и пойдёт речь в следующем разделе.

Типы балансировки нагрузки

- **Среда без балансировки нагрузки**

Среда простого приложения без балансировки нагрузки выглядит так:

Пользователь → Интернет → Веб-сервер → Сервер баз данных

То есть пользователь подключается непосредственно к веб-серверу (к вашему домену, yourdomain.com). И если этот единственный веб-сервер прекращает работу, пользователь не сможет попасть на сайт. Кроме того, если

множество пользователей одновременно попытается открыть сайт, сервер может попросту не справиться с нагрузкой: сайт будет загружаться очень медленно или же пользователь вовсе не сможет открыть его.

- **Балансировка нагрузки на уровне 4**

Самый простой способ балансировки нагрузки сетевого трафика между несколькими серверами – это балансировка на уровне 4 (также он называется транспортным уровнем).

Такая балансировка будет распределять трафик на основе диапазонов IP-адресов и портов. К примеру, если запрос поступает от http://yourdomain.com/anything, трафик будет отправлен на бэкэнд, который отвечает за поддержку yourdomain.com на порте 80.

Вот общая схема балансировки нагрузки на транспортном уровне:

Пользователь
↓
Интернет
↓
Балансировщик нагрузки
↓ ↓
Веб-сервер 1 **Веб-сервер 2**
↓ ↓
База данных

В таком случае каждый запрос сначала попадает к балансировщику, который направляет его к одному из серверов бэкэнда; этот сервер, в свою очередь, отвечает на запрос пользователя и передаёт ему результат. Как правило, все серверы бэкэнда обслуживают одинаковый контент: в противном случае пользователь может получить непоследовательное или нецелостное отображение. Обратите внимание, что оба веб-серверы подключены к одному серверу баз данных.

Вариант настройки:

```
global
  log /dev/log local0
  log /dev/log local1 notice
  chroot /var/lib/haproxy
  stats timeout 30s
  user haproxy
  group haproxy
  daemon
```

```
defaults
  log global
  mode http
  option httplog
  option dontlognull
  timeout connect 5000
  timeout client 50000
  timeout server 50000
frontend http_front
  bind *:80
  stats uri /haproxy?stats
  default_backend http_back
backend http_back
  balance roundrobin
  server server_name1 private_ip1:80 check
  server server_name2 private_ip2:80 check
```

- **Балансировка нагрузки на уровне 7**

Другой, более сложный метод настройки балансировки нагрузки – это балансировка на уровне 7 (уровень приложения). В таком случае балансировщик распределяет запросы между серверами бэкэнда на основе содержания запроса пользователя. Этот режим балансировки нагрузки позволяет запускать несколько серверов приложений на одном домене или порте.

Общая схема балансировки нагрузки на уровне приложения имеет такой вид:



В данном случае если пользователь запрашивает yourdomain.com/blog, он будет направлен на веб-сервер 2. Остальные запросы передаются на бэкэнд,

который может обслуживать другое приложение. В данном случае оба бэкэнда используют один и тот же сервер базы данных.

Настройка фронтэнда может выглядеть так:

frontend http
*bind *:80*
mode http
acl url_blog path_beg /blog
use_backend blog-backend if url_blog
default_backend web-backend

Это настроит фронтэнд http, который поддерживает входящий трафик на порте 80.

- *acl url_blog path_beg /blog соответствует запросам, которые начинаются с /blog.*
- *use_backend blog-backend if url_blog использует ACL для проксирования трафика на серверы blog-backend.*
- *default_backend web-backend обслуживает остальной трафик, который будет отправлен на web-backend.*

Алгоритмы балансировки нагрузки

Далее балансировщик выбирает из пула один из серверов согласно одному из алгоритмов. HAProxy поддерживает несколько алгоритмов балансировки.

Помимо алгоритмов балансировки нагрузки, серверы отбираются с помощью параметра weight, который определяет, как часто может использоваться тот или иной сервер по сравнению с другими серверами.

Наиболее распространёнными являются эти алгоритмы:

- Roundrobin: каждый сервер используется по очереди в соответствии со своим весом. Это самый плавный и честный алгоритм, когда время обработки серверами остается равномерно распределенным. Этот алгоритм является динамическим, что позволяет регулировать вес сервера на лету.
- Leastconn: выбирается сервер с наименьшим количеством соединений. Циклический перебор выполняется между серверами с одинаковой нагрузкой. Использование этого алгоритма рекомендуется для длинных сеансов, таких как LDAP, SQL, TSE и т. д., но он не очень подходит для коротких сеансов, таких как HTTP.

- First: первый сервер с доступными слотами для подключения получает соединение. Серверы выбираются от самого низкого числового идентификатора до самого высокого, который по умолчанию соответствует положению сервера в ферме. Как только сервер достигает значения maxconn, используется следующий сервер.
- Source: IP-адрес источника хешируется и делится на общий вес запущенных серверов, чтобы определить, какой сервер будет получать запрос. Таким образом, один и тот же IP-адрес клиента будет всегда доставаться одному и тому же серверу, в то время как серверы остаются неизменными.

Липкие сессии

Некоторым приложениям необходимо, чтобы в течение работы пользователь оставался подключенным к одному и тому же серверу бэкэнда. Алгоритм, использующий IP-адрес посетителя, может обеспечить это условие. Также привязать сессию к серверу можно с помощью так называемых липких сессий (sticky sessions): при этом балансировщик устанавливает куки, и после этого все запросы этой сессии будут направлены на один и тот же физический сервер.

Проверка состояния сервера

HAProxy должен отправлять трафик только на те активные серверы, которые в состоянии его обслужить. для этого он выполняет проверку состояния – health check, что позволяет избежать настройки бэкэнда вручную. По умолчанию проверка состояния проводится путём попытки установить TCP-соединение с сервером и, таким образом, убедиться, что он прослушивает IP-адрес и порт.

Если сервер не проходит проверку балансировщика, он автоматически удаляется из пула серверов и не будет получать трафик, пока не пройдёт проверку. Если все серверы бэкэнда не прошли проверку, сервис становится недоступным до тех пор, пока не восстановится хотя бы один сервер бэкэнда.

Другие средства балансировки

Если HAProxy – слишком сложное решение на данном этапе, вы можете выбрать более простое средство:

- Linux Virtual Servers (LVS) – простой и быстрый балансировщик нагрузки на транспортном уровне, включенный во многие дистрибутивы Linux.
- Nginx – быстрый и надёжный веб-сервер, который можно использовать в качестве обратного прокси-сервера и балансировщика нагрузки. Nginx часто используется в связке с HAProxy.

Высокая доступность

При балансировке на уровне 4 и 7 трафик направляется на один из нескольких серверов бэкэнда. Но в такой ситуации балансировщик нагрузки становится единой точкой отказа: если он выйдет из строя, приложение будет недоступно

Инфраструктуры, в которых нет единой точки отказа, обладают высокой доступностью. Это значит, что в случае отказа любого компонента инфраструктуры приложение будет работать.

Получается, что в вышеописанных примерах балансировщик нагрузки устраняет единую точку отказа с бэкэнда, но при этом сам становится этой точкой.

Чтобы «обезвредить» балансировщик как единую точку отказа, вы можете добавить в инфраструктуру второй балансировщик и создать кластер, в котором они будут проверять состояние друг друга. Каждый из них в равной степени способен обнаруживать и восстанавливать сбои.

Пользователь

↓

Интернет

↓ ↓

Балансировщик 1 Балансировщик 2

Если первый балансировщик перестанет работать, DNS передаст пользователей второму балансировщику. Изменения DNS иногда требуют много времени. Чтобы ускорить и автоматизировать переключение между балансировщиками, многие администраторы используют системы, которые поддерживают плавающие IP-адреса (или другие технологии преобразования IP). Эти технологии позволяют устранить некоторые проблемы, возникающие во время изменений DNS, предоставляя статичные IP-адреса, которые в дальнейшем при необходимости можно преобразовать. Доменное имя может быть связано с тем же IP, в то время как сам IP-адрес перемещается между серверами бэкэнда.

Балансировка нагрузки с помощью NGINX

В данной инструкции мы рассмотрим процесс настройки балансировки, в основном, http-запросов с помощью веб-сервера NGINX. По большей части, инструкция подойдет для любого дистрибутива Linux, и даже, Windows (за исключением путей расположения конфигурационных файлов). Таким образом настроенный NGINX сможет обеспечить распределение нагрузки и отказоустойчивость нашим сетевым сервисам.

Обратите внимание, что NGINX умеет распределять не только http-запросы. Его можно использовать для балансировки запросов на 4-м уровне модели OSI (TCP и UDP), например, подключения к СУБД, DNS и так далее — по сути, любой сетевой запрос может быть обработан и распределен с помощью данного программного продукта.

Постепенно рассмотрим разные варианты настройки распределения нагрузки в NGINX. Начнем с простого понимания, как работает данная функция и закончим некоторыми примерами настройки балансировки.

Основы

- Чтобы наш сервер мог распределять нагрузку, создадим группу веб-серверов, на которые будут переводиться запросы:

vi /etc/nginx/conf.d/upstreams.conf

** в данном примере мы создаем файл upstreams.conf, в котором можем хранить все наши апстримы. NGINX автоматически читает все конфигурационные файлы в каталоге conf.d.*

- Добавим:

```
upstream dmosk_backend {
   server 192.168.10.10;
   server 192.168.10.11;
   server 192.168.10.12;
}
```

** предполагается, что в нашей внутренней сети есть кластер из трех веб-серверов — 192.168.10.10, 192.168.10.11 и 192.168.10.12. Мы создали апстрим с названием dmosk_backend. Позже, мы настроим веб-сервер, чтобы он умел обращаться к данному бэкенду.*

- В настройках сайта (виртуального домена) нам необходимо теперь проксировать запросы на созданный upstream. Данная настройка будет такой:

```
server {
   ...
   location / {
      proxy_pass http://dmosk_backend;
   }
   ...
}
```

* данная настройка для нашего сайта укажет, что все запросы мы должны переводить на апстрим dmosk_backend (который, в свою очередь, будет отправлять запросы на три сервера).

- Проверяем корректность нашего конфигурационного файла:

nginx -t

- Если ошибок нет, перезапускаем сервис:

systemctl restart nginx

Приоритеты

При настройке бэкендов мы можем указать, кому наш веб-сервер будет отдавать больше предпочтение, а кому меньше.

Синтаксис при указании веса:

server <имя сервера> weight=<числовой эквивалент веса>;

По умолчанию приоритет равен 1

Также мы можем указать опции:

- backup, которая будет говорить о наличие резервного сервера, к которому будет выполняться подключение только при отсутствии связи с остальными.
- down, при указании которой, сервер будет считаться постоянно недоступным. Может оказаться полезной, чтобы остановить временно запросы для проведения обслуживания.

Давайте немного преобразуем нашу настройку upstreams:

vi /etc/nginx/conf.d/upstreams.conf

```
upstream dmosk_backend {
   server 192.168.10.10 weight=100;
   server 192.168.10.11 weight=10;
   server 192.168.10.12;
   server 192.168.10.13 backup;
}
```

Итак, мы указали нашему серверу:

- переводить на сервер 192.168.10.10 в 10 раз больше запросов, чем на 192.168.10.11 и в 100 раз больше — чем на 192.168.10.12.
- переводить на сервер 192.168.10.11 в 10 раз больше запросов, чем на 192.168.10.12.
- на сервер 192.168.10.13 запросы переводятся, только если не доступны все три сервера.

Задержки, лимиты и таймауты

По умолчанию, NGINX будет считать сервер недоступным после 1-й неудачной попытки отправить на него запрос и в течение 10 секунд не будут продолжаться попытки работы с ним. Также каждый сервер не имеет ограничений по количеству к нему подключений.

Изменить поведение лимитов и ограничений при балансировке можно с помощью опций:

- max_fails — количество неудачных попыток, после которых будем считать сервер недоступным.
- fail_timeout — время, в течение которого сервер нужно считать недоступным и не отправлять на него запросы.
- max_conns — максимальное число подключений, при превышении которого запросы на бэкенд не будут поступать. По умолчанию равно 0 (безлимитно).

Синтаксис:

server <имя сервера> max_fails=<число попыток> fail_timeout=<числовой показатель времени><едениица времени>;

В нашем примере мы преобразуем настройку так

vi /etc/nginx/conf.d/upstreams.conf

```
upstream dmosk_backend {
   server 192.168.10.10 weight=100 max_conns=1000;
   server 192.168.10.11 weight=10 max_fails=2 fail_timeout=90s;
   server 192.168.10.12 max_fails=3 fail_timeout=2m;
   server 192.168.10.13 backup;
}
```

В итоге:

- сервер 192.168.10.10 будет принимать на себя, максимум, 1000 запросов.
- сервер 192.168.10.10 будет иметь настройки по умолчанию.
- если на сервер 192.168.10.11 будет отправлено 2-е неудачные попытки отправки запроса, то в течение 90 секунд на него не будут отправлять новые запросы.
- сервер 192.168.10.12 будет недоступен в течение 2-х минут, если на него будут отправлены 3 неудачных запроса.

Метод балансировки

Рассмотрим способы балансировки, которые можно использовать в NGINX:

- Round Robin.
- Hash.
- IP Hash.
- Least Connections.
- Random.
- Least Time (только в платной версии NGINX).

Настройка метода балансировки выполняется в директиве upstream. Синтаксис:

```
upstream <название апстрима> {
   <метод балансировки>
   ...
}
```

- Round Robin

Веб-сервер будет передавать запросы бэкендам по очереди с учетом их весов. Данный метод является методом по умолчанию и его указывать в конфигурационном файле не нужно.

- Hash

Данный метод определяет контрольную сумму на основе переменных веб-сервера и ассоциирует каждый полученный результат с конкретным бэкендом. Пример настройки:

```
upstream dmosk_backend {
   hash $scheme$request_uri;
   server 192.168.10.10;
   server 192.168.10.11;
   server 192.168.10.12;
}
```

* это самый распространенный пример настройки hash — с использованием переменных $scheme (http или https) и $request_uri. При данной настройке каждый конкретный URL будет ассоциирован с конкретным сервером.

- IP Hash

Ассоциация выполняется исходя из IP-адреса клиента и только для HTTP-запросов. Таким образом, для каждого посетителя устанавливается связь с одним и тем же сервером. Это, так называемый, Sticky Session метод.

Для адресов IPv4 учитываются только первые 3 октета — это позволяет поддерживать одинаковые соединения с клиентами, чьи адреса меняются (получение динамических адресов от DHCP провайдера). Для адресов IPv6 учитывается адрес целиком.

Пример настройки:

```
upstream dmosk_backend {
   ip_hash;
   server 192.168.10.10;
   server 192.168.10.11;
   server 192.168.10.12;
}
```

- Least Connections

NGINX определяет, с каким бэкендом меньше всего соединений в данный момент и перенаправляет запрос на него (с учетом весов).

Настройка выполняется с помощью опции least_conn:

```
upstream dmosk_backend {
   least_conn;
   server 192.168.10.10;
   server 192.168.10.11;
   server 192.168.10.12;
}
```

- Random

Запросы передаются случайным образом (но с учетом весов). Дополнительно можно указать опцию two — если она задана, то NGINX сначала выберет 2 сервера случайным образом, затем на основе дополнительных параметров отдаст предпочтение одному из них. Это следующие параметры:

least_conn — исходя из числа активных подключений.

least_time=header (только в платной версии) — на основе времени ответа (расчет по заголовку).

least_time=last_byte (только в платной версии) — на основе времени ответа (расчет по полной отдаче страницы).

Пример настройки:

```
upstream dmosk_backend {
   random two least_conn;
   server 192.168.10.10;
   server 192.168.10.11;
   server 192.168.10.12;
}
```

- Least Time

Данная опция будет работать только в NGINX Plus. Балансировка выполняется исходя из времени ответа сервера. Предпочтение отдается тому, кто отвечает быстрее.

Опция для указания данного метода — least_time. Также необходимо указать, что мы считаем ответом — получение заголовка (header) или когда страница возвращается целиком (last_byte).

Пример 1:

```
upstream dmosk_backend {
   least_time header;
   server 192.168.10.10;
   server 192.168.10.11;
   server 192.168.10.12;
}
```

* в данном примере мы будем делать расчет исходя из того, как быстро мы получаем в ответ заголовки.

Пример 2:

```
upstream dmosk_backend {
   least_time last_byte;
   server 192.168.10.10;
   server 192.168.10.11;
   server 192.168.10.12;
}
```

Сценарии настройки

В реальной жизни настройки могут быть несколько сложнее, чем приведенные здесь или в официальной документации. Рассмотрим несколько примеров, что может понадобиться настроить при балансировке.

1. **Backend на https**

Предположим, что наши внутренние серверы отвечают по SSL-каналу. Таким образом, нам нужно отправлять запрос по порту 443. Также схема проксирования должна быть https.

Настройка сайта:

```
server {
   ...
   location / {
      proxy_pass https://dmosk_backend;
      proxy_set_header   Host          $host;
      proxy_set_header   X-Real-IP       $remote_addr;
      proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
   }
   ...
}
```

* обратите внимание на 2 момента:

Мы в схеме подключения proxy_pass указали https. В противном случае при подключении NGINX будет возвращать ошибку 400.

Мы задали дополнительные опции proxy_set_header, которых не было в примерах выше.

Настройка upstream:

```
upstream dmosk_backend {
   server 192.168.10.10:443;
   server 192.168.10.11:443;
   server 192.168.10.12:443;
}
```

* в данном примере мы указали конкретный порт, по которому должно выполняться соединение с бэкендом. Для упрощения конфига дополнительные опции упущены.

2. Разные бэкенды для разных страниц

Нам может понадобиться разные страницы сайта переводить на разные группы внутренних серверов.

Настройка сайта:

```
server {
   ...
   location /page1 {
     proxy_pass http://backend1;
```

```
    }

    location /page2 {
      proxy_pass http://backend2;
    }
    ...
}
```

* при такой настройке мы будем передавать запросы к странице page1 на группу backend1, а к page2 — backend2.

Настройка upstream:

```
upstream backend1 {
    server 192.168.10.10;
    server 192.168.10.11;
}

upstream backend2 {
    server 192.168.10.12;
    server 192.168.10.13;
}
```

* в данном примере у нас есть 2 апстрима, каждый со своим набором серверов.

3. На другой хост

Может быть необходимым делать обращение к внутреннему ресурсу по другому hostname, нежели чем будет обращение к внешнему. Для этого в заголовках проксирования мы должны указать опцию Host.

Настройка сайта:

```
server {
    ...
    location / {
        proxy_pass https://dmosk_backend;
        proxy_set_header   Host             internal.domain.com;
        proxy_set_header   X-Real-IP        $remote_addr;
        proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
    }
    ...
}
```

* в данном примере мы будем проксировать запросы на бэкенды, передавая им имя хоста internal.domain.com.

4. TCP-запрос

Рассмотрим, в качестве исключения, TCP-запрос на порт 5432 — подключение к базе PostgreSQL.

Настройка сайта:

```
server {
   listen 5432;
   proxy_pass tcp_postgresql;
}
```

* в данном примере мы слушаем TCP-порт 5432 и проксируем все запросы на апстрим tcp_postgresql.

Настройка upstream:

```
upstream tcp_postgresql {
   server 192.168.10.14:5432;
   server 192.168.10.15:5432;
}
```

* запросы будут случайным образом передаваться на серверы 192.168.10.14 и 192.168.10.15.

5. UDP-запрос

Рассмотрим также и возможность балансировки UDP-запросов — подключение к DNS по порту 53.

Настройка сайта:

```
server {
   listen 53 udp;
   proxy_pass udp_dns;
   proxy_responses 1;
}
```

* в данном примере мы слушаем UDP-порт 53 и проксируем все запросы на апстрим udp_dns. Опция proxy_responses говорит о том, что на один запрос нужно давать один ответ.

Настройка upstream:

```
upstream udp_dns {
   server 192.168.10.16:53;
   server 192.168.10.17:53;
}
```

* запросы будут случайным образом передаваться на серверы 192.168.10.16 и 192.168.10.17.

* в данном примере мы будем делать расчет исходя из того, как быстро мы получаем в ответ целую страницу.

5. Disaster recovery

Что такое Disaster Recovery?

Disaster Recovery (аварийное восстановление) — это сервис восстановления ИТ-систем и данных после сбоя любого уровня. Как правило, предлагается облачными провайдерами как отдельная услуга или включается в состав крупного решения. Условно можно разбить на три составляющие: резервная площадка, программные решения и план восстановления.

Disaster Recovery (DR) — катастрофоустойчивость или аварийное восстановление, инструмент реагирования на критические сбои в IT-инфраструктуре компании, направленный на быстрое восстановление работы всех систем. То есть отказ серверной или даже всего ЦОДа не должен надолго (в отдельных случаях речь о секундах) прерывать бизнес-процессы.

Реализация DR основана на управлении рисками (анализ, оценка, нужен ли вообще план восстановления, планирование). Один из вариантов — дублирующая инфраструктура, способная взять на себя всю нагрузку, если основная система откажет. В идеале ключевые узлы инфраструктуры должны работать по принципу георезервирования:

- Георезерв инфраструктуры — запасную инфраструктуру разворачивают в другой геолокации, не подверженной или менее подверженной аналогичным природным или техногенным ЧС. Да, от взрыва Солнца даже ЦОД на Марсе не спасёт, но уберечься от наводнения, развернув дублирующую инфраструктуру в разных городах, получится;
- Георезерв электросети — зачастую используют другие магистральные каналы. Например, выходит из строя местная электростанция, основной ЦОД переходит на дизельные генераторы, а запасной ЦОД продолжит питаться от ГЭС. Так как второй вариант дешевле, можно остановить работу первого ЦОДа, пока аварийные службы восстанавливают электропитание. Такой подход также используют и в рамках одного ЦОДа;
- Георезерв интернет-каналов — другие магистральные интернет-провайдеры, каналы которых не проходят в одном месте. Если экскаватор пробьет один из каналов, когда будет копать рядом с ЦОДом, это не остановит работу.

Без чего не обойтись: подготовка и DRP

Всем без исключения нужен план аварийного восстановления DRP (Disaster Recovery Plan). Его составлением нужно заниматься комплексно: со

стороны бизнеса и со стороны IT-инфраструктуры и специалистов. Даже малому бизнесу с одним сервером неплохо бы иметь план на салфетке, что делать, когда тот выйдет из строя, чтобы минимизировать финансовые потери.

Перед тем как вы приступите к составлению DRP, нужно провести анализ воздействия на бизнес, оценить риски и провести ряд других превентивных мер. В конце должен получиться документ, который согласован руководством и IT-службами. В этом документе будет прописан четкий пошаговый план и регламент действий по скорейшему восстановлению инфраструктуры после аварии.

Пункты, которые нужно проработать при составлении DRP:

- Состав штатной DR-команды и уровни доступа у сотрудников;
- Предварительная оценка рисков;
- Определение критически важных бизнес-процессов и данных;
- Бюджет;
- Необходимость в подрядчиках и поставщиках (услуг в том числе), а также их надежность;
- Используемые технологии;
- Необходимые уровни доступности;
- Нужен ли SLA (Service-Level Agreement), подробнее о том, что это такое и с чем едят — ниже;
- Быстрое восстановление или параллельная инфраструктура;
- В каком формате будет составлен DRP. Вы можете продумать свой документ с нуля или воспользоваться готовым шаблоном американской компании TechTarget (или любым другим).

Разберём некоторые пункты поподробнее.

1. Состав штатной DR-команды

Во-первых, кто-то должен спланировать, утвердить, реализовать и обслуживать инфраструктуру DR. Во-вторых, должен быть регламент конкретных действий каждого сотрудника, чтобы как можно быстрее начать процедуру аварийного восстановления. Полностью автоматизировать весь процесс невозможно, поэтому нужно выделить и/или нанять сотрудников под эти задачи. Если компания и инфраструктура небольшая, то хватит одного человека, если речь о дата-центрах Amazon, то нужна целая команда или департамент.

При этом нужно чётко определить необходимые уровни доступа для каждого специалиста, чтобы не случалось сюрпризов, когда у члена DR-команды

нет доступа к северу или ЦОДу. Или, наоборот, когда новый член команды будет иметь слишком много прав.

2. Предварительная оценка рисков

Оценку рисков можно провести самостоятельно — силами штатных сотрудников, исходя из собственного опыта. Или можно отдать эту задачу на аутсорс. Методик много, универсальной модели не существует, но есть общие фундаментальные принципы, от которых нужно отталкиваться. Так как всё охватить не получится, я буду говорить о первостепенных рисках, способных повлиять на IT-инфраструктуру.

Внешние риски

Этот вид рисков требует особого внимания и анализа, так как включает в себя политическую, экономическую, экологическую и другие повестки. Всё — от наводнения до тестирования автономного рунета и санкций — может повлиять на ваши бизнес-процессы.

Санкции против IT-сектора

После событий в 2014 году наметился явный курс на импортозамещение. К сожалению, IT-системы на западном ПО и железе сложно заменить — иногда получается хорошо, но чаще всего — не очень. Если завтра Россию отключат от исходников GitHub, то у разработчиков возникнут большие проблемы. Западные организации могут в одно мгновение удалить репозитории российских компаний без предоставления бэкапов, и многолетний труд безвозвратно исчезнет. Вот вам и открытый исходный код.

И не забывайте про привычные вам облачные сервисы: Gmail, Slack, Jira, Discord, YouGile, Google Docs и многие другие. Даже если их не заблокируют, то могут возникнуть проблемы с оплатой, продлением и т.д. Молчу про окирпиченное сетевое оборудование и удаление учётных записей Microsoft. Список почти бесконечный.

Поэтому критиковать импортозамещение также легко, как и вернуться на уровень развития девяностых-нулевых.

Другие риски

Обстоятельствам непреодолимой силы не получится противостоять, но вы сможете подготовиться к ним. К счастью, некоторые риски уже проанализированы и оценены. Нужно только правильно воспользоваться общедоступной информацией.

Немного поискав информацию, вы сможете оценить, в каком районе вашего города лучше размещать ЦОД или серверную, на какой высоте от уровня моря это сделать безопаснее и т.д. Не все факторы риска поддаются такой же

точной оценке, но не пренебрегайте тем, что доступно бесплатно и поможет уберечь оборудование.

Да, не у всех компаний есть возможность выбрать желаемое место, поэтому оценка внешних рисков позволит понять, какие меры нужно предпринять для вашей точки, понадобятся ли резервные серверы DR или нет.

Вам может показаться, что риски, такие как наводнение, явление крайне редкое. Тогда посмотрите, как в сентябре 2009 года за несколько минут затопило офис и ЦОД Vodafone.

Disaster Recovery Plan не защитит серверное оборудование от потопа или пожара, но позволит спрогнозировать форс-мажор и быстро вернуться к работе бизнес-процессов.

Внутренние риски

С определением внутренних рисков ситуация обстоит одновременно проще и сложнее, так как они напрямую связаны с деятельностью компании. С одной стороны — ими легче управлять и легче предвидеть, а с другой — их может быть значительно больше, а чтобы провести точный анализ, нужно много данных и времени. Например:

- Риски человеческого фактора

Невозможно на 100% процентов быть уверенным в ком-то. Человек может недобросовестно работать, устать, допустить ошибку, обидеться, внезапно уволиться, заболеть, взять отгул по семейным обстоятельствам, переоценить/недооценить свои возможности и т.д. Поэтому человеческий фактор всегда стоит закладывать в оценку рисков. Это не значит, что у вас будут точные цифры. Скорее вы будете готовы к тому, что даже бульдозер могут перевернуть на ровном месте, а значит было бы неплохо иметь второй.

- Технические риски

Если составить технические регламенты для сотрудников и строго следовать им, то можно минимизировать риск ошибок. Однако всё равно остаётся вероятность, что ошибку допустят на этапе написания инструкции, или сотрудник, читающий её, что-то не так поймёт. Это может быть опасно как для оборудования, так и для сотрудников. Поэтому нужно со вниманием отнестись к соблюдению регламентов, инструкций, техники безопасности и т.д., а также учесть, что и они могут быть не идеальны и требуют доработок со временем.

- Технологические риски

Технологические риски связаны с типом используемого оборудования, ПО, интенсивностью нагрузок т.д. Стоит учесть, что продукция крупных вендоров-пионеров заслуживает большего доверия: она проверена временем, оснащена проприетарными разработками, да и технологии тестирования и производства более совершенны, так как на отладку всех процессов ушли десятилетия. У серверов от Dell, HPE и Lenovo выше показатели надежности MTBF (Mean Time Between Failures) — это время, в течение которого не ожидается сбоев.

Как оценить внутренние риски?

Вряд ли вам удастся с точностью до процента определить, какова вероятность возникновения человеческого фактора; насколько сервер Dell или HPE будет отличаться надёжностью от серверов Atos, NEC или Supermicro; как часто выходит из строя оборудование, обслуживаемое по регламенту и без. Разве что вы целенаправленно не собирали статистику лет 5-10 по всем направлениям.

В противном случае вы можете поискать общедоступную информацию в смежных или аналогичных областях, а все точки отказа, которые кажутся наиболее уязвимыми, защитить.

3. Определите критически важные бизнес-процессы и уровень доступности для них

Чтобы IT-инфраструктура не стоила чудовищных денег, нужно расставить приоритеты. Если взять критерий непрерывной работы, то бизнес-процессам можно присвоить уровни важности: высокий, средний или низкий:

- Высокий. Даже несколько минут простоя могут привести к большим и(или) необратимым потерям и негативным последствиям. Бизнес-процессы должны вернуться к работе как можно быстрее — от нескольких секунд, до нескольких минут

Примеры компаний: IT-гиганты, автоматизированные заводы, социальные сети, банки, биржевые и другие финансовые учреждения, многие государственные структуры.

Примеры бизнес-процессов компании: банковские транзакции, производство процессоров, работа СКУД в охраняемой зоне.

- Средний. Бизнесу требуется быстрое восстановление, но он может пережить простой от нескольких минут до часа без существенных потерь.

Примеры компаний: розничные магазины, производитель мебели, печатное издательство.

Примеры бизнес-процессов компании: планирование продаж в соответствующих программах, создание новой линейки мебели в CAD-программе, верстка ежемесячного выпуска журнала.

- Низкий. Это могут быть любые компании, которым больше важна сохранность данных, чем их непрерывная доступность. Простой от двух до нескольких часов не станет большой проблемой.

Примеры компаний: малый бизнес, небольшие компании и предприятия.

Примеры бизнес-процессов внутри компании: бухгалтерский учёт, составление рабочего расписания, архив данных, обновление базы 1С.

Когда вы закончите с приоритетностью бизнес-процессов, можно приступать к определению уровня доступности системы:

- Постоянная доступность (continuous availability). Если речь, например, об электронном движении капитала, то инфраструктура должна быть готова к запланированным и незапланированным простоям, а также непрерывно работать 24/7 годами.

Для обеспечения постоянной доступности IT-системам присваивают статус Mission Critical. Инфраструктура этого уровня всесторонне резервируется и георасределяется, а также работает в режиме постоянного дежурства и должна полностью восстановиться в течение минут после аварии.

- Непрерывный режим работы (continuous operations). Чуть менее доступная инфраструктура, не готовая к авариям на всех узлах, но по-прежнему работающая 24/7 годами, если не возникает сбоев.

Для обеспечения непрерывного режима работы IT-системам присваивают статус Business Critical. Инфраструктура этого уровня частично резервируется и георасределяется на важных точках отказа, а также работает в режиме непрерывной готовности и должна полностью восстановиться в течение 1-2 часов после аварии.

- Высокая доступность (high availability). Режим работы инфраструктуры, при котором система периодически останавливается в нерабочие дни или на запланированное обслуживание с предварительным уведомлением пользователей. Режим работы определяется работой организации: 8/5, 8/7, 12/5, 12/7 и т.д.

Для обеспечения высокой доступности IT-системам присваивают статус Business Operational. Инфраструктура этого уровня частично резервируется (сети,

данные, электропитание), а также работает в режиме повышенной готовности и должна полностью восстановиться в течение 4-6 часов после аварии.

- Обычная доступность.

Если высокой доступности и(или) сохранности данных не требуется, IT-системам присваивают статус Office Production. Инфраструктура этого уровня может не резервироваться или резервироваться на самых простых уровнях под конкретные задачи. Время восстановления от одного до нескольких дней.

Важно! Высокая доступность не равноценна DR, так как включает в себя отказоустойчивость отдельных (или всех) узлов IT-инфраструктуры, но не рассчитана на катастрофы. То есть она готова к отказу магистральной электросети, серверной стойки или интернет-сети, но не к случаю пожара или взрыва.

4. Заключите SLA — соглашение об уровне обслуживания

SLA (Service-Level Agreement) — это единый документ, договор между поставщиком и клиентом, в котором закрепляются конкретные услуги и их объем, ожидания сторон, их обязанности, точки контактов для решения проблем, метрики эффективности, а также время исправления инцидента *(Время реакции на аварию + Время решения проблемы = Время исправления инцидента)*.

Метрики SLA (цифры, которым должен соответствовать бизнес-процесс).

Доступность	Время простоя за год (максимум)
99 %	3.65 дней
99,9 %	8.76 часов
99,95 %	4.38 часов
99,99 %	52.56 минут
99,999 %	5.26 минут

В третьем пункте говорилось о том, как определиться с необходимым уровнем доступности бизнес-процессов, часть из которых зависит от партнёров и подрядчиков, что привносит фактор неопределённости в DRP. И здесь есть две стороны:

1. Ваш внутренний уровень SLA, который выставляется (как требование) сотрудникам и внутренним бизнес-процессам.

2. Уровень SLA, который требуется от контрагентов.

Если требования к уровню доступности вашего бизнес-процесса составляют 99.9%, то ваш контрагент, связанный с этим бизнес-процессом, не может работать на уровне 99%, так как станет слабым звеном. Поэтому ищут либо резервного подрядчика (на случай аварии), либо подрядчика с более высоким уровнем SLA.

Когда речь идёт о срочном аварийном восстановлении IT-инфраструктуры, все системы должны работать, как атомные часы, так как потери из-за увеличенного простоя в некоторых сценариях будут колоссальными.

Поэтому заключение SLA — это важное условие доступности системы. А если SLA не будет соблюдаться, то стороны смогут возместить убытки, защитить себя в спорах, но не смогут сослаться на незнание или на некоторые внешние факторы при возникновении нештатных ситуаций.\

5. Определите RPO и RTO

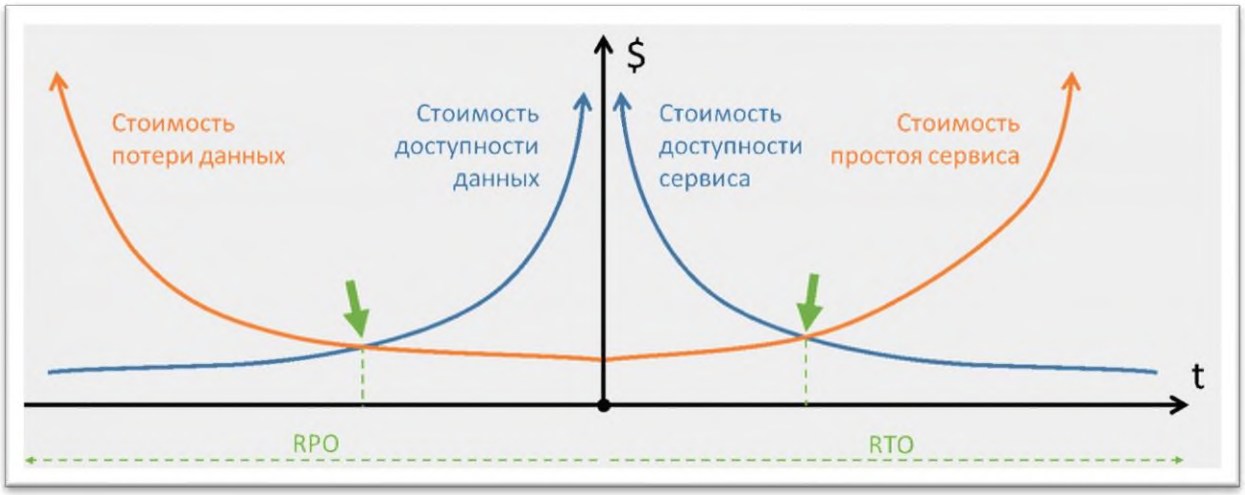


Рисунок 2. RPO и RTO

Важными параметрами аварийного восстановления являются RPO (Recovery Point Objective) и RTO (Recovery Time Objective). Требования по ним идут от бизнеса — директоров, руководителей и менеджеров, а IT-команда пытается максимально приблизиться к требуемым показателям, учитывая бюджет и реальное положение дел.

RPO — это целевая точка восстановления, которая устанавливается после определения точки безубыточности: затраты на сохранность актуальных данных не должны превышать ущерба от их потери. Если RPO равен 1-ой минуте, значит резервная копия будет обновляться каждую минуту. При этом копия будет только одна.

RTO — это целевое время восстановления. Устанавливается, чтобы понимать, за какой отрезок времени бизнес-процесс должен вернуться к

полноценной работе. Если RTO установили в полчаса, значит через 30 минут система должна работать в штатном режиме — ни минутой позже.

Как это выглядит на практике. IT-команда запрашивает у бизнеса требуемые показатели RPO\RTO, получает ответ. Далее специалисты считают бюджет, после чего показатели корректируются до приемлемых.

В конце проводятся согласованные с бизнесом учения (репетиции аварийных ситуаций и восстановлений), чтобы убедиться, что реальные показатели не отличаются от требуемых. Если показатели не сходятся, составляется дальнейший план действий.

6. Выбор конкретного решения DR

Все предыдущие оценки позволят вам определиться, какой уровень DR подойдет под конкретный бизнес-процесс и бюджет.

- Бэкапы и быстрое восстановление из облака (DRaaS Backup & Restore).

Это компромиссный и относительно недорогой вариант, поскольку не приходится покупать, настраивать и обслуживать дублирующую инфраструктуру; стоимость такого решения минимальна, а платить нужно только за объем данных в облаке и за временное использование виртуальных машин в случае ЧС. Быстрое восстановление допускает непродолжительную остановку бизнес-процессов.

Чаще всего малый и средний бизнес делает так: бэкапируют систему в облако (получается георезервирование минимальными усилиями), в котором можно быстро восстановиться. Если нагрянет локальный апокалипсис, то запустятся (автоматически или по команде) виртуальные машины, которые возьмут на себя текущие нагрузки. Следом восстанавливают основную IT-систему и переносят нагрузки на неё; виртуалки закрывают, а в облако копируют последующие бэкапы на случай новой аварии. Таким образом показатели RTO/RPO достигают нескольких часов.

Существуют готовые решения DR в облаке:

- Veeam Disaster Recovery Orchestrator;
- Mail.ru Cloud DRaaS;
- CloudMTS Disaster Recovery;
- Множество других DRaaS сервисов.

- Параллельная инфраструктура (Multi-Site Solution)

Название говорит само за себя — параллельно основной будет подготовлена резервная инфраструктура с репликой системы. Преднастроенные WAN-каналы, сеть и виртуальные машины со всем необходимым софтом будут ожидать своего часа.

Аналогично первому пункту можно организовать всё в облаке, но тогда стоит посчитать: в какие затраты это выльется за несколько лет. Возможно, в перспективе 5-10 лет на аренду вы потратите соизмеримую с покупкой и обслуживанием сумму, но владеть ничем не будете. Параллельная инфраструктура позволяет восстановиться очень быстро — от пары секунд до нескольких минут.

Существует две основных стратегии репликации:

- Пассивная репликация:

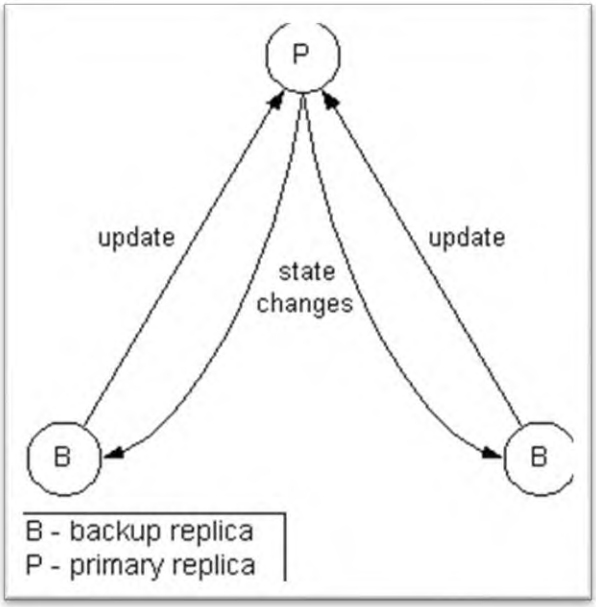


Рисунок 3. Пассивная репликация

Операции модификации сначала происходят в первичной реплике, а следом инструкции отправляются на параллельную инфраструктуру, где происходят локальные расчёты и изменения. При отказе первичной реплики включается пассивная с небольшой задержкой.

- Активная репликация:

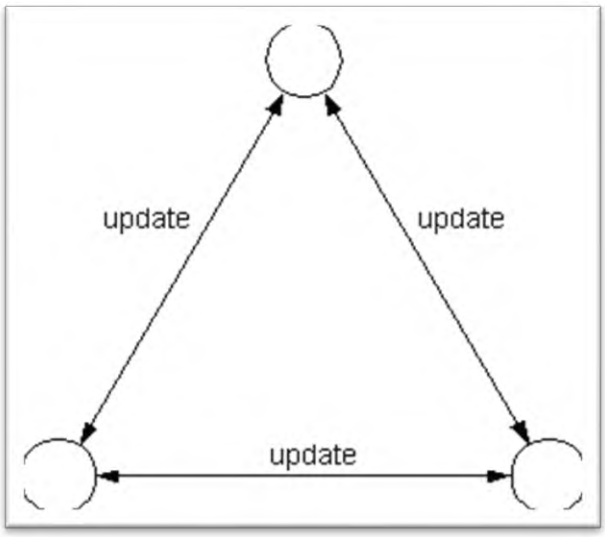


Рисунок 4. Активная репликация

Операции модификации одновременно отправляются на основную и параллельную инфраструктуры, где происходят локальные расчёты и изменения. При отказе первичной реплики одна из активных берет на себя нагрузку в тот же момент.

Даже реализуя параллельную инфраструктуру, есть несколько вариантов хранения, копирования и резервирования данных:

- Основная и параллельная системы имеют общую отказоустойчивую СХД, а потому данные всегда поддерживаются в актуальном состоянии. Надёжность, как вы понимаете, зависит от отказоустойчивости СХД;
- Данные хранятся в разных хранилищах, но сначала записываются в рабочую систему, а потом копируются из основной в резервную. В случае аварии возможна потеря копируемых данных;
- Данные хранятся в разных хранилищах и параллельно записываются в них. Таким образом получается достигнуть полной сохранности информации и максимальной надежности.

7. Проверка работоспособности: учения и тесты failover

Будет обидно, если вы заранее побеспокоитесь об аварийном восстановлении, но система не запустится в критической ситуации, так как никто не проводил тесты. И важно учесть, что единоразовым начальным тестированием тут не отделаешься. Сформированная ответственная группа должна постоянно проводить учения и поддерживать актуальное состояние DR-системы, чтобы соответствовать меняющимся внешним и внутренним рискам.

Учения чаще всего проводят отдельно для каждого риска. Например, DR-команда отключает сетевой кабель и тестирует падение интернет-провайдера.

Другой сценарий — оценивают влияние человеческого фактора, когда сисадмин пропал (не вышел на работу, не вернулся с обеда). Аналогично проводят и учения по ИБ (информационной безопасности), имитируя различные кибератаки и вмешательства.

Такой подход называется Chaos Engineering (хаос-инженерия).

Важнейшим инструментом Disaster Recovery является работающее аварийное переключение или failover. Чтобы процесс запускался автоматически и максимально быстро, основная и резервная системы должны постоянно обмениваться так называемым “пульсом” — специальным периодическим сигналом. Как только резервная система не получает сообщений от основной в течение заданного времени, запускается протокол failover.

Канал, по которому передаётся “пульс” должен быть стабильным, надёжным и дублироваться, чтобы избежать ложных переключений. Когда тесты/восстановление закончатся, основная система должна вернуться к штатному режиму работы. Таким образом запускается протокол обратный от failover, называемый failback.

Почему вам нужен DRP, а одних бекапов может не хватить?

Некоторые организации ограничиваются бэкапами в вопросах отказоустойчивости. Даже надёжная схема бэкапирования 3-2-1 — не панацея, хоть и может выступать единственной мерой DRP при ограниченном бюджете.

Если проводить аналогию, то бэкапы — это подушки безопасности в автомобиле. До того, как они сработают, есть ещё несколько линий обороны для пассажиров: реакция водителя, тормозная система, особая конструкция кузова, специальный вывод из строя некоторых систем (рулевой колодки), ремни безопасности, подголовники и т.д.

Бэкапы, как и подушки безопасности — последний рубеж. А план аварийного восстановления должен предусмотреть, какие системы будут принимать на себя удар до восстановления систем из бэкапов. В ЦОДах реализуется резервирование: отказоустойчивые кластеры, независимые системы охлаждения, несколько БП и источников питания (ИБП, дизельные генераторы). Все меры направлены, чтобы увеличить метрики SLA, о которых мы говорили выше.

Disaster Recovery Plan разрабатывается, чтобы при катастрофах максимально снизить уровень недоступности системы или её части, чтобы ни цепочка сбоев, ни внезапная антропогенная катастрофа не остановили важные бизнес-процессы. Да, в отдельных случаях это требует значительных

финансовых вложений, но потому мы и обсуждали оценку рисков — затраты не должны превышать возможных потерь.

Рассмотрим в таблице, какие преимущества даёт полноценная катастрофоустойчивая инфраструктура.

Таблица 1. Отличия DR от бэкапов

Параметр	**DR**	**Бэкапы**
RPO	Чем чаще, тем лучше. Настраивают исходя из задач и бюджета. В особо критичных системах резервирование происходит в реальном времени. **Результат:** Одна актуальная резервная копия. Выход из строя системы не приводит к потере данных.	Настройки можно установить аналогичные DR, но обычно бэкапы не делаются слишком часто из-за технических ограничений. **Результат:** Копий обычно несколько. Можно выбрать, до какой версии откатиться, но кратковременные данные будут утеряны, что недопустимо, например, в банковской сфере.
RTO	Целевое время восстановления — от секунд до нескольких часов на запасной инфраструктуре. Зависит от критичности бизнес-процессов. **Результат:** Серьезные происшествия, например пожар, не остановят бизнес-процессы больше чем на секунды или часы.	Целевое время восстановления — от часов до нескольких дней, так как придётся восстанавливать данные на отказавшей инфраструктуре. А это может потребовать замены комплектующих или переезда. **Результат:** Если система полностью выйдет из строя, например, из-за пожара, то восстановление будет длиться больше нескольких дней.
Отказоустойчивость	Георезервирование позволяет добиться полной катастрофоустойчивости.	Бэкапы сами по себе не обеспечивают отказоустойчивости, но

		позволяют восстановиться до последней резервной копии после аварии.
Инфраструктура	Для реализации нужна аналогичная инфраструктура или её часть, если DRP учитывает только некоторые узлы системы.	Для реализации достаточно СХД, дополнительных жёстких дисков или облачного хранилища.
Стоимость	Дорогостоящее решение.	Стоимость зависит от глубины хранения, частоты и объема резервируемых данных, а также типа накопителей или облака.

6. Keepalived/vrrp

Keepalived – сервисное программное обеспечение, которое гарантирует высокую доступность кластера при управлении кластером Linux, его функция аналогична «пульсу» и используется для предотвращения единичных точек отказа.

Как работает keepalived?

Keepalived основан на протоколе VRRP. VRRP означает протокол резервирования виртуального маршрутизатора.

Протокол избыточности виртуальной маршрутизации может рассматриваться как протокол для достижения высокой доступности маршрутизаторов. То есть N маршрутизаторов, которые предоставляют одну и ту же функцию, образуют группу маршрутизаторов. Эта группа имеет главное резервное копирование и несколько резервных копий. Основное устройство имеет vip (предоставляющий внешние услуги). Маршрут по умолчанию для других машин в локальной сети, где расположен маршрутизатор, — это vip. Мастер будет отправлять многоадресную передачу. Если резервная копия не может получить пакет vrrp, считается, что мастер не работает.

Keepalived в основном имеет три модуля, а именно core, check и vrrp. Основной модуль является ядром keepalived и отвечает за запуск и обслуживание основного процесса, а также за загрузку и анализ глобальных файлов конфигурации. check отвечает за проверки состояния, включая различные распространенные методы проверки. Модуль vrrp предназначен для реализации протокола VRRP.

Keepalived имеет только один файл конфигурации keepalived.conf, который в основном включает следующие области конфигурации, а именно: global_defs, static_ipaddress, static_routes, vrrp_script, vrrp_instance и virtual_server.

- **область global_defs** – это, главным образом, объект уведомления и идентификация машины при сбое конфигурации

```
global_defs {
   notification_email {
     a@abc.com
     b@abc.com
     ...
   }
   notification_email_from alert@abc.com
```

smtp_server smtp.abc.com

smtp_connect_timeout 30

enable_traps

router_id host163

}

- *messages_email. Кому отправлять уведомления по электронной почте при возникновении ошибки.*
- *tification_email_from – адрес, с которого отправляются уведомления по электронной почте.*
- *smpt_server SMTP – адрес уведомления по электронной почте.*
- *smtp_connect_timeout – время ожидания для подключения к серверу SMTP*
- *enable_trap – включение SNMP-ловушки (Простой протокол управления сетью).*
- *router_id – строка, идентифицирующая узел, обычно имя хоста. Уведомление по электронной почте будет использоваться при возникновении ошибки.*
- **области static_ipaddress и static_routes**

В областях static_ipaddress и static_routes указываются IP-адрес и информация о маршрутизации этого узла. Если IP и маршрутизация уже настроены на вашем компьютере, то эти две области могут быть пропущены. Фактически, в нормальных условиях ваш компьютер будет иметь IP-адрес и информацию о маршрутизации, поэтому нет необходимости настраивать эти две области.

static_ipaddress {

10.210.214.163/24 brd 10.210.214.255 dev eth0

...}

static_routes {

10.0.0.0/8 via 10.210.214.1 dev eth0

...}

Следующие команды выполняются локально при запуске / завершении keepalived:

 # /sbin/ip addr add 10.210.214.163/24 brd 10.210.214.255 dev eth0# /sbin/ip route add 10.0.0.0/8 via 10.210.214.1 dev eth0# /sbin/ip addr del 10.210.214.163/24 brd 10.210.214.255 dev eth0# /sbin/ip route del 10.0.0.0/8 via 10.210.214.1 dev eth0

- область vrrp_script

используется для проверки работоспособности. При неудачной проверке приоритет vrrp_instance будет уменьшен на соответствующее значение.

vrrp_script chk_http_port {
script "</dev/tcp/127.0.0.1/80"
interval 1
weight -10}

Вышеуказанное означает, что, если выполнение инструкции в сценарии завершится неудачно, соответствующий приоритет vrrp_instance будет уменьшен на 10 пунктов.

- **области vrrp_instance и vrrp_sync_group**

vrrp_instance используется для определения VIP-зоны и связанных атрибутов, которые предоставляют услуги внешним сторонам.

vrrp_rsync_group используется для определения группы vrrp_intance, так что члены этой группы действуют сообща.

Пример, иллюстрирующий его функцию:

Два vrrp_instance принадлежат одной и той же vrrp_rsync_group, затем, когда один из vrrp_instance переключается на другой ресурс, другой vrrp_instance будет следовать за переключением.

```
vrrp_sync_group VG_1 {
   group{
      inside_network   # name of vrrp_instance (below)
      outside_network  # One for each moveable IP.
      ...
   }
   notify_master /path/to_master.sh
   notify_backup /path/to_backup.sh
   notify_fault "/path/fault.sh VG_1"
   notify /path/notify.sh
   smtp_alert
}
vrrp_instance VI_1 {
   state MASTER
   interface eth0
   use_vmac <VMAC_INTERFACE>
   dont_track_primary
   track_interface {
      eth0
```

```
    eth1
  }
  mcast_src_ip <IPADDR>
  lvs_sync_daemon_interface eth1
  garp_master_delay 10
  virtual_router_id 1
  priority 100
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass 12345678
  }
  virtual_ipaddress {
    10.210.214.253/24 brd 10.210.214.255 dev eth0
    192.168.1.11/24 brd 192.168.1.255 dev eth1
  }
  virtual_routes {
    172.16.0.0/12 via 10.210.214.1
    192.168.1.0/24 via 192.168.1.1 dev eth1
    default via 202.102.152.1
  }
  track_script {
    chk_http_port
  }
  nopreempt
  preempt_delay 300
  debug
  notify_master <STRING>|<QUOTED-STRING>
  notify_backup <STRING>|<QUOTED-STRING>
  notify_fault <STRING>|<QUOTED-STRING>
  notify <STRING>|<QUOTED-STRING>
  smtp_alert
}
```

- notify_master / backup / fault соответственно представляет скрипт, выполняемый при переключении на master / standby / error.
- notify указывает, что сценарий будет вызываться при переключении любого состояния, и сценарий будет вызываться после завершения выполнения

трех указанных выше сценариев; keepalived автоматически передаст три параметра ($ 1 = "GROUP" | "INSTANCE", $ 2 = имя группы или экземпляр, $ 3 = целевое состояние перехода (MASTER / BACKUP / FAULT)).

- smtp_alert указывает, нужно ли включать уведомление по электронной почте (отправлять уведомления с настройками почты глобальной зоны).
- Состояние может быть MASTER или BACKUP, но, когда другие узлы продолжают активироваться, узел с более высоким приоритетом будет выбран в качестве MASTER, поэтому этот пункт не имеет практической цели.
- Интерфейс Сетевая карта с IP-адресом узла (не VIP), который используется для отправки пакетов VRRP.
- use_vmac Использовать ли виртуальный MAC-адрес VRRP.

- dont_track_primary игнорирует ошибки сетевой карты VRRP. (Не установлено по умолчанию)
- track_interface контролирует следующие сетевые карты, если любая из них дает сбой, он переключается в состояние FALT. (Необязательный)
- mcast_src_ip изменяет адрес источника многоадресных пакетов vrrp, адрес источника по умолчанию - IP-адрес мастера. (Поскольку это многоадресная передача, даже если адрес источника изменен, мастер все равно может получить ответ)
- lvs_sync_daemon_interface Привязать сетевую карту lvs syncd.
- garp_master_delay Как часто обновляется кэш ARP после перехода в основное состояние, по умолчанию это 5 секунд.
- Значение virtual_router_id находится в диапазоне 0-255 и используется для различения нескольких экземпляров многоадресной рассылки VRRP.

Примечание. Значение virtual_router_id в том же сегменте сети не может быть повторено, в противном случае произойдет ошибка, и соответствующая информация об ошибке будет следующей:

Keepalived_vrrp[27120]: ip address associated with VRID not present in received packet :

one or more VIP associated with VRID mismatch actual MASTER advert

bogus VRRP packet received on eth1 !!!

receive an invalid ip number count associated with VRID!

VRRP_Instance(xxx) ignoring received advertisment...

Вы можете использовать эту команду для просмотра существующего vrid в сети: tcpdump -nn -i any net 224.0.0.0/8

- приоритет используется для выбора мастера. Чтобы стать мастером, значение этой опции лучше, чем 50 точек других машин. Диапазон значений этого элемента составляет 1–255 (вне этого диапазона он будет распознаваться как значение по умолчанию 100).
- advert_int Интервал между отправкой пакетов VRRP, то есть как часто нужно проводить мастер-выборы (его можно рассматривать как интервал проверки работоспособности).
- область аутентификации аутентификации, типы аутентификации PASS и HA (IPSEC), рекомендуется использовать PASS (пароль распознает только первые 8 цифр).
- virtual_ipaddress vip, без объяснения причин.
- virtual_routes Виртуальная маршрутизация, информация о маршрутизации, которая должна быть добавлена после смещения IP.
- virtual_ipaddress_excluded IP-адрес, не включенный в отправленный пакет VRRP, чтобы уменьшить количество отвечающих пакетов VRRP. Используется, когда к сетевой карте привязано больше IP-адресов.
- nopreempt позволяет узлу с более низким приоритетом быть ведущим, даже если запущен узел с более высоким приоритетом.

- **область virtual_server_group и virtual_server**

virtual_server_group обычно используется в очень больших LVS, что случается крайне редко.

virtual_server IP Port{
delay_loop <INT>
lb_algo rr|wrr|lc|wlc|lblc|sh|dh
lb_kind NAT|DR|TUN
persistence_timeout <INT>
persistence_granularity <NETMASK>
protocol TCP
ha_suspend
virtualhost <STRING>
alpha
omega
quorum <INT>

```
hysteresis <INT>
quorum_up <STRING>|<QUOTED-STRING>
quorum_down <STRING>|<QUOTED-STRING>
sorry_server <IPADDR><PORT>
real_server <IPADDR><PORT>{
  weight <INT>
  inhibit_on_failure
  notify_up <STRING>|<QUOTED-STRING>
  notify_down <STRING>|<QUOTED-STRING>
  # HTTP_GET|SSL_GET|TCP_CHECK|SMTP_CHECK|MISC_CHECK
  HTTP_GET|SSL_GET {
    url {
      path <STRING>
      # Digest computed with genhash
      digest <STRING>
      status_code <INT>
    }
    connect_port <PORT>
    connect_timeout <INT>
    nb_get_retry <INT>
    delay_before_retry <INT>
  }
}}
```

- delay_loop – время опроса задержки (в секундах).
- Алгоритм отладки бэкэнда lb_algo (алгоритм балансировки нагрузки).
- lb_kind – тип планирования LVS NAT / DR / TUN.
- virtualhost используется для настройки заголовков запросов для HTTP_GET и SSL_GET.
- sorry_server. Когда все реальные серверы выйдут из строя, сервер извинения вступит во владение.
- real_server – сервер, который действительно предоставляет услуги.
- notify_up / down – сценарий, выполняемый, когда реальный сервер выключается или запускается.
- path – запросить путь на реальном сервере.

- digest / status_code – указывает результат, рассчитанный с помощью genhash и http status code соответственно.
- проверка работоспособности connect_port, если порт открыт, сервер считается нормальным.
- connect_timeout, nb_get_retry, delay_before_retry соответственно указывают продолжительность тайм-аута, количество повторных попыток и временную задержку для следующей повторной попытки.

1. Установка keepalived

Загрузите соответствующую версию с официального сайта keepalived, разархивируйте и выполните следующую команду для завершения установки.

 # cd keepalived-xxx# ./configure --bindir=/usr/bin --sbindir=/usr/sbin --sysconfdir=/etc --mandir=/usr/share# make && make install

Вы также можете упаковать его в пакет RPM и установить его.

Выполните следующие шаги: загрузите keepalived-1.1.15.tar.gz, а затем распакуйте его и установите.

#tar zxvf keepalived-1.1.15.tar.gz
#cd keepalived-1.1.15
#./configure
#make
#make install

2. Настройка keepalived

Состояние MASTER в конфигурации определяет, что узел является главным.

Конфигурация главного узла, следующая:

```
global_defs {
   router_id NodeA
}
vrrp_instance VI_1 {
    укажите MASTER #Set в качестве главного сервера
        интерфейс eth0 # Сетевой интерфейс монитора
        virtual_router_id 51 # Мастер и раб должны быть одинаковыми
        приоритет 100 # (основной и резервный компьютеры имеют разные приоритеты, значение хоста больше, значение резервного компьютера меньше, чем больше значение, тем выше приоритет)
        advert_int 1 #VRRP Цикл многоадресной передачи секунд
    authentication {
```

auth_type PASS #VRRP метод аутентификации, ведущий и ведомый должны быть согласованы

auth_pass 1111 # (пароль)

}

virtual_ipaddress {

192.168.8.100/24 #VRRP HA виртуальный адрес

Конфигурация резервного узла выглядит следующим образом:

global_defs {

router_id NodeB

}

vrrp_instance VI_1 {

состояние BACKUP #Set as главный сервер

интерфейс eth0 # Сетевой интерфейс монитора

virtual_router_id 51 # Мастер и раб должны быть одинаковыми

приоритет 90 # (основной и резервный компьютеры имеют разные приоритеты, значение хоста больше, значение резервного компьютера меньше, чем больше значение, тем выше приоритет)

advert_int 1 #VRRP Цикл многоадресной передачи секунд

authentication {

auth_type PASS #VRRP метод аутентификации, мастер и резервная копия должны быть согласованы

auth_pass 1111 # (пароль)

}

virtual_ipaddress {

192.168.8.100/24 #VRRP HA виртуальный адрес

}

3. Поддержание активности

keepalived -D -f /usr/local/etc/keepalived/keepalived.conf

Просмотр сообщений журнала:

tail -f /var/log/messages

Журнал после запуска главного узла A: ARP-сообщения будут транслироваться

[root@srv4 ~]# tail -f /var/log/messages

Sep 20 01:45:29 srv4 Keepalived_vrrp: Configuration is using : 34546 Bytes

Sep 20 01:45:29 srv4 Keepalived_vrrp: VRRP sockpool: [ifindex(2), proto(112), fd(8,9)]

Sep 20 01:45:30 srv4 Keepalived_vrrp: VRRP_Instance(VI_1) Transition to MASTER STATE

Sep 20 01:45:31 srv4 Keepalived_vrrp: VRRP_Instance(VI_1) Entering MASTER STATE

Sep 20 01:45:31 srv4 Keepalived_vrrp: VRRP_Instance(VI_1) setting protocol VIPs.

Sep 20 01:45:31 srv4 Keepalived_vrrp: VRRP_Instance(VI_1) Sending gratuitous ARPs on eth0 for 192.168.8.100

Sep 20 01:45:31 srv4 Keepalived_vrrp: Netlink reflector reports IP 192.168.8.100 added

Sep 20 01:45:31 srv4 Keepalived_healthcheckers: Netlink reflector reports IP 192.168.8.100 added

Sep 20 01:45:31 srv4 avahi-daemon[4029]: Registering new address record for 192.168.8.100 on eth0.

Sep 20 01:45:36 srv4 Keepalived_vrrp: VRRP_Instance(VI_1) Sending gratuitous ARPs on eth0 for 192.168.8.100

Вы можете видеть, что 192.168.8.100/24 связан с eth0 через команду ip a

[root@srv4 bin]# ip a

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

inet6 ::1/128 scope host

valid_lft forever preferred_lft forever

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000

link/ether 00:0c:29:50:2d:9d brd ff:ff:ff:ff:ff:ff

inet 192.168.8.4/24 brd 192.168.8.255 scope global eth0

inet 192.168.8.100/24 scope global secondary eth0

inet6 fe80::20c:29ff:fe50:2d9d/64 scope link

valid_lft forever preferred_lft forever

Журнал после запуска резервного узла B:

Sep 20 01:47:31 hadoopsrv Keepalived_vrrp: Configuration is using : 34262 Bytes

Sep 20 01:47:31 hadoopsrv Keepalived_vrrp: VRRP_Instance(VI_1) Entering BACKUP STATE

Sep 20 01:47:31 hadoopsrv Keepalived_vrrp: VRRP sockpool: [ifindex(2), proto(112), fd(7,8)]

Sep 20 01:47:31 hadoopsrv Keepalived: Starting VRRP child process, pid=20567

4. Установите tomcat на две машины (шаги установки пропущены)

После завершения установки создайте html-файл на компьютере узла A со следующим содержимым:

this is the test page

*
*

from server 192.168.8.4

Доступ обычно осуществляется через следующую проверку URL

http://192.168.8.4:8080/test/test.html

После завершения установки создайте html-файл на компьютере узла B со следующим содержимым:

this is the test page

*
*

from server 192.168.8.6

Доступ обычно осуществляется через следующую проверку URL:

http://192.168.8.6:8080/test/test.html

На главном узле A перейдите по следующему URL:

192.168.8.100:8080/test/test.html

Возвращаемый контент должен быть html на главном узле.

this is the test page

*
*

from server 192.168.8.4

Остановить keepalived узла A:

killall keepalived

Доступ через следующий URL:

192.168.8.100:8080/test/test.html

Возвращаемый контент должен быть контентом на резервном узле:

this is the test page

*
*

from server 192.168.8.6

Также проверьте журнал узла B: обнаружено, что узел B становится главным узлом и передает сообщения ARP.

Sep 20 01:55:44 hadoopsrv Keepalived_vrrp: VRRP_Instance(VI_1) Transition to MASTER STATE

Sep 20 01:55:45 hadoopsrv Keepalived_vrrp: VRRP_Instance(VI_1) Entering MASTER STATE

Sep 20 01:55:45 hadoopsrv Keepalived_vrrp: VRRP_Instance(VI_1) setting protocol VIPs.

Sep 20 01:55:45 hadoopsrv Keepalived_vrrp: VRRP_Instance(VI_1) Sending gratuitous ARPs on eth0 for 192.168.8.100

Sep 20 01:55:45 hadoopsrv avahi-daemon[3769]: Registering new address record for 192.168.8.100 on eth0.

Sep 20 01:55:50 hadoopsrv Keepalived_vrrp: VRRP_Instance(VI_1) Sending gratuitous ARPs on eth0 for 192.168.8.100

7. Отказоустойчивость в облаке

Что такое отказоустойчивость?

Отказоустойчивость – это свойство системы сохранять свою работоспособность после отказа одного или нескольких составных компонентов.

Отказоустойчивая инфраструктура – это комплекс решений, направленных на поддержание постоянной работоспособности оборудования: компьютеров, комплектующих, ПО и локальной сети. Публичное облако на базе VMware позволяет быстро расширить ИТ-инфраструктуру без значительных вложений в модернизацию оборудования.

Современные реалии бизнеса требует максимально эффективного использования дискового пространства. Недостаточно просто разместить данные в облачный сервис и предоставить к ним доступ с применением популярных протоколов. Все чаще на первое место выходит вопрос поддержания высокой отказоустойчивости хранилища. Это означает, что нужно обеспечить постоянный доступ к базам данных и их мобильность.

К счастью, все эти задачи достаточно легко решаются при помощи современных облачных решений. Новые технологии позволяют наращивать ресурсы и увеличивать производительность инфраструктуры, не затрагивая бизнес-процессы. Многие компании, использующие cloud-технологии практические не ощущают разницы в подходах.

Давайте разбираться, каким образом осуществляется отказоустойчивость облачных серверов и действительно ли безопасно размещать инфраструктуру в подобных хранилищах.

Как поддерживается отказоустойчивость

Чаще всего для повышения эффективности облачного хранения применяется многоуровневая система. Этот подход предполагает, что далеко не все данные одинаково востребованы в текущий момент. Следовательно, необходимо варьировать активность доступа и хранить файлы на разных уровнях.

За счет создания иерархии данных можно добиться значительной экономии ресурсов. Для малоактивных и невостребованных файлов могут применяться более дешевые ресурсы. А информация, которая с большой вероятностью потребуется в ближайшее время, будет храниться на более производительном уровне. При необходимости можно автоматически или

вручную настраивать доступность файлов и тем самым распределять нагрузку на облачные сервисы.

Такой подход в большинстве случаев является более рациональным, чем размещение баз данных на физическом оборудовании. Дополнительная отказоустойчивость облачных серверов достигается за счет катастрофоустойчивости. Этот термин означает, что облако продолжит выполнять свои функции даже при выходе из строя всего ЦОДа. Но, как правило, катаклизмы случаются крайне редко, поэтому подобный сценарий можно назвать маловероятным.

Кроме этого, дополнительная отказоустойчивость облачных хранилищ достигается благодаря новейшим технологиям репликации. За счет этого обеспечивается мобильность данных, а также возможность переноса ресурсов с одного сервера на другой без прекращения фактического доступа.

Чаще всего облако базируется на нескольких серверах дата-центра уровня Tier III. Уровень Tier III позволяет обеспечить практически полную отказоустойчивость физической инфраструктуры, а это в свою очередь повышает доступность облачного хранилища.

Дополнительно может применяться принцип географической распределенности облака. В этом случае хранилище базируется на двух независимых площадках, которые находятся на расстоянии друг от друга. Например, это могут быть дата-центры одного поставщика в разных городах. На базе ЦОДов развертывается георасределительный кластер.

В дата-центре присутствуют точки провайдеров связи, благодаря чему обеспечивается наилучшее качество соединения. Такой подход удобен и тем, что в случае отключения связи одним провайдером, всегда есть возможность быстро перестроить маршрутизацию и подключиться к сети через других провайдеров.

Отказоустойчивость на физическом уровне

Для того чтобы добиться высокой отказоустойчивости облака, необходима надежная работа и самого физического оборудования, на котором размещается облачный сервер. На физическом уровне используются следующие решения:

- Отказоустойчивость энергоснабжения. Достигается за счет подключения каждого элемента облака к двум линиям питания, независящим друг от друга. Система АВР помогает переключать нагрузку между основным и резервным питанием, исключая резкие скачки питания. При отсутствии основного питания используются дизель-роторные источники бесперебойного подключения.

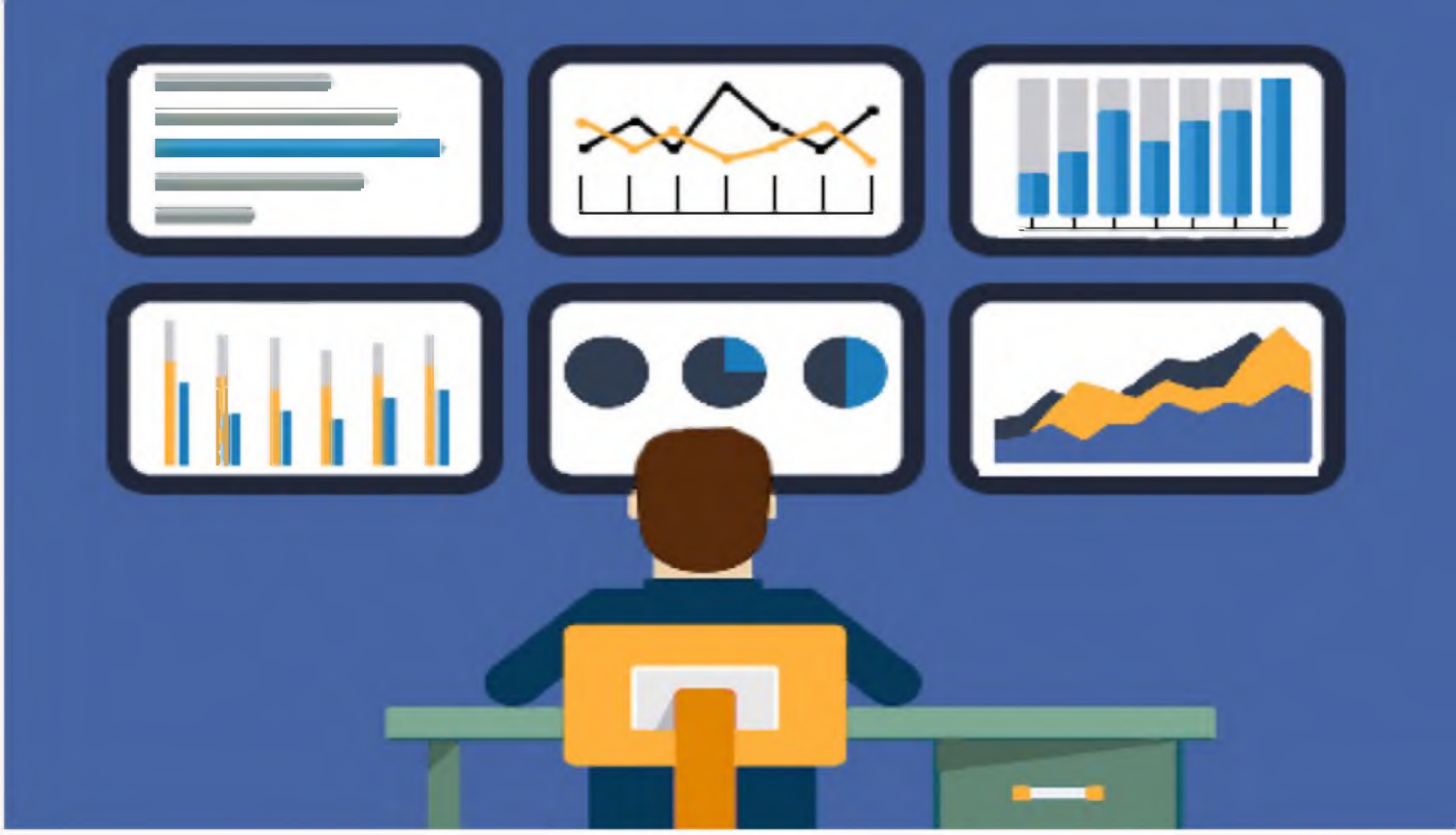
- Резервирование ресурсов. Этот подход предполагает резервирование ресурсов на другом хосте. Если основной сервер выходит из строя, то виртуальные машины сразу запускаются на резервном. В результате облако продолжает работать без сбоев и простоя, а все сервисы и ресурсы остаются доступными для пользователей.
- Миграция в момент техобслуживания. В случае техобслуживания текущего хостинга используется автоматическая бесшовная миграция на другой. Этот процесс не влияет на работу облачных серверов, а значит не затрагивает инфраструктуру компании.

Дополнительная отказоустойчивость физической инфраструктуры достигается и на уровне приложений. Используются сервисы для динамической маршрутизации, балансировки трафика, трансляции API-команд на инфраструктуру и пр.

Рекомендации для отказоустойчивого облака

При развертывании облачной инфраструктуры в облаке стоит учитывать ряд факторов, которые позволят добиться высокой отказоустойчивости:

- Рассчитывать объем ресурсов, которые потребуются для отказоустойчивости. Многие упускают момент резервирования ресурсов как минимум в размере одного хоста, что в случае непредвиденных ситуаций отрицательно сказывается на отказоустойчивости среды.
- Учитывать максимальную загрузку серверов и систем хранения. Серверы архитектуры не стоит загружать более чем на 70-80%, но это правило часто игнорируется. Большая нагрузка может привести к просадке производительности и временным простоям в работе сервисов.
- Заранее рассчитать издержки на масштабирование. Стоит закладывать определенный резерв на случай роста вашей инфраструктуры. Конечно, облако предоставляет возможности масштабирования, но рост можно предсказать заранее и к нужному времени спланировать необходимые ресурсы.
- Учитывать особенности используемого ПО. Некорректное программное обеспечение и приложения не могут напрямую повлиять на безопасность и отказоустойчивость облачного сервиса, однако они могут привести к замедлению работы и различным сбоям. Поэтому перед построением cloud-инфраструктуры стоит заранее понять, подойдет ли сервер для работы с определенным ПО.



МОДУЛЬ 8. Мониторинг, логирование и оповещение событий

1. Зачем и что нужно мониторить
2. Системы для мониторинга
3. Централизованное управление логами Graylog
4. Система визуализации, мониторинга и анализа данных Grafana
5. ELK (Elasticsearch, Logstash и Kibana)
6. Менеджер очередей RabbitMQ
7. Сбор и анализ ошибок при помощи Sentry
8. Инцидент-менеджмент Zabbix

1. Зачем и что нужно мониторить

Что такое мониторинг?

Мониторинг — это постоянный сбор и анализ различных параметров (метрик) поведения системы. С его помощью можно описать и измерить в числовом выражении каждый важный аспект проекта.

Данные из разных точек среды собираются системой мониторинга, которая отвечает за хранение, агрегацию, визуализацию данных и автоматические реагирует на изменения, когда значения соответствует заданным условиям.

Системы мониторинга выполняют множество взаимосвязанных функций. Их первая обязанность – принимать и хранить входящие данные и историю. Текущие значения более полезно видеть в сравнении с прошлыми значениями, чтобы сформировать контекст изменений и тенденций. Это означает, что система мониторинга должна быть способна управлять данными в течение определенного периода времени, а также визуализировать.

Задача мониторинга — предоставлять информацию о сбоях в работе. Она не выполняется разово, изменения должны внедряться вместе с изменениями самого приложения.

Что такое метрики и зачем их собирать?

Метрика – это стандарт для измерения ресурса. Метрики могут ссылаться либо на ресурс и его единицы измерения, либо на данные, собранные об этом ресурсе.

Сбор метрик делится на несколько типов. Есть 3 основных типа метода сбора метрик:

- Метод сбора и анализа проблем с производительностью инфраструктуры (железо, сеть).
- Метод сбора высокоуровневых данных и анализа (веб сервисы, базы данных, очереди и тд).
- Метод сбора и анализа бизнес-метрик.



Рисунок 1. Процесс сбора и анализа метрик

Метрика процесса – количественная мера степени достижения процессом своей цели.

Она должна включать как меры эффективности процесса, так и рейтинги уровней зрелости.

Целевая точка – желаемое значение метрики процесса

Текущее измерение процесса – значение метрики процесса до реализации мероприятия по усовершенствованию

Результат усовершенствования процесса – значение метрики процесса после осуществления мероприятий по усовершенствованию

Программное обеспечение для мониторинга:

- Grafana — универсальная обертка для работы с аналитическими данными, которые хранятся в разных источниках. Она сама ничего не хранит и не собирает, а является лишь универсальным клиентом для систем хранения метрик. Например, с помощью нее можно ходить за цифрами как

в традиционную базу PostgreSQL, так и в специализированные аналитические системы типа Prometheus.

- Prometheus — Система сбора данных временных рядов, разработанная музыкальной компанией SoundCloud для решения внутренних потребностей в быстрой и гибкой обработке продуктовых метрик. Продукт с задачей справился настолько хорошо, что был выпущен за границы SoundCloud и теперь доступен как opensource для всех желающих.
- Zabbix — свободная система мониторинга и отслеживания статусов разнообразных сервисов компьютерной сети, серверов и сетевого оборудования, написанная Алексеем Владышевым.
- Nagios — программа с открытым кодом, предназначенная для мониторинга компьютерных систем и сетей: наблюдения, контроля состояния вычислительных узлов и служб, оповещения администратора в том случае, если какие-то из служб прекращают свою работу.

Методы сбора метрик

- **USE Method:**

USE Method — метод был предложен Brendan Gregg для анализа производительности любой системы. USE — это акроним от терминов Utilization, Saturation и Errors (Утилизация, Насыщение и Ошибки). Таким образом, для каждого ресурса системы необходимо определить и собирать метрики, отражающие каждый из этих показателей (Утилизация, Насыщение и Ошибки) для того, чтобы их можно было использовать при анализе проблем с производительностью.

Использовать USE нужно для выбора и анализа низкоуровневых метрик, например утилизацию процессора, количество свободного места, памяти, превышение допустимой температуры оборудования или нагрузка самой сети.

USE-метод идентифицирует проблемы, которые могут быть узкими местами в системе. К сожалению, системы могут страдать от нескольких проблем с производительностью, поэтому первая обнаруженная проблема может быть не основной. Каждое открытие может быть исследовано с использованием других методологий, прежде чем продолжать использовать USE-метод, по мере необходимости, для итераций над большим количеством ресурсов.

Стратегии дальнейшего анализа включают определение характеристик рабочей нагрузки и детальный анализ. После их завершения (при

необходимости) у вас должны быть доказательства того, является ли корректирующим действием применяемые операции или настройки самого ресурса.

Обратите внимание, что пойманные Ошибки должны быть проверены перед Нагрузочным тестированием (они, как правило, быстрее и проще для исправления).

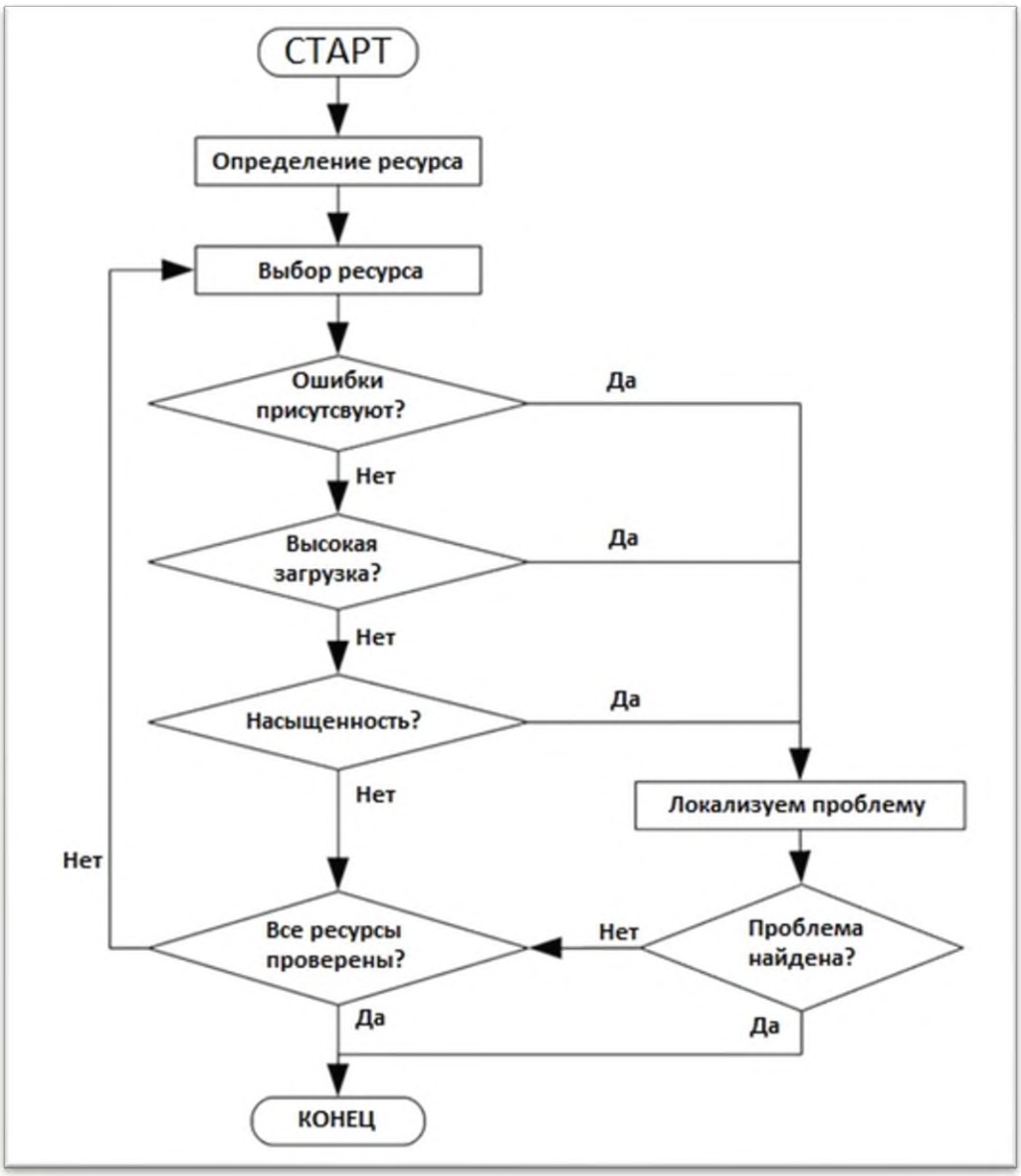


Рисунок 2. Использование USE Method

- **RED Method:**

RED Method — подход, предложенный Tom Wilkie и является акронимом от (Requests) Rate, Errors, Duration (Запросы, Скорость, Ошибки, Продолжительность). Он рассчитан на сбор метрик с самих приложений.

RED нужно использовать для более высокоуровневых сервисов, которые обслуживают запросы. Например, различные веб сервисы, запросы базы данных, очереди и т.д.

RED предлагает способ взглянуть на функционирование и производительность услуг, согласованных для разных сервисов. Таким образом снижается когнитивная нагрузка на дежурных инженеров, что очень важно во время простоев.

Преимущество метода RED в том, что он помогает вам подумать о том, как отображать информацию на ваших информационных панелях. С помощью только этих трех показателей вы можете стандартизировать макет ваших информационных панелей, чтобы упростить их чтение и создание предупреждений при возникновении проблемы.

Справедливо сказать, что этот метод работает только для сервисов, управляемых запросами – например, он не работает для сервисов, ориентированных на пакетную обработку или потоковых данных.

Метод RED следует принципам, изложенным в четырех золотых сигналах, разработанных инженерами по надежности сайтов Google или SRE, которые фокусируются на измерении того, что волнует конечных пользователей при использовании ваших веб-сервисов. В методе RED используются три ключевых показателя, которые отслеживают каждый микросервис в вашей архитектуре:

- (Скорость) Rate – количество запросов в секунду, обслуживаемых вашими сервисами.
- (Ошибки) Errors – количество неудачных запросов в секунду.
- (Продолжительность) Duration – Количество времени, каждый запрос принимает выражаются в виде временного интервала

Rate, Errors и Duration пытаются охватить наиболее очевидные проблемы веб-сервисов. Эти метрики также фиксируют частоту ошибок, которая выражается как доля от частоты запросов (rps).

Для каждого приложения отслеживайте:

- Задержка - время затраченное на запрос
- Трафик - какой объём сети занимает приложение от доступного
- Ошибки - прогнозирование и оценка количества ошибок
- Насыщенность - на какую часть загружена система

- **UCA method:**

UCA акроним от Users, Conversions, Activity (Пользователи, Преобразования, Деятельность) и он нацелен на измерение бизнес метрик сервиса. Его предложил Mike Julian и рассмотрел его в своем курсе Monitor Anything.

Для того, чтобы определить какие бизнес метрики согласно UCA можно собирать, нужно понимать как работает сервис, кто является его пользователями (Users). Для различных отраслей это могут быть количество активных пользователей, посетителей на сайте, количество звонков, количество покупателей сейчас/среднее в день/месяц и т.д.

Далее, Conversions. Это про то, что пользователь из предыдущего пункта каким-либо образом принес доход вашему сервису. Например, пользователь купил платную подписку на сервис, забронировал номер в отеле, купил товар в интернет-магазине, сделал заказ.

Activity показывает, что пользователи продолжают пользоваться вашим продуктом. Например, это могут быть входы на сайт, просмотры товаров, поиски, добавления в корзину, запросы дополнительных сервисов, время, проведенное на сайте и прочее.

UCA позволяет собирать метрики, которые показывают, как бизнес себя чувствует. И, отталкиваясь от этих метрик, можно искать и собирать инфраструктурные метрики с помощью USE и RED и понимать, из-за ухудшения каких показателей нужно поднимать в первую очередь инженеров (алертить).

Что такое система оповещений?

Система оповещений является компонентом системы мониторинга, который выполняет действия на основе изменений метрических значений.

Определения системы оповещений состоят из двух компонентов: условия (или порога), основанного на метриках, и действия, которое нужно выполнить, когда значения выходят за пределы приемлемых условий.

Основная цель системы оповещений – это привлечь внимание человека к текущему состоянию системы. Предупреждение должно содержать информацию о том, что пошло не так и где найти дополнительную информацию. Затем пользователь, отвечающий на предупреждение, может использовать систему мониторинга и связанные с ней инструменты (например, логи) для изучения причины проблемы и ее устранения.

Так же в систему оповещений может входить автоматическое реагирование на проблемы, заложенные заранее определенным шаблоном действий.

Виды мониторинга

1. **Мониторинг оборудования:**

- нагрузка на процессор;
- свободное место в оперативной памяти;
- свободное место на жестком диске;
- нагрузка на сеть;
- нагрузка на жесткий диск — количество операций на запись и чтение;
- количество задач, запущенных на исполнение.

2. **Мониторинг состояния приложений:**

- Количество запросов в единицу времени: час, секунду, день, минуту — зависит от обилия трафика в вашей программе.
- Количество активных пользователей в системе в единицу времени.
- Количество различных записей в СУБД — в целом и новых в единицу времени.
- Количество ошибок в сервисе, которые вы успешно отловили и зарегистрировали.

3. **Мониторинг бизнес-метрик:**

- Сколько пользователей зарегистрировано в системе и сколько пользуется приложением раз в день/неделю/месяц?
- Какой процент людей после регистрации умудряется дойти до формы оплаты и оплатить подписку на ваши услуги? Сколько они платят?
- Как люди пользуются вашими услугами? Какие фичи используют почти все, а какие — единицы клиентов?
- Как много денег вы зарабатываете на вашей программе? Сколько человек оплатили подписку? Какие средние и медианные чеки на клиента? Какова общая выручка и прибыль?

Основные ошибки в настройке и процессе сбора данных

- Человеческий фактор.

- Отсутствие связи между бизнес-задачами и настройкой аналитики.
- «Плавающее» руководство по сбору данных.
- Разрозненные данные.
- Отсутствие проверок.

Комплексный мониторинг – преимущества

- Оперативно обнаруживать и решать проблемы в работе приложений до того, как они повлияют на пользователей;
- Оценивать качество работы приложений на основе объективного мониторинга работы пользователей с приложением;
- Анализировать клиентский опыт на протяжении всего процесса взаимодействия пользователя с системой;
- Оценивать работу подрядчиков. Сравнивать их по количеству пользователей, которые выбирают данную платформу;
- Получить единую точку доступа к общей картине состояния ИТ, возможность исследования «вглубь» – до сырых данных, а также анализировать корреляции.

Комплексный мониторинг – вывод

- Система комплексного мониторинга является зонтичным решением для существующих систем мониторинга. Она объединяет агрегированные данные из других узкоспециализированных систем.
- Комплексное решение закрывает все «слепые зоны», не покрытые существующими в компании системами мониторингами – сбор сырых данных из систем источников в комплексный мониторинг.
- Появляется единая точка для анализа данных и расследования причин инцидентов.
- Бизнес-аналитики, маркетологи, администраторы и руководители подразделений смотрят на одни и те же данные.
- Все события имеют корреляцию по времени.

2. Системы для мониторинга

В современном мире, учитывая архитектурную сложность и территориальную распределенность ИТ-инфраструктур, непросто встретить среднюю или крупную компанию, не использующую системы ИТ-мониторинга. Сбои в работе ИТ-оборудования при отсутствии адекватного мониторинга, использующего инструменты оповещения, визуализации и реагирования, способны серьезно пошатнуть бизнес-процессы, а следовательно, принести существенные финансовые потери. Однако на практике внедрение большинства систем мониторинга ИТ-инфраструктуры сопряжено с различными ограничениями и сложностями.

Выбирая систему мониторинга ИТ-инфраструктуры, необходимо принимать во внимание не только характеристики и возможности отдельно взятого комплекса, но и учитывать такие нюансы, как сложность развертывания, стоимость сопровождения и наличие соответствующих компетенций в компании. Далее рассмотрим наиболее популярные open-source системы мониторинга ИТ-инфраструктуры, сегментируем их на категории в зависимости от основных принципов и парадигмы работы, выявим достоинства и недостатки и определим, существует ли единственно лучшая система или же каждая из рассмотренных в статье может найти своего функционального заказчика.

Zabbix, Nagios

Наиболее популярными и узнаваемыми системами ИТ-мониторинга являются такие решения, как Zabbix и Nagios. Они построены на базе программного обеспечения с открытым исходным кодом и давно зарекомендовали себя как качественные и успешно решающие целевые задачи продукты. И Zabbix, и Nagios способны осуществлять мониторинг большинства компонентов любой современной ИТ-инфраструктуры, включая сетевое оборудование, ОС, различные приложения, базы данных, платформы виртуализации и т. д. Обе системы поддерживают агентский и безагентский сбор данных с целевых источников, имеют инструменты оповещения, визуализации и реагирования, а также сторонние плагины и возможность модернизации логики работы с помощью внешних скриптов. С коммерческой точки зрения у обоих решений предусмотрена платная поддержка, а у Nagios еще и платная версия системы – Nagios XI с дополнительными возможностями и более современной визуальной оболочкой.

Данные решения находятся в одной категории, достаточно схожи по своему функционалу и поэтому имеют аналогичные достоинства и недостатк.

Таблица 1. Достоинства и недостатки Zabbix, Nagios

Достоинства	**Недостатки**
Продвинутые возможности по настройке триггеров и оповещений об инцидентах	Использование РСУБД в качестве подсистемы хранения данных и, как следствие, высокая степень утилизации дисковой подсистемы
Большая библиотека плагинов, существенно расширяющая возможности решений	Ограниченный интерфейс как в части функционала, так и в части визуализации
Наличие официальной поддержки и крупного сообщества	Отсутствие инструментов по масштабированию и отказоустойчивости
Большое количество сторонних систем, поддерживающих интеграцию	Ограниченные возможности по ретроспективному анализу собираемых данных

Prometheus, Graphite

Во вторую категорию входят современные решения, к которым можно отнести такие системы, как Prometheus и Graphite. Они появились сравнительно недавно и активно развиваются. Архитектура обоих решений направлена именно на работу с time-series-данными. Независимо от метода сбора (SNMP/агенты), итоговое представление и хранение данных в обоих решениях будет в формате временных рядов, за исключением, что Graphite хранит данные в кольцевой СУБД Whisper, а Prometheus – в файлах (используя многомерную модель с продвинутыми механизмами индексирования и тегирования).

Поскольку рассматриваемые решения появились сравнительно недавно, разработчики учли многие недостатки предыдущих систем и постарались сделать решения более гибкими и удобными. Помимо наличия

основополагающего функционала по мониторингу ИТ-метрик, Graphite и Prometheus имеют ряд преимуществ, но не обошлось и без недостатков.

Таблица 2. Достоинства и недостатки Prometheus, Graphite

Достоинства	**Недостатки**
Современная архитектура хранения данных и относительно низкая степень утилизации дисковой подсистемы	Ограниченность функционала в части настройки логики триггеров и автоматизации реагирования на инциденты
Оптимизированный язык запросов, позволяющий более удобно работать с собираемыми данными	Отсутствие цельности решения – подсистема сбора данных требует множество сторонних агентов и плагинов. При этом решения не имеют встроенной подключаемой библиотеки
Продвинутые возможности по созданию и кастомизации дашбордов, их сортировке и расположению	Отсутствие официальной поддержки и относительно небольшое сообщество
Продвинутые возможности по настройке временных срезов и применению различных функций прямо на визуализациях	Отсутствие инструментов по масштабированию и отказоустойчивости

Custom IT-monitoring stacks

В третью категорию следует отнести индивидуальные разработки, основанные на различных технологических стеках. Многие крупные компании, учитывая недостатки вышерассмотренных решений, стремятся избавиться от них путем комбинирования технологических компонентов и внутренней разработки.

Принимая во внимание, что Graphite и Prometheus не являются готовыми решениями и не рассчитаны для работы «из коробки», их часто используют как основу для проектирования итоговой системы. Например, в

том же Graphite можно заменить подсистему хранения данных с Whisper популярным InfluxDB, оптимизировав тем самым хранение time-series-данных. Или, при необходимости обеспечения отказоустойчивости и реализации OLAP-сценария обработки данных, можно выбрать связку ClickHouse+ZooKeeper, обеспечив максимально эффективные хранение и обработку данных. Если же требуется более красивый и функциональный интерфейс, к любому из четырех вышерассмотренных решений можно добавить инструмент Grafana, позволяющий по-новому взглянуть на собираемые ИТ-метрики. А если добавить пару самописных сервисов для решения узкоспециализированных задач, удастся получить практически идеальную систему ИТ-мониторинга. Может возникнуть вопрос, почему «практически»? Ответ прост: крайней высокая сложность сопровождения подобной системы. При возникновении внештатных ситуаций или необходимости доработки системы, в случае отсутствия высококвалифицированных специалистов, компания может столкнуться с серьезными проблемами. Чтобы избежать такого развития событий, в штате компании обязательно должны присутствует специалисты с соответствующими компетенциями. В таблице 3 представлены ключевые достоинства и недостатки, присущие рассматриваемому подходу в построении систем по мониторингу ИТ-инфраструктуры.

Таблица 3. Достоинства и недостатки Custom IT-monitoring stacks

Достоинства	**Недостатки**
Возможность комбинирования компонентов и создания индивидуальной системы под конкретные задачи	Архитектурная сложность решения
Наличие инструментов по масштабированию и отказоустойчивости	Низкий уровень ИБ – отсутствие полноценной ролевой модели доступа и сложность организации безопасного взаимодействия между компонентами системы
Продвинутые архитектура хранения и визуализация данных	Отсутствие официальной поддержки

Возможность доработки функционала системы с помощью самописных скриптов и сервисов	Отсутствие поддерживаемой библиотеки плагинов

Альтернативным решением, позволяющим нивелировать рассмотренные выше недостатки, является использование современных коммерческих систем. На зарубежном рынке представлено достаточно много решений от крупных вендоров – ManageEngine OpManager, IBM Tivoli Monitoring, Solarwinds Network Performance Monitor и других. На российском рынке также имеются активно развивающиеся продукты в сфере ИТ-мониторинга, в частности Naumen Network Manager и NGRSOFTLAB Dataplan. Каждый из этих продуктов предлагает современный стек технологий, продвинутые инструменты по оповещению и реагированию, а также качественную официальную поддержку.

Итоги

Проанализировав наиболее популярные решения в области ИТ-мониторинга, разбив их на категории и выделив ключевые преимущества и недостатки, можно подвести итоги. Как оказалось, единственного наилучшего решения не существует. При выборе системы мониторинга ИТ-инфраструктуры необходимо руководствоваться в первую очередь тем, какие задачи и бизнес-цели стоят перед внедряемой системой и сложностью внедрения и сопровождения.

Если вам нужна проверенная временем система для классического мониторинга утилизации аппаратных и программных ресурсов вашей инфраструктуры, с отличными возможностями по оповещению и наличием официальной поддержки, то Zabbix или Nagios – отличный выбор.

Если в вашей инфраструктуре ИТ-мониторинг в первую очередь означает сбор узкоспециализированных метрик с различных приложений, самописных сервисов и систем, подсистемы хранения и визуализации данных в Zabbix или Nagios кажутся откровенно устаревшими, а наличие официальной поддержки для компании не является обязательным условием, то предпочтительны такие решения, как Prometheus или Graphite.

При выборе же решения, в котором необходимо наличие функционала по обеспечению отказоустойчивости хранения собираемых данных, возможности ретроспективного анализа и решения сложных комплексных

задач ИТ-мониторинга, следует обратить внимание на современные коммерческие решения, предлагаемые в том числе и на российском рынке.

3. Централизованное управление логами Graylog

Что такое Graylog?

Graylog –это платформа, которая используется для ведения логов, это файлы, которые содержат данные об успешных операциях, ошибках, предупреждениях и прочее. То есть это отчет о проделанной работе на сервере.

Сервер — это ПК, который специально создан для обработки большого количества данных. Облако в интернете — хранится как раз на сервере. Веб-сайты, включая Microsoft.com — тоже хранятся на серверах. При появлении ошибок или странного поведения программы, процесса, как раз пригодиться лог-файлы, при помощи которых можно будет найти причину ошибки. Так вот Graylog — это инструмент создания этих логов.

Особенности Graylog:

- Open Source — то есть у сервиса открытые исходники.
- Бесплатная версия имеет все необходимые функции.
- Небольшой функционал, но присутствует все самое необходимое.
- После установки инструмент сразу готов к работе, необходима только минимальная настройка.
- Если сравнивать с ELK ресурсоемкость заметно ниже.

Установка и базовые настройки Graylog

Для установки потребуется виртуальный сервер с CentOS и компонентами ElasticSearch, ModgoDB, Graylog4.

В примере используем CentOS 8.

Prerequisites

Желательно проверить правильность настройки таймзоны и имени.

 # timedatectl

 # hostnamectl

Если что-то не так - устанавливаем так как нужно, например:

 # timedatectl set-timezone Europe/Moscow

 # hostnamectl set-hostname graylog.itsft.ru

Обновления обязательны, также установим пакеты для удобства работы, top-ы, etc.

 # dnf upgrade --refresh

 # dnf install epel-release

 # dnf update epel-release

 # dnf -y install nc mc vim atop iftop htop mytop iotop pciutils dos2unix tar lbzip2 zip unzip bash-completion python36 tmux screen telnet wget fail2ban cockpit pwgen

Cockpit - веб-интерфейс управления сервером

(опционально)

 # systemctl enable --now cockpit.socket

 # systemctl start cockpit

проверяем

 # systemctl status cockpit

? cockpit.service - Cockpit Web Service

Loaded: loaded (/usr/lib/systemd/system/cockpit.service; static; vendor preset: disabled)

Active: active (running) since Tue 2021-04-27 12:18:42 MSK; 4s ago

Установим SSL-сертификат

смотрим, где лежат сертификаты (пока самоподписанные).

 # remotectl certificate

certificate: /etc/cockpit/ws-certs.d/0-self-signed.cert

кладём crt и key файлы в /etc/cockpit/ws-certs.d/

Cockpit сам найдёт файлы с сертификатами (2 файла - серт + ключ, или в одном)

 # systemctl restart cockpit

 # remotectl certificate

certificate: /etc/cockpit/ws-certs.d/itsoft.crt

Firewall

(настройки будут индивидуальными для вашей сети, все команды даны для примера)

Если не используем ipv6 - лучше отключить:

 # firewall-cmd --remove-service=dhcpv6-client

Cockpit должен быть доступен только админам, снаружи ему делать нечего. Создадим зону для интранета и добавим нужные адреса подсетей:

 # firewall-cmd --permanent --new-zone=itsoft

 # firewall-cmd --permanent --zone=itsoft --add-source=100.200.200.0/24

Порты (9000 - временно, для веб-интерфейса graylog, 10514 и 5044 для данных):

 # firewall-cmd --permanent --zone=itsoft --add-port=9000/tcp

 # firewall-cmd --permanent --zone=itsoft --add-port=10514/tcp

 # firewall-cmd --permanent --zone=itsoft --add-port=10514/udp

 # firewall-cmd --permanent --zone=itsoft --add-port=5044/tcp

Необходимые сервисы:

 # firewall-cmd --permanent --zone=itsoft --add-service=https

 # firewall-cmd --permanent --zone=itsoft --add-service=http

 # firewall-cmd --permanent --zone=itsoft --add-service=cockpit

После добавления/удаления перезагрузим сервис:

 # firewall-cmd --reload

Проконтролируем, что все правила верные:

 # firewall-cmd --list-all --zone=public

 # firewall-cmd --list-all --zone=itsoft

Проверяем - идём браузером на https://graylog.itsft.ru:9090

SELinux

(настраиваем, а не отключаем его!)

Устанавливаем пакеты, если ещё не установлены, разрешаем апачу подключения

 # dnf -y install policycoreutils policycoreutils-python-utils policycoreutils-python-utils

 # setsebool -P httpd_can_network_connect 1

Проверяем порты 9000, 9200, 27017:

 # semanage port -l | egrep '\s(9000|9200|27017)'

http_port_t tcp 80, 81, 443, 488, 8008, 8009, 8443, 9000

mongod_port_t tcp 27017-27019, 28017-28019

wap_wsp_port_t tcp 9200

wap_wsp_port_t udp 9200

Небольшая проблема, если попытаемся добавить тип для порта 9200, получим ошибку - порт уже определён

 # semanage port -a -t http_port_t -p tcp 9200

ValueError: Port tcp/9200 already defined

Модифицируем запись - добавится локальная и у неё приоритет будет выше, чем у глобальной, но отображаться будут обе:

 # semanage port -m -t http_port_t -p tcp 9200

 # semanage port -l | egrep '\s(9000|9200|27017)'
http_port_t tcp 9200, 80, 81, 443, 488, 8008, 8009, 8443, 9000
mongod_port_t tcp 27017-27019, 28017-28019
wap_wsp_port_t tcp 9200
wap_wsp_port_t udp 9200

Java

 # dnf -y install java-1.8.0-openjdk java-1.8.0-openjdk-devel

Тест:

 # java -version
openjdk version "1.8.0_292"
OpenJDK Runtime Environment (build 1.8.0_292-b10)
OpenJDK 64-Bit Server VM (build 25.292-b10, mixed mode)

Elastic Search

Импортируем ключ репозитория, добавляем конфигурационный файл репозитория, устанавливаем (вместо vim можно использовать nano, mcedit или любой другой редактор по вашему выбору):

 # rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch
 # vim /etc/yum.repos.d/elasticsearch.repo
[elasticsearch-6.x]
name=Elasticsearch repository for 6.x packages
baseurl=https://artifacts.elastic.co/packages/oss-6.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
enabled=1
autorefresh=1
type=rpm-md
 # dnf clean all
 # dnf makecac
 # dnf -y install elasticsearch-oss
. . .
Installed:
elasticsearch-oss-6.8.15-1.noarch

Чтобы Elasticsearch работал с Graylog, необходимо установим имя кластера “graylog”

 # vim /etc/elasticsearch/elasticsearch.yml

В секции --- Cluster --- меняем параметр:

cluster.name: graylog

В секции --- Various --- добавляем последней строкой:

action.auto_create_index: false

Запускаем:

 # systemctl daemon-reload

 # systemctl enable --now elasticsearch

Synchronizing state of elasticsearch.service with SysV service script with /usr/lib/systemd/systemd-sysv-install.

Executing: /usr/lib/systemd/systemd-sysv-install enable elasticsearch

Created symlink /etc/systemd/system/multi-user.target.wants/elasticsearch.service → /usr/lib/systemd/system/elasticsearch.service.

Проверяем:

 # systemctl status elasticsearch

● elasticsearch.service - Elasticsearch

Loaded: loaded (/usr/lib/systemd/system/elasticsearch.service; enabled; vendor preset: disabled)

Active: active (running) since Tue 2021-04-27 12:51:50 MSK; 17s ago

Запускается довольно долго, в зависимости от конфигурации сервера или виртуальной машины. Проверяем:

 # ss -ntlp | grep java

Файлы Elasticsearch расположены здесь:

Configuration: /etc/elasticsearch

JVM settings: /etc/default/elasticsearch

Data files: /var/lib/elasticsearch/data

Log files: /var/log/elasticsearch/

Проверяем:

 # curl -X GET http://localhost:9200

{

"name" : "M_ysIuG",

"cluster_name" : "graylog",

"cluster_uuid" : "CzqgacNrQ5SfByXz85v0wg",

..

 # curl -XGET 'http://localhost:9200/_cluster/health?pretty=true'

{

"cluster_name" : "graylog",

"status" : "green",

...

MongoDB

Добавляем конфигурационный файл репозитория:

 # vim /etc/yum.repos.d/mongodb-org-4.4.repo

[mongodb-org-4.4]

name=MongoDB Repository

baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/4.4/x86_64/

gpgcheck=1

enabled=1

gpgkey=https://www.mongodb.org/static/pgp/server-4.4.asc

 # dnf -y install mongodb-org

 # systemctl enable --now mongod

Проверяем:

 # systemctl status mongod.service

● *mongod.service - MongoDB Database Server*

Loaded: loaded (/usr/lib/systemd/system/mongod.service; enabled; vendor preset: disabled)

Active: active (running) since Tue 2021-04-27 12:59:36 MSK; 5s ago

Файлы MongoDB расположены здесь:

Configuration /etc/mongod.conf

Data files /var/lib/mongodb/

Log files /var/log/mongodb/

Graylog

 # dnf -y install https://packages.graylog2.org/repo/packages/graylog-4.0-repository_latest.rpm

 # dnf -y install graylog-server graylog-enterprise-plugins graylog-integrations-plugins graylog-enterprise-integrations-plugins

Graylog не запускается самостоятельно (об этом есть сообщение в процессе установки).

Сначала настраиваем, потом пробуем запускать.

Генерируем хеш пароля:

 # pwgen -N 1 -s 96

XzN25fRfHv7NGbVODAGDcRWhzd6QAQQa11RXLIVgSXCgaDZexaxE4VlLA1 abmEQjN25BlRVyEp3LNhN6HcJL8GYjV3RQgwkA

 # echo -n MyPassword123 | sha256sum

bc7b8851671f2fda237a53f5057a0376037b6d062e65f965c62aa1d047498 759 -

 # pwgen -N 1 -s 96

g3SpB……5x7U

 # echo -n YourPassword | sha256sum

7b2f50…..b28d0 -

 # vim /etc/graylog/server/server.conf

is_master = true

password_secret = XzN25fRfHv7NEQ...8GYjV3RQgwkA

root_username = admin

root_password_sha2 = bc7b88516...1d047498759

root_email = “admin@itsft.ru”

root_timezone = Europe/Moscow

Параметры эластика только для первого запуска, дальше конфигурация будет производиться уже из веб-интерфейса грейлога:

rotation_strategy = count

elasticsearch_max_docs_per_index = 20000000

elasticsearch_max_number_of_indices = 20

elasticsearch_shards = 1

elasticsearch_replicas = 0

http_bind_address = 0.0.0.0:9000

http_publish_uri = https://graylog.itsft.ru9000/

Также полезно будет настроить Email transport:

transport_email_enabled = true

transport_email_protocol = smtp

transport_email_hostname = mail.itsft.ru

transport_email_port = 25

transport_email_use_auth = true

transport_email_use_tls = false

transport_email_use_ssl = false

transport_email_auth_username = graylog@itsft.ru

transport_email_auth_password = YourMailPassword

transport_email_subject_prefix = [graylog4]

transport_email_from_email = graylog@itsft.ru

transport_email_from_name = Graylog4

Регистрируемся и заходим в панель:

https://www.maxmind.com/en/account/login

Download files → В секции GeoLite2 City → Get Permalinks (1, 2)

 # Database URL

https://download.maxmind.com/app/geoip_download?edition_id=GeoLite2-City&license_key=YOUR_LICENSE_KEY&suffix=tar.gz

 # SHA256 URL

https://download.maxmind.com/app/geoip_download?edition_id=GeoLite2-City&license_key=YOUR_LICENSE_KEY&suffix=tar.gz.sha256

Как выглядит:

В Services → Manage License Keys

Нажимаем "Generate new license key" создастся новый License Key. На email придёт уведомление о создании ключа.

Лицензия активируется в течение 5 минут.

Подставляем в ссылки, полученные на первом шаге ключ, полученный на втором шаге, загружаем файл с базой и файл контрольной суммы:

 # wget --content-disposition 'https://download.maxmind.com/app/geoip_download?edition_id=GeoLite2-City&license_key=xxxxxxxxxx&suffix=tar.gz'

 # wget --content-disposition 'https://download.maxmind.com/app/geoip_download?edition_id=GeoLite2-City&license_key=xxxxxxxxxx&suffix=tar.gz.sha256'

Проверяем целостность архива:

 # sha256sum -c GeoLite2-City_20210427.tar.gz.sha256

GeoLite2-City_20210427.tar.gz: OK

Устанавливаем:

 # tar zxfv GeoLite2-City_20210427.tar.gz

 # mv GeoLite2-City_20210427/GeoLite2-City.mmdb /etc/graylog/server/

Установка GeoLite2 Database выполнена.

Запускаем Graylog:

 # systemctl daemon-reload

 # systemctl enable --now graylog-server.service

Немного ждём, затем смотрим логи запуска:

 # tail /var/log/graylog-server/server.log

...

2021-04-27T15:44:36.005+03:00 INFO [ServerBootstrap] Graylog server up and running.

Если видим ошибки:

ERROR [AuditLogger] Unable to write audit log entry because there is no valid license.

это нормально.

Возникают из-за того, что установлена версия Enterprise, но нет соответствующей лицензии.

Заходим браузером, проверяем что веб-интерфейс доступен, можем залогиниться в веб-интерфейс:

https://graylog.itsft.ru:9000

user: admin

pass: MyPassword123

NGINX

 # vim /etc/yum.repos.d/nginx.repo

[nginx-stable]

name=nginx stable repo

baseurl=http://nginx.org/packages/centos/$releasever/$basearch/

gpgcheck=1

enabled=1

gpgkey=https://nginx.org/keys/nginx_signing.key

module_hotfixes=true

[nginx-mainline]

name=nginx mainline repo

baseurl=http://nginx.org/packages/mainline/centos/$releasever/$basearch/

gpgcheck=1

enabled=0

gpgkey=https://nginx.org/keys/nginx_signing.key

module_hotfixes=true

 # dnf config-manager --enable nginx-mainline

 # dnf -y install nginx

Installed:

nginx-1:1.20.0-1.el8.ngx.x86_64

Создаём конфиг nginx, размещаем файлы сертификатов в /etc/nginx/ssl (у нас будут в одном файле сертификат + ключ):

```
# vim /etc/nginx/conf.d/graylog.conf
server
{
 listen 80 default_server;
 server_name graylog.itsft.ru;
 return 301 https://$host$request_uri;
}
server
{
 listen      443 ssl http2;
 server_name graylog.itsft.ru;
 access_log  /var/log/nginx/graylog_access.log;
 error_log   /var/log/nginx/graylog_error.log;
 ssl_certificate     /etc/nginx/ssl/star_itsft.crt;
 ssl_certificate_key /etc/nginx/ssl/star_itsft.crt;
 ssl_session_timeout 1d;
 ssl_session_cache   shared:MozSSL:10m;  # about 40000 sessions
 ssl_session_tickets off;
 # Supports Firefox 27, Android 4.4.2, Chrome 31, Edge, IE 11 on Windows 7,
 # Java 8u31, OpenSSL 1.0.1, Opera 20, and Safari 9 and above
 ssl_protocols TLSv1.2 TLSv1.3;
 ssl_ciphers         ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384;
 ssl_prefer_server_ciphers off;
 # HSTS (ngx_http_headers_module is required) (63072000 seconds)
 add_header Strict-Transport-Security "max-age=63072000" always;
 # OCSP stapling
 ssl_stapling on;
 ssl_stapling_verify on;
 location /
```

```
        {
          proxy_set_header Host $http_host;
          proxy_set_header X-Forwarded-Host $host;
          proxy_set_header X-Forwarded-Server $host;
          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
          proxy_set_header X-Graylog-Server-URL https://$server_name/;
          proxy_pass       http://127.0.0.1:9000;
        }
      }
      # systemctl enable nginx
      # systemctl start nginx
      # systemctl status nginx
      ● nginx.service - nginx - high performance web server
        Loaded: loaded (/usr/lib/systemd/system/nginx.service; disabled; vendor
preset: disabled)
        Active: active (running) since Tue 2021-04-27 17:12:52 MSK; 6s ago
```

Если получили ошибку:

Oct 11 10:00:00 itsft.ru systemd[1]: Starting nginx - high performance web server...

Oct 11 10:00:00 itsft.ru nginx[315253]: nginx: [emerg] open() "/var/run/nginx.pid" failed (13: Permission denied)

Oct 11 10:00:00 itsft.ru systemd[1]: nginx.service: Failed to parse PID from file /var/run/nginx.pid: Invalid argument

Oct 11 10:00:00 itsft.ru systemd[1]: nginx.service: Failed with result 'protocol'.

Oct 11 10:00:00 itsft.ru systemd[1]: Failed to start nginx - high performance web server.

То это из-за SELinux. Исправляем:

```
# systemctl stop nginx
# touch /var/run/nginx.pid
# chcon -u system_u -t httpd_var_run_t nginx.pid
# systemctl start nginx
```

Контролируем:

```
# ll -Z nginx.pid
-rw-r--r--. 1 root root system_u:object_r:httpd_var_run_t:s0 7 Oct 15 16:08
nginx.pid
```

 # firewall-cmd --permanent --zone=itsoft --remove-port=9000/tcp

 # firewall-cmd –reload

Немного реконфигурим Graylog:

 # vim /etc/graylog/server/server.conf

http_bind_address = 127.0.0.1:9000

http_publish_uri = https://graylog.itsft.ru/

 # systemctl restart graylog-server.service

Теперь Graylog слушает только на локалхосте, если открывали порт 9000, то необходимости в нём больше нет:

 # firewall-cmd --permanent --zone=itsoft --remove-port=9000/tcp

 # firewall-cmd –reload

Размер БД

Установим размер индекса, так как виртуальный сервер имеет ограниченный объём диска.

System → Indices → Default index set → Edit

В Index Rotation Configuration:

Select rotation strategy - Index Size

Max size per index - 4294965097 (4GiB)

В Index Retention Configuration:

Select retention strategy - Delete Index

Max number of indices - 8

Итого общий размер индексов будет не более 32GiB

4. Система визуализации, мониторинга и анализа данных Grafana

Что такое Grafana?

Grafana — это платформа с открытым исходным кодом для визуализации, мониторинга и анализа данных. Этот инструмент, в сочетании с Graylog, — часть нашей двухсторонней системы мониторинга поведения пользователей и производительности системы. Grafana позволяет пользователям создавать дашборды с панелями, каждая из которых отображает определенные показатели в течение установленного периода времени. Каждый дашборд универсален, поэтому его можно настроить для конкретного проекта или с учетом любых потребностей разработки и/или бизнеса.

Grafana — отличный инструмент для визуализации данных. Доступность и наблюдаемость для всех пользователей обеспечивается за счет упрощенной установки и настройки. Все пользователи могут найти приложение для отслеживания производительности удовлетворительным благодаря привлекательному и настраиваемому интерфейсу.

Панель управления хорошо оснащена для понимания сложных данных, таких как графики, тепловые карты, гистограммы и географические карты, и она постоянно обновляется. Инструмент предоставляет множество решений визуализации для понимания данных в соответствии с бизнес-потребностями конкретного проекта. Вот ключевые особенности Grafana, которые вам следует знать:

- Создание шаблонов приборной панели. Одна из самых важных функций Grafana, создание шаблонов информационных панелей, помогает вам создавать информационные панели, которые можно повторно использовать для различных целей и совместно использовать в командах вашей организации.
- Подготовка. Хотя создать единую панель мониторинга просто, щелкнув, перетащив и отпустив ее, пользователи, которым требуется много панелей мониторинга, захотят использовать сценарий для автоматизации процесса. Grafana позволяет создавать сценарии для чего угодно и получать контроль над большим количеством информационных панелей.

- Аннотации. В Grafana эта функция отображается как маркер графика и полезна для сопоставления данных в случае, если что-то пойдет не так. Вы можете создавать аннотации вручную (щелкните график и введите текст, удерживая клавишу Control) или получить данные из любого источника данных.
- Режим киоска и плейлисты. Если вы хотите отображать свои информационные панели Grafana на телевизионном мониторе, вы можете использовать функцию списка воспроизведения, чтобы выбрать, какие информационные панели вам нужно просматривать, и заставить их циклически перемещаться по экрану. В режиме только для просмотра режим киоска скрывает все элементы пользовательского интерфейса, которые вам не нужны.
- Пользовательские плагины. Вы можете использовать плагины для расширения Grafana и интеграции его с другим программным обеспечением, визуализациями и т. Д. Все, что создает метку времени и значение, можно визуализировать в Grafana с помощью всего нескольких строк кода.
- Оповещения и ловушки для предупреждений. Если происходит ожидаемый сценарий, предупреждения активируются как растяжки. Об этих событиях можно сообщить команде мониторинга через Slack или другой канал связи.
- Разрешения и команды. Когда в компании есть одна Grafana и несколько команд, они часто хотят разделить вещи, но при этом использовать информационные панели. Если вы используете Grafana Enterprise, вы можете создать команду пользователей, а затем установить разрешения для папок, информационных панелей и вплоть до уровня источника данных.
- Источники данных SQL. Встроенная поддержка SQL в Grafana позволяет вам превращать все, что есть в базе данных SQL, в данные метрик, которые вы можете графически отображать. Источники данных SQL используются опытными пользователями для выполнения множества увлекательных вещей, например для создания бизнес-панелей.
- Аутентификация. Grafana поддерживает различные стили аутентификации, включая LDAP и OAuth, и позволяет отображать пользователей в организации. Вы также можете сопоставить пользователей с командами в Grafana Enterprise: Grafana поможет вам сопоставить команды

в ваших внутренних системах с командами Grafana, если в вашей организации есть собственная система аутентификации. В результате вы сможете автоматически предоставлять доступ к информационным панелям сотрудникам, принадлежащим к определенным командам.

Термины Grafana

Панель — базовый элемент визуализации выбранных показателей. Grafana поддерживает панели с графиками, единичными статусами, таблицами, тепловыми картами кликов и произвольным текстом, а также интеграцию с официальными и созданными сообществом плагинами (например, карта мира или часы) и приложениями, которые также можно визуализировать. Можно настроить стиль и формат каждой панели; все панели можно перетаскивать на новое место, перестраивать и изменять их размер.

Дашборд — набор отдельных панелей, размещенных в сетке с набором переменных (например, имя сервера, приложения и датчика). Изменяя переменные, можно переключать данные, отображаемые на дашборде (например, данные с двух отдельных серверов). Все дашборды можно настраивать, а также секционировать и фрагментировать представленные в них данные в соответствии с потребностями пользователя. В проекте Grafana участвует большое сообщество разработчиков кода и пользователей, поэтому существует большой выбор готовых дашбордов для разных типов данных и источников.

В дашбордах можно использовать аннотации для отображения определенных событий на разных панелях. Аннотации добавляются настраиваемыми запросами в Elasticsearch; на графике аннотация отображается вертикальной красной линией. При наведении курсора на аннотацию можно получить описание события и теги, например, для отслеживания ответа сервера с кодом ошибки 5xx или перезапуска системы. Благодаря этому можно легко сопоставить время, конкретное событие и его последствия в приложении и исследовать поведение системы.

Передовые методы использования инструмента Grafana на примере компании Logicify

Использование Grafana во внутренних проектах

Для своего внутреннего проекта компания подключила Grafana к InfluxDB, базе данных временных рядов, чтобы визуализировать изменения параметров микроклимата в офисе и реагировать на них соответствующим

образом. Набор датчиков измеряет температуру, влажность, атмосферное давление и уровень CO2 в каждой зоне офиса; эти параметры собираются и визуализируются в виде графиков Grafana на большом мониторе и в режиме онлайн.

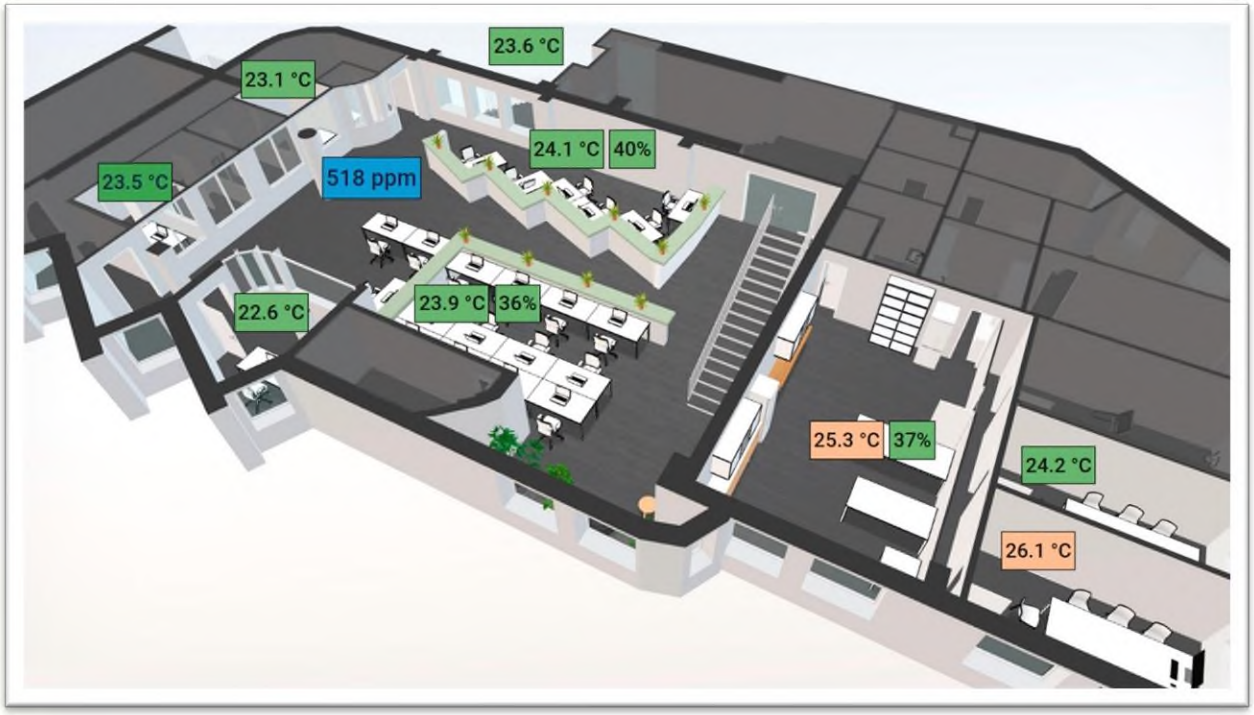


Рисунок 3. Визуализация параметров микроклимата в офисе

Это позволило компании отслеживать параметры качества воздуха и реагировать на изменения: открывать окна, если уровень CO2 слишком высок, включать и выключать кондиционер и увлажнители воздуха.

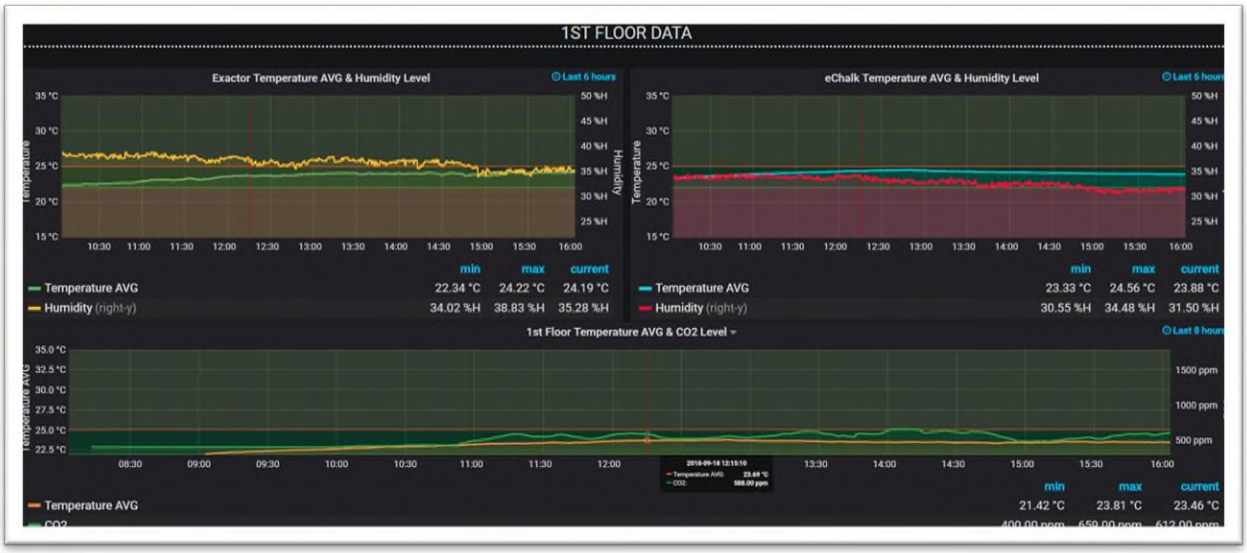


Рисунок 4. Дашборд Grafana с параметрами микроклимата в офисе

Благодаря графикам и аннотациям временных рядов, отображаемым в Grafana, были анализированы тенденции изменения микроклимата в

офисах в течение несколько месяцев и времен года. Также данный инструмент был использован для визуализации некоторых полезных виджетов и сведений (прогноз погоды, курсы обмена валют, внутренние календари) на большом кухонном мониторе.

Как использовать Grafana в настраиваемых веб-приложениях

Grafana + Graylog

Инструмент Graylog может быть использован для хранения журналов веб-приложений, управления ими и мониторинга их производительности как на этапе разработки, так и на этапе их эксплуатации. Grafana — это инструмент, который преобразует журналы, хранящиеся в Graylog, в визуальные формы для аналитического и системного мониторинга. Инструменты Graylog и Grafana существуют независимо друг от друга, не нужно создавать никаких специальных сложных средств интеграции для подключения их друг к другу. Поскольку Graylog хранит все данные журнала в Elasticsearch, одном из источников данных Grafana, можно использовать определенный индекс Elasticsearch, где хранятся журналы, для подключения Grafana к Graylog.

Какие показатели можно визуализировать в Grafana для веб-приложения?

Журналы с простым текстом или уведомления об ошибках не «интересны» инструменту Grafana, поскольку его основная цель — визуализировать данные в виде графиков, диаграмм и таблиц. Был написан пользовательский модуль для Django для сбора данных, которые хотелось бы отслеживать по каждому обработанному запросу и отклику сети/работника. Данные включали в себя не только статус «успех»/«неудача», но и набор структурированных полей (как общего характера, так и относящихся к проекту), как то:

- версия приложения
- уникальный идентификатор каждого запроса
- время отклика и статус
- код ошибки (если есть)
- IP-адрес, с которого был отправлен запрос
- информация о пользователе (e-mail, юзернейм для зарегистрированных пользователей, роль, разрешения)
- устройство и т. д.

Django помещает пользовательские структурированные аналитические записи в Graylog, который сохраняет их в отдельном потоке. Хотя эти данные можно визуализировать с помощью встроенных дашбордов Graylog, они не так хорошо выглядят, как дашборды Grafana. Поэтому Grafana используется для считывания этих аналитических данных и их визуализации. Таким образом, можно отслеживать производительность приложения и загружать данные как в режиме реального времени, так и в ретроспективе.

Grafana как инструмент отладки

Дашборды Grafana главным образом помогают в отладке приложений. Если конечный клиент сообщает о проблеме, Grafana дает возможность отличить ошибки на стороне клиента/сервера от реальных ошибок или брешей в логике приложения. Отслеживаются все веб-запросы, инициированные клиентом (используя адрес электронной почты), администраторами приложений и самим приложением в течение заданного периода времени и методом исключения находим причину.

Также проводится отладка и исправление ошибки, если на дашборде замечена аномалия в графиках загрузки и производительности приложения. Следующий пример графика Grafana показывает время отклика на веб-запросы в течение определенного периода времени. Для каждого веб-запроса отслеживаются максимальное, минимальное и среднее время отклика. Если получен запрос, на обработку которого ушло слишком много времени, масштабируется определенная часть графика и исследуется проблема.



Рисунок 5. График Grafana, отображающий время отклика на веб-запросы

Другой график показывает загрузку системы в течение установленного периода времени и полезен для отслеживания трафика. Если замечен необычный всплеск активности, например, в нерабочее время или в выходные дни, то он непременно исследуется. Такой всплеск может быть вызван, например, сканерами Google, индексирующими контент веб-сайта, или вредоносными ботами, сканирующими систему на наличие уязвимостей. Опять-таки, каждый случай исследуется и рассматривается соответствующим образом.

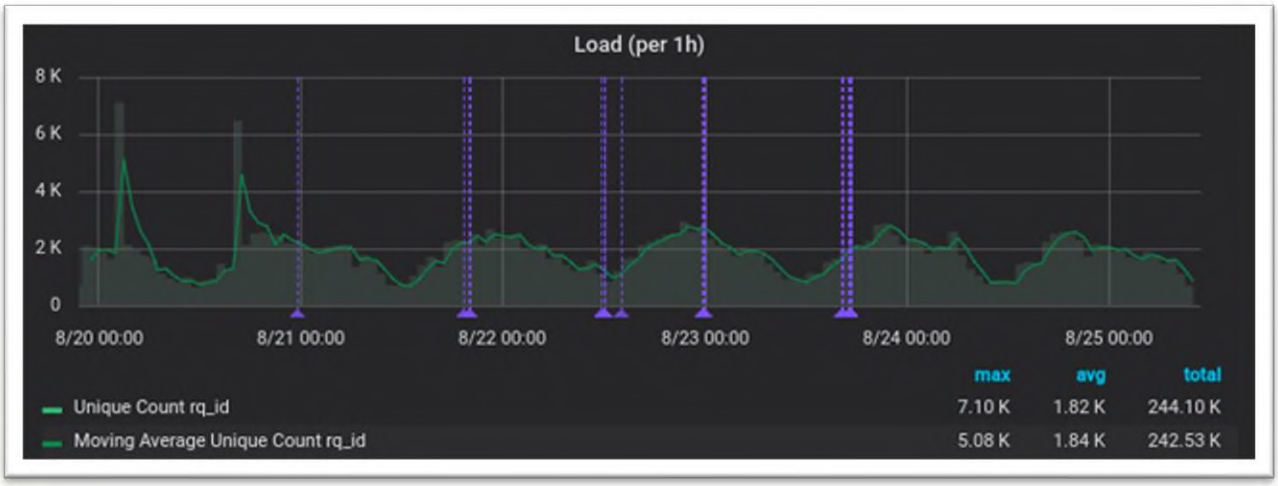


Рисунок 6. График Grafana, отображающий загрузку приложения

Grafana имеет встроенный механизм оповещения (например, по электронной почте или с помощью уведомлений через Slack) согласно определенным правилам. Данная возможность инструмента Grafana не используется, поскольку все уведомления настроены в Graylog. Однако некоторые проблемы с производительностью системы можно увидеть только после прогона программы, например, необычно долгое время отклика на веб-запрос. Уведомления Graylog об этом не придет, но аномалия будет четко видна на графике Grafana. Итак, оба инструмента дополняют друг друга, когда появляется какая-то проблема: на высоком уровне проверяется Grafana, чтобы понять, что произошло и почему, а затем изучается Graylog, используя конкретный идентификатор запроса.

В отличие от Graylog, используемого как для разрабатываемых, так и для используемых приложений, Grafana применяется только для используемых приложений. Единственным исключением, когда Grafana используется для приложения, которое все еще на этапе разработки, является тестирование производительности. Загрузку системы эмулируется с

помощью JMeter, затем проверяются дашборды Grafana, чтобы увидеть, как она реагирует.

Grafana как инструмент бизнес-аналитики

Помимо целей отслеживания производительности и отладки, дашборды Grafana являются мощным инструментом для принятия обоснованных бизнес-решений. При правильной настройке (желательно в тандеме с сервисом Google Analytics) Grafana может визуализировать настраиваемую аналитику поведения пользователя в системе в виде круговых диаграмм, гистограмм времени и других графических элементов. На их основе стороны, заинтересованные в продукте, могут принимать решения о дальнейшем масштабировании приложения, добавлении или удалении некоторых функций и улучшении цикла взаимодействия с клиентами.

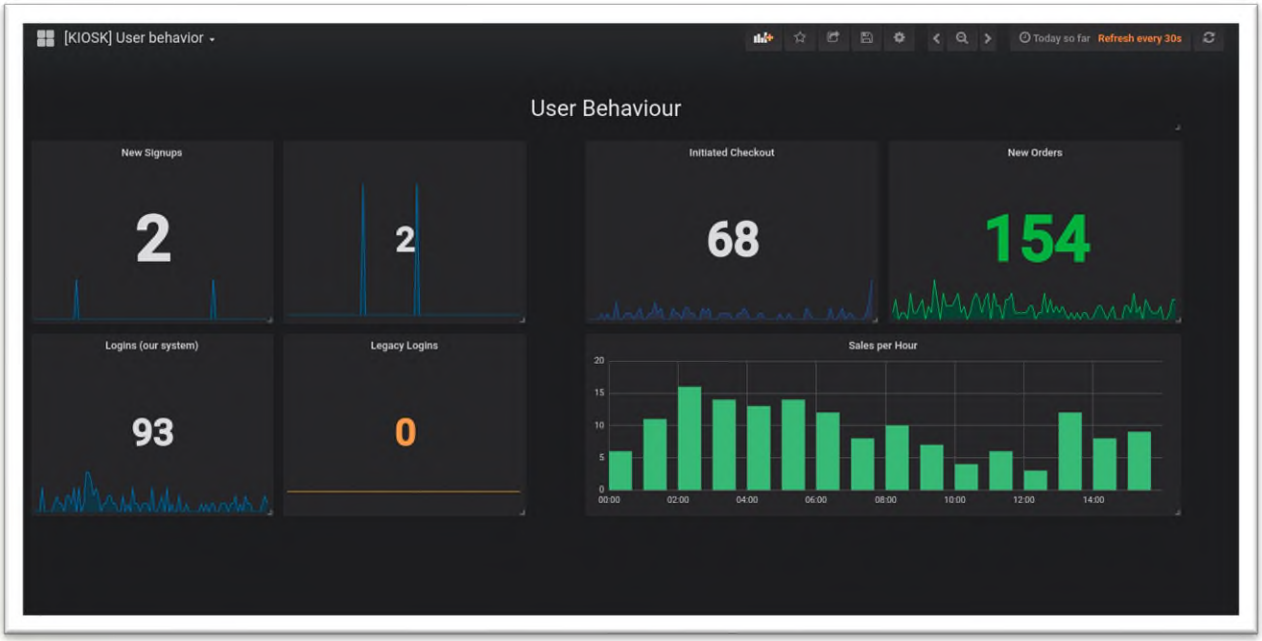


Рисунок 7. Дашборд Grafana, отображающий поведение пользователя в приложении для электронной коммерции

Поскольку вышеприведенный дашборд ориентирован в большей мере на бизнес, разработчики используют его для внутренних целей, скорее, как инструмент обеспечения, чтобы отслеживать поток клиентов в приложении для электронной коммерции: регистрации, авторизации, заказы, размещенные в течение заданного периода времени.

Вот 2 реальных проекта, где с помощью Grafana удалось повысить удобство использования веб-приложения.

- С помощью Grafana регулярно контролируется состояние периодических заказов в системе и фильтруются несостоявшиеся заказы. Эти заказы основаны на подписке, то есть генерируются в системе каждый месяц, а деньги автоматически списываются с банковских счетов клиентов. Иногда платежи не проходят (недостаточно денег/отказ финансового учреждения), и тогда администраторы системы проверяют ситуацию и связываются с клиентами, чтобы повторно сгенерировать заказ вручную. Таким образом, ни один заказ не выпадает из поля зрения; довольны и клиенты, и поставщики.
- При использовании отчетов, созданных с помощью Grafana для приложения электронной коммерции, выяснилось, что большой процент новых клиентов уходит со страницы оформления заказа, хотя в их корзинах уже есть товары. Этот вывод был подкреплен отчетами сервиса Google Analytics, поэтому процедура оформления заказа была проанализирована шаг за шагом и улучшена: теперь пользователи могут оформить заказ в 2 клика. В результате коэффициент обращаемости посетителей в покупателей вырос — выросла и прибыль поставщика.

Grafana является важным компонентом системы мониторинга как для внутренних, так и для внешних проектов. У этого программного средства открытый исходный код, плюс большое и активное сообщество разработчиков. Но больше всего нравится гибкость — оно поддерживает множество источников данных и позволяет легко настраивать дашборды и панели.

Установка и настройка Grafana

Grafana поддерживает установку на самые популярные операционные системы: Linux (Debian, Ubuntu, CentOS, Fedora, OpenSuse, RedHat), Mac и Windows. Допустимыми базами данных, в которых возможно хранение данных о пользователях и дашбордах, являются SQLite (используется по умолчанию), MySQL и PostgreSQL. В качестве демонстрационных серверов будем использовать «Облачные серверы» с операционными системами CentOS Stream и Ubuntu, на которые установим Grafana с базой данных SQLite.

Перед началом установки Grafana, подготовим серверы. В консоли управления Selectel перейдём в раздел Облачная платформа и нажмём кнопку Создать сервер.

На следующем представлении выберем готовую конфигурацию сервера с 1 vCPU и 1 ГБ RAM с операционной системой CentOS 8 64-bit.

Наш сервер будет иметь 10 ГБ на жестком диске. Для целей демонстрации этого достаточно. Прокручиваем вниз и нажимаем кнопку Создать. Аналогичным образом создадим сервер с Ubuntu 20.04 LTS 64-bit.

После проделанных манипуляций оба сервера появятся в списке доступных. Приступим к установке Grafana.

Установка Grafana на CentOS Stream

Установка Grafana на CentOS Stream выполняется из репозитория разработчиков. Скачать дистрибутив можно также с официального сайта. Создадим новую конфигурацию репозитория:

sudo nano /etc/yum.repos.d/grafana.repo

Впишем в этот файл следующие переменные со значениями:

[grafana] name=grafana baseurl=https://packages.grafana.com/oss/rpm repo_gpgcheck=1 enabled=1 gpgcheck=1 gpgkey=https://packages.grafana.com/gpg.key sslverify=1 sslcacert=/etc/pki/tls/certs/ca-bundle.crt

Далее установим Grafana:

sudo yum install grafana -y

После установки включаем автозагрузку и запускаем демон Grafana:

sudo systemctl enable grafana-server

sudo systemctl start grafana-server

Установка завершена. Grafana на CentOS Stream готова к использованию. Проверим текущий статус сервиса:

sudo systemctl status grafana-server

В результате получим вывод:

grafana-server.service - Grafana instance Loaded: loaded (/usr/lib/systemd/system/grafana-server)

Установка Grafana на Ubuntu 20.04

На Ubuntu Grafana устанавливается из репозитория производителя. Скачаем GPG-ключ и добавим его в список надежных:

wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -

После этого добавим в систему репозиторий Grafana:

sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"

Далее обновляем кэш APT и установим Grafana:

sudo apt update

sudo apt install grafana

После установки включаем автозагрузку и запускаем демон Grafana:

sudo systemctl enable grafana-server

sudo systemctl start grafana-server

Установка завершена. Grafana на Ubuntu 20.04 готова к использованию. Проверим текущий статус сервиса:

sudo systemctl status grafana-server

В результате получим вывод:

Установка Grafana на Ubuntu 20.04

На Ubuntu Grafana устанавливается из репозитория производителя. Скачаем GPG-ключ и добавим его в список надежных:

wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -

После этого добавим в систему репозиторий Grafana:

sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"

Далее обновляем кэш APT и установим Grafana:

sudo apt update

sudo apt install grafana

После установки включаем автозагрузку и запускаем демон Grafana:

sudo systemctl enable grafana-server

sudo systemctl start grafana-server

Установка завершена. Grafana на Ubuntu 20.04 готова к использованию. Проверим текущий статус сервиса:

sudo systemctl status grafana-server

В результате получим вывод:

grafana-server.service - Grafana instance Loaded: loaded (/usr/lib/systemd/system/grafana-server)

Настройка брандмауэра для доступа к Grafana

По умолчанию Grafana доступна на порту 3000. При использовании firewalld или iptables, необходимо разрешить сетевой доступ к этому порту из внешнего мира.

Набор команд для firewalld (добавление нового правила и перезагрузка сервиса firewalld для применения конфигурации):

firewall-cmd --zone=public --add-port=3000/tcp --permanent

systemctl reload firewalld

Набор команд для iptables:

nano /etc/sysconfig/iptables

Вписываем в раздел OUTPUT ACCEPT следующую строку:

-A INPUT -p tcp -m tcp --dport 3000 -m state --state NEW -j ACCEPT

И перезапускаем iptables:

sudo systemctl restart iptables

После выполнения описанных в этом разделе манипуляций сетевой экран будет разрешать подключения по порту 3000. В этом можно убедиться, открыв в браузере интерфейс Grafana. Логин и пароль по умолчанию admin.

Если не удается войти в Grafana или вы поменяли пароль и не можете его вспомнить, через CLI административный пароль Web-интерфейса можно сбросить:

grafana-cli admin reset-admin-password --homepath "/usr/share/grafana" новый_пароль

После выполнения команды выше, можно входить с новым паролем.

Источники данных Grafana и установка плагинов

Grafana поддерживает из коробки различные источники данных:

- Prometheus,
- Graphite,
- OpenTSDB,
- InfluxDB,
- Elasticsearch и другие.

Полный список можно посмотреть в документации. Перечень встроенных источников данных может быть расширен при помощи сторонних плагинов. Один из таких плагинов — плагин Zabbix для Grafana. Мониторинг — одно из основных назначений использования Grafana.

Установим этот плагин и добавим его в Grafana.

grafana-cli plugins install alexanderzobnin-zabbix-app

Теперь перезагрузим сервис Grafana:

systemctl restart grafana-server

В интерфейсе Grafana перейдём в Configuration -> Plugins и обнаружим здесь плагин для Zabbix.

Перейдем к настройке плагина и нажмём на кнопку Enable.

С момента включения плагина, в меню Data Sources появится новый источник данных — Zabbix. Нажмём на кнопку Select и настроим источник данных.

В настройках плагина укажем URL-адрес, имя пользователя и пароль для доступа по API к Zabbix.

Нажимаем на кнопку Save & Test и получаем подтверждение корректности настроек.

С этого момента данных из Zabbix могут быть использованы для создания визуализаций.

Создание и настройка панелей и дашбордов

Панели и дашборды — основные сущности представлений внутри Grafana. Каждый дашборд состоит из набора панелей. Для создания дашбордов перейдём в представление Dashboards и нажмём на кнопку New Dashboard.

На следующем шаге будет предложено добавить на дашборд новую панель.

Grafana имеет встроенные панели, которые можно сразу начинать использовать. По умолчанию представлены следующие типы:

- Graph — панель с графиками с возможностью комбинировать несколько метрик на одной панели.
- Stat (ранее SingleStat) — панель с одиночным графиком и возможностью отображения моментального значения метрики.
- Gauge — панель в формате спидометра, есть возможность ограничить верхнее значение на шкале.
- Bar Gauge — панель с возможностью отображения нескольких метрик на вертикальной гистограмме.
- Table — панель с представлением в виде таблицы, на которой можно отображать значения нескольких метрик.
- Text — панель для отображения произвольного текста (подписи).
- Heatmap — панель для отображения тепловой карты значений метрик.
- Alert list — панель для отображения событий из внешних систем.
- Dashboard list — комбинированная панель для отображения дашбордов, добавленных в избранное.
- News — панель для отображения новостной ленты из внешних источников.
- Zabbix problems — панель для отображения событий из системы мониторинга Zabbix.

- Logs — панель для отображения строчек лога, которые собираются одной из внешних систем.

Далее выберем в выпадающем меню источник данных Zabbix и укажем группу, хост, приложение и элемент данных. Если всё выполнено правильно — на графике появятся данные.

Grafana поддерживает различные типы визуализаций:

- графические,
- табличные,
- гистограммы,
- тепловые карты,
- карты сетевого взаимодействия в Grafana и другие.

При создании новой панели, в правой части экрана в разделе Visualization есть возможность выбрать тип визуализации. Перейдем к созданию SingleStat (отображается как Stat) — панели с комбинацией численного и графического представлений. После ввода специфических данных для Zabbix, нажмём кнопку Apply.

В результате выполненных действий, увидим на дашборде две панели.

У каждой панели и каждого дашборда в Grafana есть свой набор настроек. Настройки первой открываются через выпадающее меню панели, а настройки второго откроются после нажатия на шестеренку в верхнем правом углу экрана.

На изображении ниже показаны настройки дашборда.

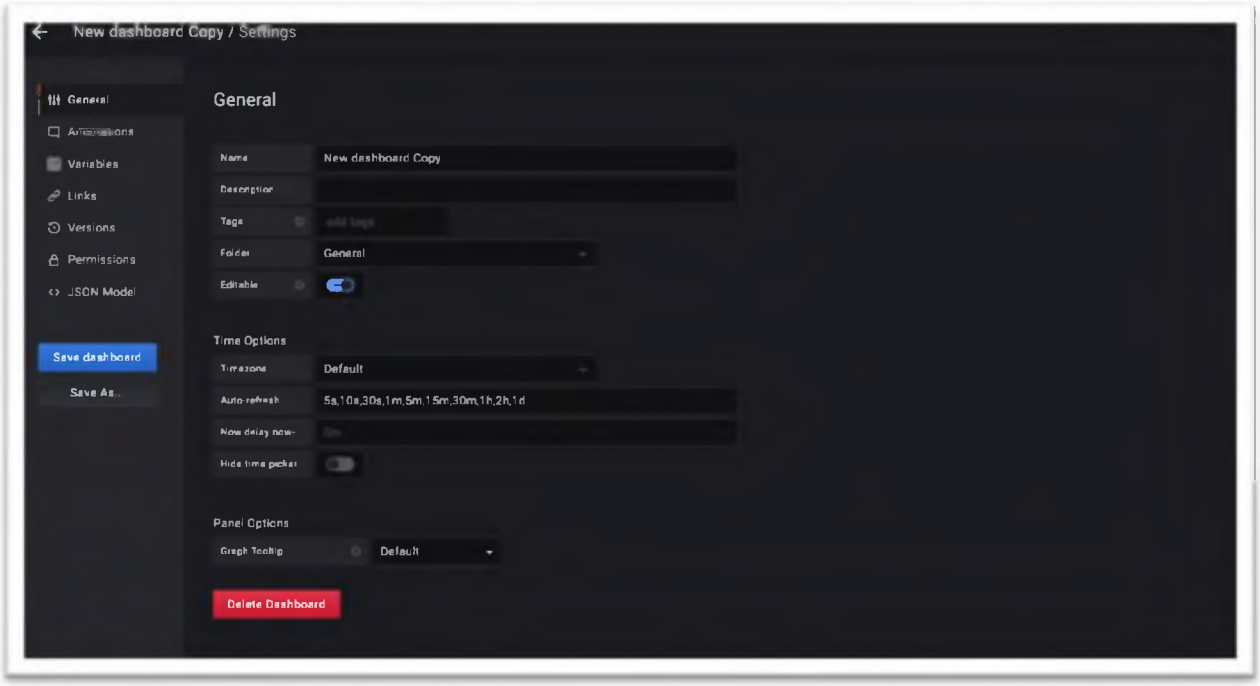


Рисунок 8. Настройки дашборда

5. ELK (Elasticsearch, Logstash и Kibana)

Для чего нужен ELK?

Чтобы определить, почему деньги пропали с банковского счета или кто именно из сотрудников слил персональные данные клиентов в сеть и найти причины прочих инцидентов в корпоративном масштабе, нужна полная картина произошедшего. Составить ее поможет анализ системных логов – журналов, где зафиксированы все события, случившиеся в системе за конкретный промежуток времени. Сегодня наиболее популярным инструментом для построения решений по мониторингу корпоративной инфраструктуры считается ELK – Elasticsearch, Logstash и Kibana. Изначально эти 3 продукта с открытым исходным кодом развивались отдельно друг от друга, но в 2015 году стали принадлежать одной компании Elastic. Сегодня Elasticsearch, Logsatsh и Kibana используются вместе для задач сбора, хранения и анализа данных.

Мощные возможности ELK-инфраструктуры позволяют не только анализировать системные логи в рамках задачи администрирования корпоративного ИТ-ландшафта. Также эта Big Data система отлично подходит для решения следующих бизнес-задач:

- агрегация товаров с множества интернет-магазинов, фильтрация и поиск разных свойств продукции;
- агрегация данных из различных систем, вычисление и отображение показателей для комплексного анализа бизнес-процессов;
- просмотр и анализ различной неструктурированной статистической информации;
- автоматизированная обработка анкет и опросников.

Основными достоинствами ELK-стека считаются следующие:

- Масштабируемость – кластер Elasticsearch (ES) расширяется «на лету» добавлением новых серверов. При этом распределение нагрузки по узлам происходит автоматически.
- Отказоустойчивость — в случае сбоя кластерных узлов данные не потеряются, а будут перераспределены, и поисковая система сама продолжит работу. Операционная стабильность достигается ведением логов на каждое изменение данных в хранилище сразу на нескольких узлах кластера.

- Гибкость поисковых фильтров, включая нечеткий поиск, возможности работы с восточными языками (китайский, японский, корейский) и мультиарендность, когда в рамках одного объекта ES можно динамически организовать несколько различных поисковых систем. Благодаря наличию встроенных анализаторов текста Elasticsearch автоматически выполняет токенизацию, лемматизацию, стемминг и прочие преобразования текста для решения NLP-задач, связанных с поиском данных.
- Управляемость ES по HTTP с помощью JSON-запросов за счет REST API и визуального веб-интерфейса Kibana.
- Универсальность – Logsatsh в потоковом режиме работает одновременно со множеством разных источников данных (СУБД, файлы, системные логи, веб-приложения и пр.), фильтруя и преобразуя их для отправки в хранилище ES. А NoSQL-природа Elasticsearch (отсутствие схемы) позволяет загружать в него JSON-объекты, которые автоматически индексируются и добавляются в базу поиска. Это позволяет ускорить прототипирование поисковых Big Data решений.

При том, что ELK позволяет быстро выполнять полнотекстовый поиск и анализировать неструктурированные данные в режиме real-time, эта Big Data система обладает **следующими недостатками:**

- высокое потребление ресурсов (CPU и RAM) из-за использования JVM – Elasticsearch и Logstash написаны на Java. На практике это может привести к недостаточной производительности на высоких нагрузках. Впрочем, показатели скорости – это субъективный вопрос.
- сложность QueryDSL – внутреннего языка запросов к базе ES;
- проблемы с информационной безопасностью – Elasticsearch не имеет никакой встроенной системы авторизации и ограничения прав доступа. После установки движок по умолчанию связывается с портом 9200 на все доступные интерфейсы, что открывает доступ к базе данных. Поэтому БД необходимо обязательно защищать паролем. Именно с этим связана одна из крупных утечек информации, случившаяся в 2019 году в банковской сфере. В открытом доступе был обнаружен кластер Elasticsearch американской компании Ascencion, на котором была расположена база данных объемом 51 ГБ с 24 млн банковских документов, включая записи о займах и ипотечных кредитах клиентов крупных банков и финансовых организаций (CitiFinancial, Wells Fargo, CapitalOne и др). Также в сеть утекли

персональные данные клиентов: полные имена, сферы деятельности, даты рождения, адреса, телефоны, номера социального страхования и банковских счетов, а также информация о кредитах.

Установка Elasticsearch

Для установки одиночного инстанса с полным набором компонентов ELK необходимы следующие системные ресурсы.

Таблица 4. Системные требования для ELK Stack

	минимальные	рекомендуемые
CPU	2	4+
Memory	6 Gb	8+ Gb
Disk	10 Gb	10+ Gb

Устанавливаем ядро системы по сбору логов - Elasticsearch. Его установка очень проста за счет готовых пакетов под все популярные платформы.

Centos 7 / 8

Копируем публичный ключ репозитория:

 # rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch

Подключаем репозиторий Elasticsearch:

 # mcedit /etc/yum.repos.d/elasticsearch.repo

[elasticsearch]

name=Elasticsearch repository for 7.x packages

baseurl=https://artifacts.elastic.co/packages/7.x/yum

gpgcheck=1

gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch

enabled=0

autorefresh=1

type=rpm-md

Приступаем к установке еластика:

 # yum install --enablerepo=elasticsearch elasticsearch

В завершении установки добавим elasticsearch в автозагрузку и запустим его с дефолтными настройками:

 # systemctl daemon-reload

 # systemctl enable elasticsearch.service

 # systemctl start elasticsearch.service

Проверяем, запустилась ли служба:

 # systemctl status elasticsearch.service

Проверим теперь, что elasticsearch действительно нормально работает. Выполним к нему простой запрос о его статусе.

```
# curl 127.0.0.1:9200
{
  "name" : "centos8",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "XSsvp-nvTxi68TaEFN_OUg",
  "version" : {
    "number" : "7.15.1",
    "build_flavor" : "default",
    "build_type" : "rpm",
    "build_hash" : "83c34f456ae29d60e94d886e455e6a3409bba9ed",
    "build_date" : "2021-10-07T21:56:19.031608185Z",
    "build_snapshot" : false,
    "lucene_version" : "8.9.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Все в порядке, сервис реально запущен и отвечает на запросы.

Ubuntu / Debian

Копируем себе публичный ключ репозитория:

 # wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -

Если у вас нет пакета apt-transport-https, то надо установить:

 # apt install apt-transport-https

Добавляем репозиторий Elasticsearch в систему:

 # echo "deb https://artifacts.elastic.co/packages/7.x/apt stable main" | sudo tee /etc/apt/sources.list.d/elastic-7.x.list

Если вы выполняете установку с российских ip адресов, то доступ к репозиториям elastic будет заблокирован. Используйте зеркало Yandex.

 # echo "deb [trusted=yes] https://mirror.yandex.ru/mirrors/elastic/7/ stable main" | sudo tee /etc/apt/sources.list.d/elastic-7.x.list

Устанавливаем Elasticsearch на Debian или Ubuntu:

 # apt update && apt install elasticsearch

После установки добавляем elasticsearch в автозагрузку и запускаем.

 # systemctl daemon-reload

 # systemctl enable elasticsearch.service

 # systemctl start elasticsearch.service

Проверяем, запустился ли он:

 # systemctl status elasticsearch.service

Проверим теперь, что elasticsearch действительно нормально работает. Выполним к нему простой запрос о его статусе.

```
# curl 127.0.0.1:9200
{
  "name" : "debian10",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "zsYgZQSeY6uvhWCkFJPiAA",
  "version" : {
    "number" : "7.15.1",
    "build_flavor" : "default",
    "build_type" : "deb",
    "build_hash" : "8ced7813d6f16d2ef30792e2fcde3e755795ee04",
    "build_date" : "2021-10-07T21:56:19.031608185Z",
    "build_snapshot" : false,
    "lucene_version" : "8.9.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Если все в порядке, то переходим к настройке Elasticsearch.

Настройка Elasticsearch

Настройки Elasticsearch находятся в файле /etc/elasticsearch/elasticsearch.yml. На начальном этапе нас будут интересовать следующие параметры:

path.data: /var/lib/elasticsearch # директория для хранения данных

network.host: 127.0.0.1 # слушаем только локальный интерфейс

По умолчанию Elasticsearch слушает localhost. Нам это и нужно, так как данные в него будет передавать logstash, который будет установлен локально. Обратите отдельное внимание на параметр для директории с данными. Чаще всего они будут занимать значительное место, иначе зачем нам Elasticsearch. Подумайте заранее, где вы будете хранить логи. Все остальные настройки можно оставить дефолтными. После изменения настроек надо перезапустить службу:

 # systemctl restart elasticsearch.service

Смотрим, что получилось:

 # netstat -tulnp | grep 9200

tcp6 0 0 127.0.0.1:9200 ::: LISTEN 19788/java*

Elasticsearch повис на локальном интерфейсе. Причем мы видим, что он слушает ipv6, а про ipv4 ни слова. Но его он тоже слушает, так что все в порядке. Переходим к установке Kibana. Если вы хотите, чтобы elasticsearch слушал все сетевые интерфейсы, настройте параметр:

network.host: 0.0.0.0

Только не спешите сразу же запускать службу. Если запустите, получите ошибку.

Чтобы ее избежать, дополнительно надо добавить еще один параметр:

discovery.seed_hosts: ["127.0.0.1", "[::1]"]

Этим мы указываем, что хосты кластера следует искать только локально.

Установка Kibana

Дальше устанавливаем web панель Kibana для визуализации данных, полученных из Elasticsearch. Тут тоже ничего сложного, репозиторий и готовые пакеты есть под все популярные платформы. Репозитории и публичный ключ для установки Kibana будут такими же, как в установке Elasticsearch. Можно устанавливать только Kibana, без всего остального. Это продукт законченный и используется не только в связке с Elasticsearch.

Centos

Импортируем ключ репозитория:

 # rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch

Добавляем конфиг репозитория:

 # mcedit /etc/yum.repos.d/kibana.repo

[kibana-7.x]

name=Kibana repository for 7.x packages

baseurl=https://artifacts.elastic.co/packages/7.x/yum

gpgcheck=1

gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch

enabled=1

autorefresh=1

type=rpm-md

Запускаем установку Kibana:

 # yum install kibana

Добавляем Кибана в автозагрузку и запускаем:

 # systemctl daemon-reload

 # systemctl enable kibana.service

 # systemctl start kibana.service

Проверяем состояние запущенного сервиса:

 # systemctl status kibana.service

По умолчанию, Kibana слушает порт 5601. Только не спешите его проверять после запуска. Кибана стартует долго. Подождите примерно минуту и проверяйте.

 # netstat -tulnp | grep 5601

tcp 0 0 127.0.0.1:5601 0.0.0.0: LISTEN 20746/node*

Ubuntu/Debian

Установка Kibana на Debian или Ubuntu такая же, как и на центос - подключаем репозиторий и ставим из deb пакета. Добавляем публичный ключ

 # wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | apt-key add -

Добавляем рпозиторий Kibana:

 # echo "deb https://artifacts.elastic.co/packages/7.x/apt stable main" | tee -a /etc/apt/sources.list.d/elastic-7.x.list

Запускаем установку Kibana:

 # apt update && apt install kibana

Добавляем Кибана в автозагрузку и запускаем:

 # systemctl daemon-reload

 # systemctl enable kibana.service

 # systemctl start kibana.service

Проверяем состояние запущенного сервиса:

 # systemctl status kibana.service

По умолчанию, Kibana слушает порт 5601. Только не спешите его проверять после запуска. Кибана стартует долго. Подождите примерно минуту и проверяйте.

 # netstat -tulnp | grep 5601

tcp 0 0 127.0.0.1:5601 0.0.0.0: LISTEN 1487/node*

Настройка Kibana

Файл с настройками Кибана располагается по пути - */etc/kibana/kibana.yml*. На начальном этапе можно вообще ничего не трогать и оставить все как есть. По умолчанию Kibana слушает только localhost и не позволяет подключаться удаленно. Это нормальная ситуация, если у вас будет на этом же сервере установлен nginx в качестве reverse proxy, который будет принимать подключения и проксировать их в Кибана. Так и нужно делать в production, когда системой будут пользоваться разные люди из разных мест. С помощью nginx можно будет разграничивать доступ, использовать сертификат, настраивать нормальное доменное имя и т.д. Если же у вас это тестовая установка, то можно обойтись без nginx. Для этого надо разрешить Кибана слушать внешний интерфейс и принимать подключения. Измените параметр server.host, указав ip адрес сервера, например вот так:

server.host: "10.20.1.23"

Если хотите, чтобы она слушала все интерфейсы, укажите в качестве адреса 0.0.0.0. После этого Kibana надо перезапустить:

 # systemctl restart kibana.service

Теперь можно зайти в веб интерфейс по адресу *http://10.20.1.23:5601*.

Можно продолжать настройку и тестирование, а когда все будет закончено, запустить nginx и настроить проксирование. Настройку nginx можно оставить на самый конец. В процессе настройки буду подключаться напрямую к Kibana. При первом запуске Kibana предлагает настроить

источники для сбора логов. Это можно сделать, нажав на *Add your data*. Изучите интерфейс и возможности этой веб панели.

Периодически вы можете видеть в веб интерфейсе предупреждение:

server.publicBaseUrl is missing and should be configured when running in a production environment. Some features may not behave correctly.

Чтобы его не было, просто добавьте в конфиг Kibana параметр:

server.publicBaseUrl: "http://10.20.1.23:5601/"

Или доменное имя, если используете его.

Установка и настройка Logstash

Logstash устанавливается так же просто, как Elasticsearch и Kibana, из того же репозитория. Просто установим его и добавим в автозагрузку.

Установка Logstash в Centos:

 # yum install logstash

Установка Logstash в Debian/Ubuntu:

 # apt install logstash

Добавляем logstash в автозагрузку:

 # systemctl enable logstash.service

Запускать пока не будем, надо его сначала настроить. Основной конфиг logstash лежит по адресу */etc/logstash/logstash.yml*. Все настройки можно по смыслу разделить по разным конфигурационным файлам в директории */etc/logstash/conf.d*. Создаем первый конфиг *input.conf*, который будет описывать прием информации с beats агентов.

```
input {
  beats {
    port => 5044
  }
}
```

Тут все просто. Указываем, что принимаем информацию на 5044 порт. Этого достаточно. Если вы хотите использовать ssl сертификаты для передачи логов по защищенным соединениям, здесь добавляются параметры ssl. Если вы будете собирать данные из закрытого периметра локальной сети, то нет необходимости использовать ssl. Теперь укажем, куда будем передавать данные. Тут тоже все относительно просто. Рисуем конфиг *output.conf*, который описывает передачу данных в Elasticsearch.

```
output {
      elasticsearch {
```

```
        hosts   => "localhost:9200"
        index   => "nginx-%{+YYYY.MM.dd}"
      }
      #stdout { codec => rubydebug }
  }
```

Что мы настроили? Передавать все данные в elasticsearch под указанным индексом с маской в виде даты. Разбивка индексов по дням и по типам данных удобна с точки зрения управления данными. Потом легко будет выполнять очистку данных по этим индексам. Последняя строка отвечает за логирование. Если ее включить, то все поступающие данные logstash будет отправлять дополнительно в системный лог. В centos это /var/log/messages. Используйте только во время отладки, иначе лог быстро разрастется дублями поступающих данных. Остается последний конфиг с описанием обработки данных. Рисуем конфиг *filter.conf*.

```
filter {
 if [type] == "nginx_access" {
   grok {
      match => { "message" => "%{IPORHOST:remote_ip} - %{DATA:user} \[%{HTTPDATE:access_time}\] \"%{WORD:http_method} %{DATA:url} HTTP/%{NUMBER:http_version}\" %{NUMBER:response_code} %{NUMBER:body_sent_bytes} \"%{DATA:referrer}\" \"%{DATA:agent}\"" }
    }
  }
  date {
      match => [ "timestamp" , "dd/MMM/YYYY:HH:mm:ss Z" ]
  }
  geoip {
      source => "remote_ip"
      target => "geoip"
      add_tag => [ "nginx-geoip" ]
  }
}
```

Первое, что делает этот фильтр, парсит логи nginx с помощью grok, если указан соответствующий тип логов, и выделяет из лога ключевые данные, которые записывает в определенные поля, чтобы потом с ними было удобно работать. С обработкой логов у новичков возникает недопонимание. В

документации к filebeat хорошо описаны модули, идущие в комплекте, которые все это и так уже умеют делать из коробки, нужно только подключить соответствующий модуль.

Модули filebeat работают только в том случае, если вы отправляете данные напрямую в Elasticsearch. На него вы тоже ставите соответствующий плагин и получаете отформатированные данные с помощью elastic ingest. Но у нас работает промежуточное звено Logstash, который принимает данные. С ним плагины filebeat не работают, поэтому приходится отдельно в logstash парсить данные. Это не очень сложно, но тем не менее. Если у вас много разрозненных данных, то отправлять их напрямую в Elasticsearch не так удобно, как с использованием предобработки в Logstash.

Для фильтра grok, который использует Logstash, есть удобный дебаггер, где можно посмотреть, как будут парситься ваши данные. Например, возьмем такую строку из лога:

180.163.220.100 - travvels.ru [05/Sep/2021:14:45:52 +0300] "GET /assets/galleries/26/1.png HTTP/1.1" 304 0 "https://travvels.ru/ru/glavnaya/" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.102 Safari/537.36"

И посмотрим, как ее распарсит правило grok, которое было использовано в конфиге выше.

%{IPORHOST:remote_ip} - %{DATA:user} \[%{HTTPDATE:access_time}\] \"%{WORD:http_method} %{DATA:url} HTTP/%{NUMBER:http_version}\" %{NUMBER:response_code} %{NUMBER:body_sent_bytes} \"%{DATA:referrer}\" \"%{DATA:agent}\"

Собственно, результат вы можете сами увидеть в дебаггере. Фильтр распарсит лог и на выходе сформирует json, где каждому значению будет присвоено свое поле, по которому потом удобно будет в Kibana строить отчеты и делать выборки. Только не забывайте про формат логов. Приведенное правило соответствует дефолтному формату main логов в nginx. Если вы каким-то образом модифицировали формат логов, внесите изменения в grok фильтр. Вы можете таким образом парсить любые логи и передавать их в еластикс. Потом на основе этих данных строить отчеты, графики, дашборды.

Дальше используется модуль date для того, чтобы выделять дату из поступающих логов и использовать ее в качестве даты документа в elasticsearch. Делается это для того, чтобы не возникало путаницы, если будут

задержки с доставкой логов. В системе сообщения будут с одной датой, а внутри лога будет другая дата. Неудобно разбирать инциденты.

В конце можно использовать geoip фильтр, который на основе ip адреса, который мы получили ранее с помощью фильтра grok и записали в поле remote_ip, определяет географическое расположение. Он добавляет новые метки и записывает туда географические данные. Для его работы используется база данных из файла */usr/share/logstash/vendor/bundle/jruby/2.5.0/gems/logstash-filter-geoip-6.0.3-java/vendor/GeoLite2-City.mmdb*. Она будет установлена вместе с logstash. Впоследствии вы скорее всего захотите ее обновлять. Раньше она была доступна по прямой ссылке, но с 30-го декабря 2019 года правила изменились. База по-прежнему доступна бесплатно, но для загрузки нужна регистрация на сайте сервиса. Регистрируемся и качаем отсюда - *https://dev.maxmind.com/geoip/geoip2/geolite2/#Download_Access*.

Передаем на сервер, распаковываем и копируем в */etc/logstash файл GeoLite2-City.mmdb*. Теперь нам нужно в настройках модуля geoip указать путь к файлу с базой. Добавляем в */etc/logstash/conf.d/filter.conf*:

```
geoip {
 database => "/etc/logstash/GeoLite2-City.mmdb"
 source => "remote_ip"
 target => "geoip"
 add_tag => [ "nginx-geoip" ]
}
```

Закончили настройку logstash. Запускаем его:

 # systemctl start logstash.service

Можете проверить на всякий случай лог */var/log/logstash/logstash-plain.log,* чтобы убедиться в том, что все в порядке. Признаком того, что скачанная geoip база успешно добавлена будет вот эта строчка в логе:

[2021-10-18T12:52:07,118][INFO][logstash.filters.geoip][main] Using geoip database {:path=>"/etc/logstash/GeoLite2-City.mmdb"}

Основные вопросы по установке и настройке ESK разобраны.

6. Менеджер очередей RabbitMQ

Что такое RabbitMQ?

RabbitMQ – это приложение для работы с очередями сообщений (message-queueing), еще его называют меседж брокер (message broker) или менеджер очередей (queue manager). Простыми словами – это программное обеспечение, в котором могут быть определены очереди к которому могут подключатся различные приложения и передавать/получать сообщения.

Сообщение может включать в себя любую информацию. Для примера у нас есть информация о процессе/задаче, которую должно начать другое приложение (или же другой сервер) или же это может быть простым текстовым сообщением. Менеджер очередей – приложение, которое сохраняет сообщение до тех пор, пока другое приложение (сервер), которому адресовано сообщение, не подключиться и не заберет (получит) сообщение из очереди. Затем приложение-получатель обработает сообщение нужным ему образом.

RabbitMq основные определения

- Producer: приложение, которое посылает сообщение
- Consumer: приложение, которое получает сообщение
- Queue: буфер, который хранит сообщения
- Message:информация, которая была отправлена от Producer k Consumer через rabbitmq
- Connection: TCP соединение между приложениями и менеджером очередей
- Channel: виртуальное соединение внутри соединения. Когда вы публикуете или получаете сообщения через очередь – это все делается в канале.
- Exchange: получает сообщение от поставщика и отправляет его в очередь. Зависит от типа.
- Binding: связь между очередью и обработчиком сообщений
- Routing key: ключ, на который смотрит обработчик и решает в какую очередь перенаправить сообщение
- AMQP: Advanced Message Queuing Protocol – протокол, используемый для обмена сообщениями в rabbitmq
- Users: возможность подключится к брокеру сообщений с помощью имени пользователя и пароля. Каждый пользователь может иметь

права доступа, такие как чтение, запись управление привилегиями. Пользователи так же могут иметь привилегии на определенный виртуальных хост.

- Vhost, virtual host: способ разделения приложений используя один и тот же экземпляр RabbitMQ. У разных пользователей могут быть разные права доступа к различным виртуальным хостам и очередям, так же обработчики сообщений могут существовать только в одном.

RabbitMQ пример

RabbitMQ может выступать как прослойка между несколькими сервисами (приложениями, серверами). Может использоваться для уменьшения нагрузки и ускорение отклика веб-приложения, поскольку задачи, которые обычно занимают довольно много времени, могут быть делегированы третьей стороне, единственной задачей которой является их выполнение.

В этом примере RabbitMQ, мы рассмотрим сценарий, когда веб приложение позволяет пользователям загружать информацию на сайт. Сайт будем обрабатывать эту информацию, генерировать PDF и отправлять обратно пользователю на электронную почту.

Обработка информации, генерация PDF и отправка email займет несколько секунд и это покажет вам как можно использовать очереди сообщений.

Когда пользователь введет информацию в веб интерфейс, приложение создаст задание на генерацию PDF и вся информация в сообщении и сообщение пользователя будут помещены в очередь определенную в RabbitMQ.

Базовая архитектура очереди сообщений довольно-таки простая, клиентское приложение (Producer) создает сообщение и доставляет его в меседж брокер. Другое приложение (Consumer) получатель подключается к очереди и подписывается на сообщение которые должно обработать. Ваше приложение может быть как поставщиком сообщений, так и получателем или же и тем и другим одновременно. Сообщения, размещенные в очереди, сохраняются до тех пор, пока получатель не получит их.



Рисунок 9. Принцип работы RabbitMQ

Когда и почему использовать RabbitMQ?

Очереди сообщений позволяют серверам получать/отправлять запросы быстрее вместо того, чтобы ожидать выполнения ресурсоемких процедур. Очередь сообщений также хорошо применима к задаче отправки сообщений набору получателей для снижения потребления памяти или балансировки нагрузки между воркерами.

Возвращаясь к примеру выше получатель может взять сообщение из очереди и начать обрабатывать, например PDF в тоже время когда поставщик добавляет новые сообщения в очередь. Получателем может быть совершенно другой сервер нежели поставщик, или же они могут находиться на одном и том же. Запрос может быть создан на одном языке программирования и обработан другим – они “общаются” только через сообщения, которые посылают один другому. Исходя из этого, приложения будут иметь низкую связь между отправителем и получателем.

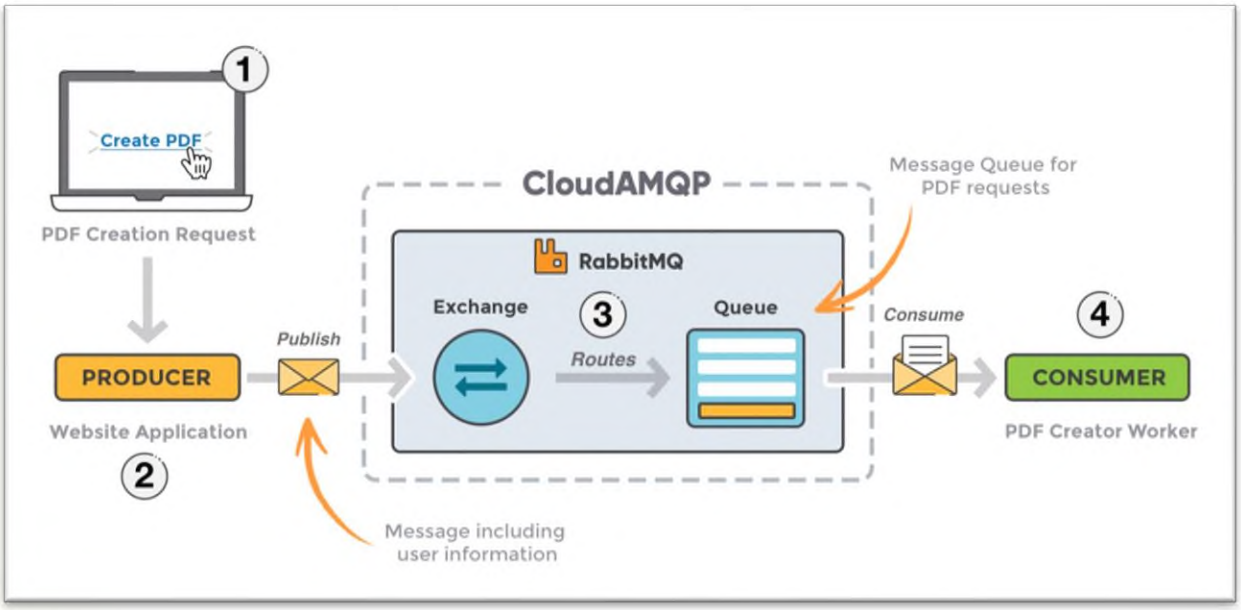


Рисунок 10. Процесс использования RabbitMQ

1. Пользователь посылает запрос на создание PDF
2. Приложение (Producer) посылает сообщение в rabbitMq, включая в запрос информацию, например имя и электронная почта.
3. Обработчик принимает сообщения от приложения поставщика и направляет его в нужную очередь сообщений.
4. Воркер обработки PDF (Consumer) получает задание – начать генерацию PDF.

Обработчик (обменник) сообщений

Сообщения публикуются в очередь не на прямую, вместо этого, поставщик шлет сообщение обработчику, который отвечает за перенаправление сообщения в нужную очередь с помощью биндинга ключей роутинга.

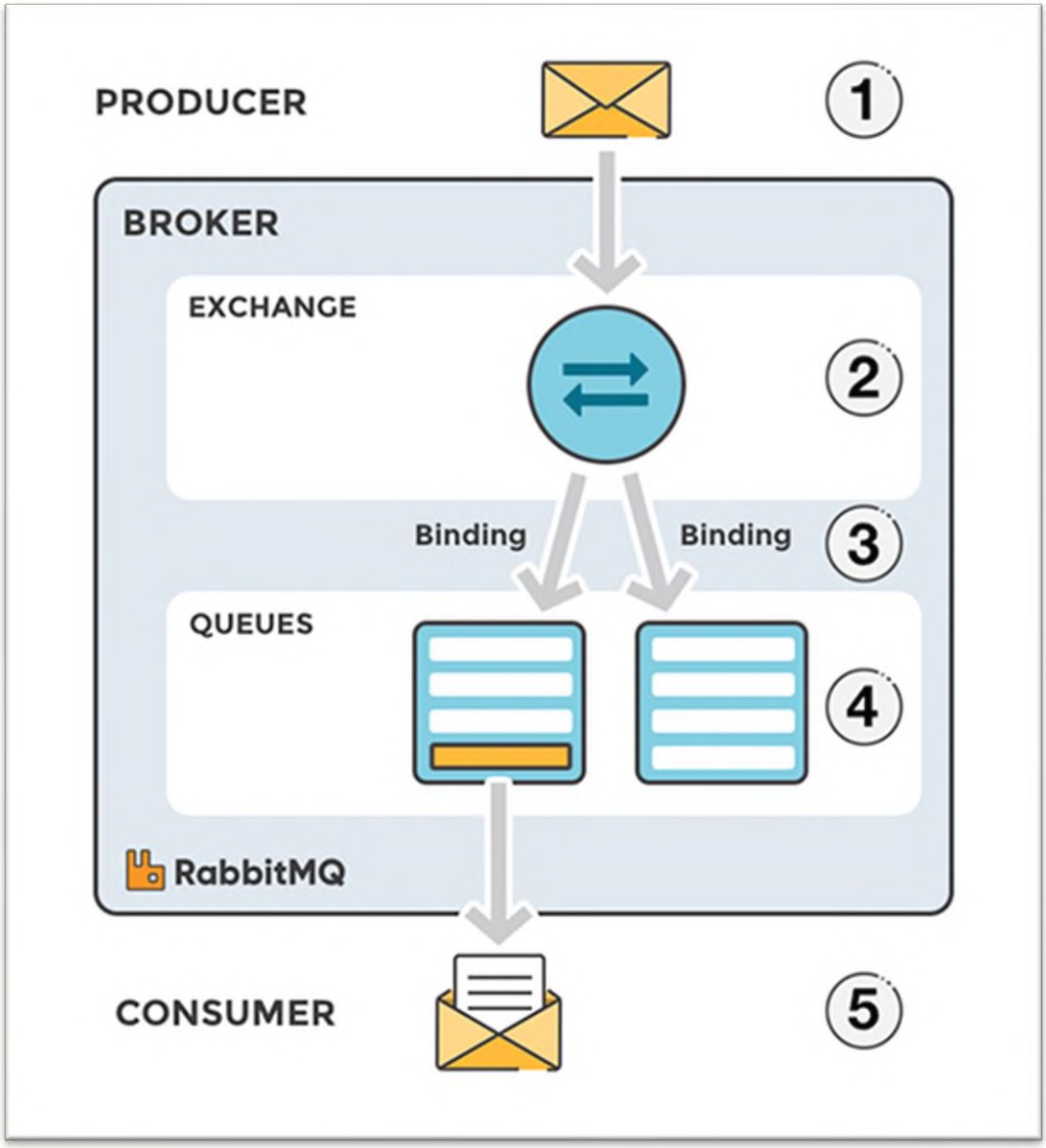


Рисунок 11. Обработчик (обменник) сообщениями

Поток сообщений в RabbitMQ

1. Поставщик публикует сообщение в обработчик. Когда создаем обработчик мы определяем его тип.
2. Обработчик получает сообщение и отвечает за его перенаправление. Обработчик берет различные атрибуты, такие как, ключ роутинга, зависимость на тип обмена и другие.
3. Создается связь между обработчиком и очередью

4. Сообщение остается в очереди до тех пор пока не будет обработано получателем.

5. Получатель обрабатывает сообщение

Типы обработчиков сообщений

- Direct: Тип пересылает сообщение в нужную очередь базируясь на ключе маршрутизации.
- Fanout: Тип перенаправляет сообщения во все связанные очереди
- Topic: Делает соответствие между ключом маршрутизации и шаблоном маршрутизации указаных в привязке
- Headers: Использует атрибуты заголовка сообщения для маршрутизации

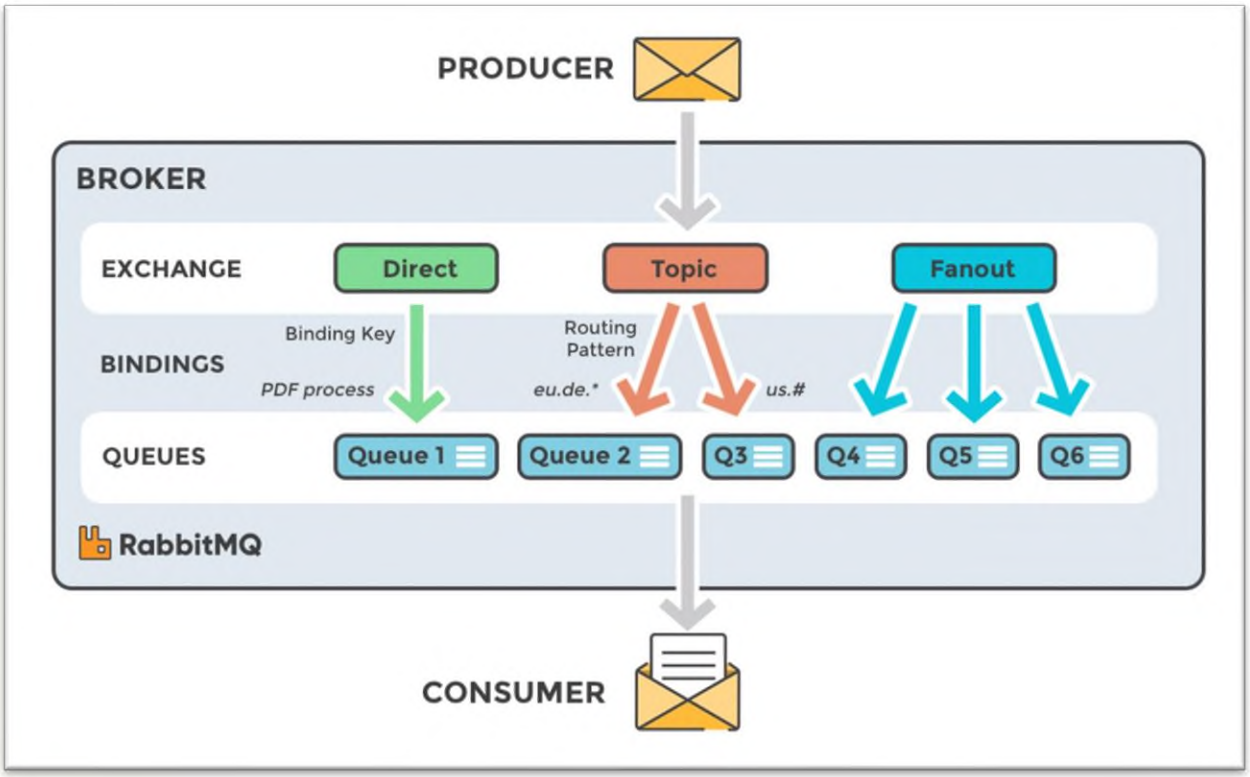


Рисунок 12. Типы обработчиков сообщений

Установка RabbitMQ в CentOS/RHEL

Прежде чем приступить к установке RabbitMQ, нужно установить зависимости программы, одной из которых является Erlang. Однако, прежде всего необходимо обновить систему и стандартные приложения; для этого запустите:

yum -y update

Для установки Erlang используйте команды:

 # Add and enable relevant application repositories:

 # Note: We are also enabling third party remi package repositories.

wget http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm

wget http://rpms.famillecollet.com/enterprise/remi-release-6.rpm

sudo rpm -Uvh remi-release-6.rpm epel-release-6*.rpm*

 # Finally, download and install Erlang:

yum install -y erlang

Теперь можно установить RabbitMQ:

 # Download the latest RabbitMQ package using wget:

wget http://www.rabbitmq.com/releases/rabbitmq-server/v3.2.2/rabbitmq-server-3.2.2-1.noarch.rpm

 # Add the necessary keys for verification:

rpm --import http://www.rabbitmq.com/rabbitmq-signing-key-public.asc

 # Install the .RPM package using YUM:

yum install rabbitmq-server-3.2.2-1.noarch.rpm

Установка RabbitMQ в Ubuntu 13/Debian 7

Процесс установки RabbitMQ в Ubuntu/Debian подобен установке в CentOS.

Для начала нужно обновить стандартные пакеты:

apt-get update

apt-get -y upgrade

Включите репозиторий приложения RabbitMQ:

echo "deb http://www.rabbitmq.com/debian/ testing main" >> /etc/apt/sources.list

Добавьте ключ проверки пакета:

curl http://www.rabbitmq.com/rabbitmq-signing-key-public.asc | sudo apt-key add -

Снова обновите систему:

apt-get update

Теперь можно загрузить и установить RabbitMQ:

sudo apt-get install rabbitmq-server

Чтобы при запуске было обработано максимальное количество подключений, откройте и отредактируйте в nano следующий конфигурационный файл:

sudo nano /etc/default/rabbitmq-server

Раскомментируйте строку limit (просто удалите символ #), а затем сохраните и закройте файл (CTRL+X + Y).

Управление RabbitMQ

Как говорилось ранее, брокер RabbitMQ очень прост в использовании. В данном разделе приведены инструкции по управлению и настройке RabbitMQ.

Включение консоли управления

Консоль управления RabbitMQ (RabbitMQ Management Console) – это один из доступных плагинов, позволяющий мониторить процессы сервера RabbitMQ через графический пользовательский веб-интерфейс.

При помощи этой консоли можно:

- Управлять обменом сообщениями, очередями сообщений, подключениями и пользователями;
- Отслеживать очереди сообщений, соединения и скорость передачи сообщений;
- Отправлять и получать сообщения;
- Отслеживать процессы Erlang и использование памяти;
- И многое другое.

Чтобы включить консоль RabbitMQ, запустите команду:

sudo rabbitmq-plugins enable rabbitmq_management

Теперь можно открыть консоль при помощи любого удобного браузера:

http://[IP_адрес_сервера]:15672/

Стандартные имя и пароль – guest.

Примечание: Запустив консоль после запуска сервиса, не забудьте перезапустить его, чтобы обновить настройки.

Управление RabbitMQ в CentOS/RHEL

После установки приложение RabbitMQ не будет запускаться автоматически при загрузке системы.

Чтобы настроить автозапуск RabbitMQ, выполните:

chkconfig rabbitmq-server on

Для запуска, остановки, перезапуска и проверки состояния используйте команды:

Запуск:

/sbin/service rabbitmq-server start

Остановка:

/sbin/service rabbitmq-server stop

Перезапуск:

/sbin/service rabbitmq-server restart

Проверка статуса:

/sbin/service rabbitmq-server status

Управление RabbitMQ в Ubuntu/Debian

Чтобы запустить, остановить перезапустить и проверит статус приложения в Ubuntu и Debian, используйте:

Запуск:

service rabbitmq-server start

Остановка:

service rabbitmq-server stop

Перезапуск:

service rabbitmq-server restart

Проверка статуса:

service rabbitmq-server status

Готово! Теперь на сервере есть готовый к работе брокер сообщений.

Установка RabbitMQ на Windows

Рассмотрим, как устанавливать RabbitMQ на Windows

1. Скачать и установить последнюю версию Erlang

https://www.erlang.org/downloads

2. Скачать и установить последнюю версию RabbitMQ

https://www.rabbitmq.com/install-windows.html

3. В установленной директории RabbitMQ создать две папки:

conf — для конфига —

C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.3.5\conf

base — база данных и логи —

C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.3.5\base

В папку conf положить конфиг файл (взять из etc) и переименовать его в rabbitmq.confi

4. Установить переменные окружения для

— конфиг файла — RABBITMQ_CONFIG_FILE

C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.3.5\conf\rabbitmq

— логов и базы данных — RABBITMQ_BASE

C:\Program Files (x86)\RabbitMQ Server\rabbitmq_server-3.3.5\base

Детали здесь *https://www.rabbitmq.com/relocate.html*

5. Проверить не занят ли порт по умолчанию командой

netstat -ano | find "5672"

Последнее число при выводе покажет ID процесса, который занял порт

Если порт занят, заменить на другой в конфиге rabbitmq.config:

{tcp_listeners, [{"127.0.0.1", 5673},{"::1", 5673}]}

6. Перезапустить RabbitMQ сервис, чтобы изменения вступили в силу

7. Запустить сервис (из папки sbin)

rabbitmq-service.bat

8. Проверить статус (из папки sbin)

rabbitmqctl.bat status

9. Установить менеджмент плагин командой

rabbitmq-plugins.bat enable rabbitmq_management

10. Рестартуем сервис RabbitMQ

rabbitmq-service.bat stop

rabbitmq-service.bat start

11. Заходим в админку RabbitMQ:

http://localhost:15672

(guest/guest)

7. Сбор и анализ ошибок при помощи Sentry

Какие проблемы поможет решить интеграции с Sentry:

- Сбор информации об ошибках, с детализацией как часто, у кого и когда она происходила. У нас будет статистика по каждой ошибке.
- Уведомление обо всех новых выявленных ошибках. Появилась ошибка в системе, sentry нас об этом уведомляет (один раз, а не заваливает сообщениями каждый раз, когда ошибка произошла)
- Моментальный поиск, отбор информации по ошибкам
- Интеллектуальная группировка сообщений об ошибках.

Sentry – это бесплатный инструмент, благодаря которому мы можем не нервничать при запуске очередного релиза, быстро находить причины возникших проблем, устранять проблемы, которые портят жизнь пользователям раньше, чем о них вам сообщат тестировщики, или helpdesk.

Преимущества использования Sentry:

- не нервничать при размещении приложений на боевом сервере,
- быстро находить причины возникших проблем,
- устранять баги раньше, чем о них вам сообщат тестировщики, коллеги из саппорта, пользователи, ПМ или директор,
- выявлять незаметные остальной команде проблемы, которые портят жизнь пользователям и снижают эффективность вашего продукта,
- бесплатен,
- легко интегрируется в проект,
- ловит ошибки и в браузере пользователя, и на вашем сервере.

Основные возможности:

- Список ошибок обновляется в режиме реального времени,
- Если ошибка была помечена как решенная и появилась снова, то она снова создается и учитывается в отдельном потоке,
- Ошибки группируются и отображаются в порядке частоты появления,
- Ошибки можно фильтровать по статусам, источнику логгирования, уровню логгирования, имени сервера и т.д.

Для начала необходимо установить Sentry. Приведенный ниже пример сделан с использованием облачного сервера Sentry - sentri.oi

1. Создаем свой проект и команду

Необходимо зарегистрироваться на sentry.io по ссылке *https://sentry.io/signup/*. После того как мы это сделали, можно приступить к созданию своего проекта.

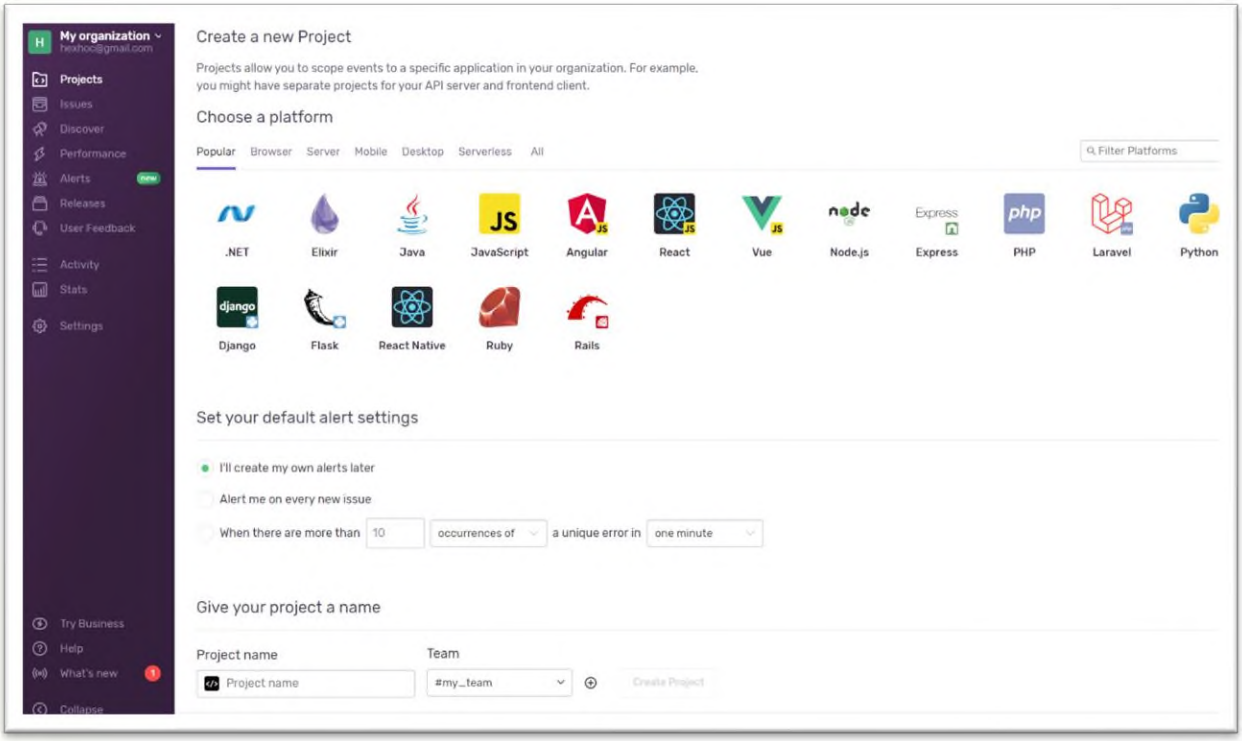


Рисунок 13. Интерфейс Sentry

При создании нового проекта нас будут настоятельно просить выбрать платформу, на которой мы ведем его разработку.

Указываем имя, и указываем команду, которая им занимается. Людей из созданной команды мы можем назначать ответственным по выявленным проблемам, так же все участникам команды будут приходить письменное уведомление по новым ошибкам.

Далее мы переходим в настройки нашего проекта. В них нам необходимо получить наш public key

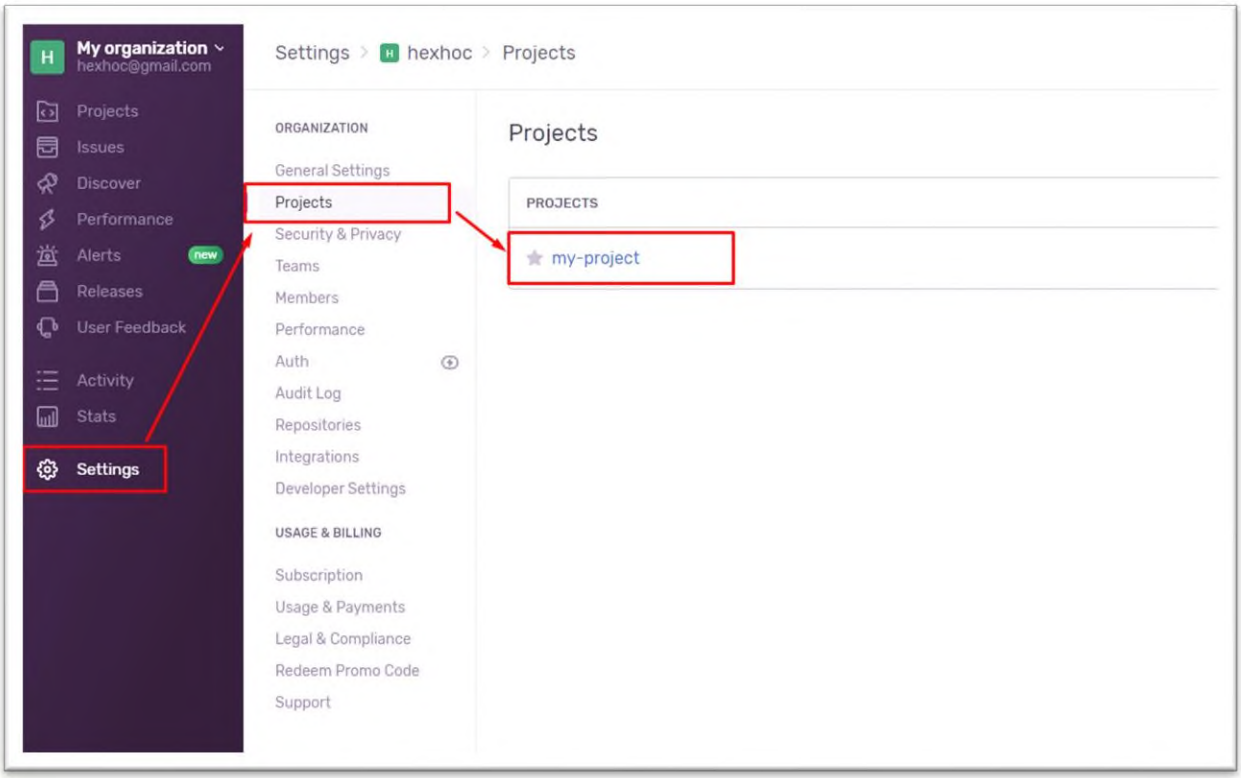


Рисунок 14. Настройки проекта

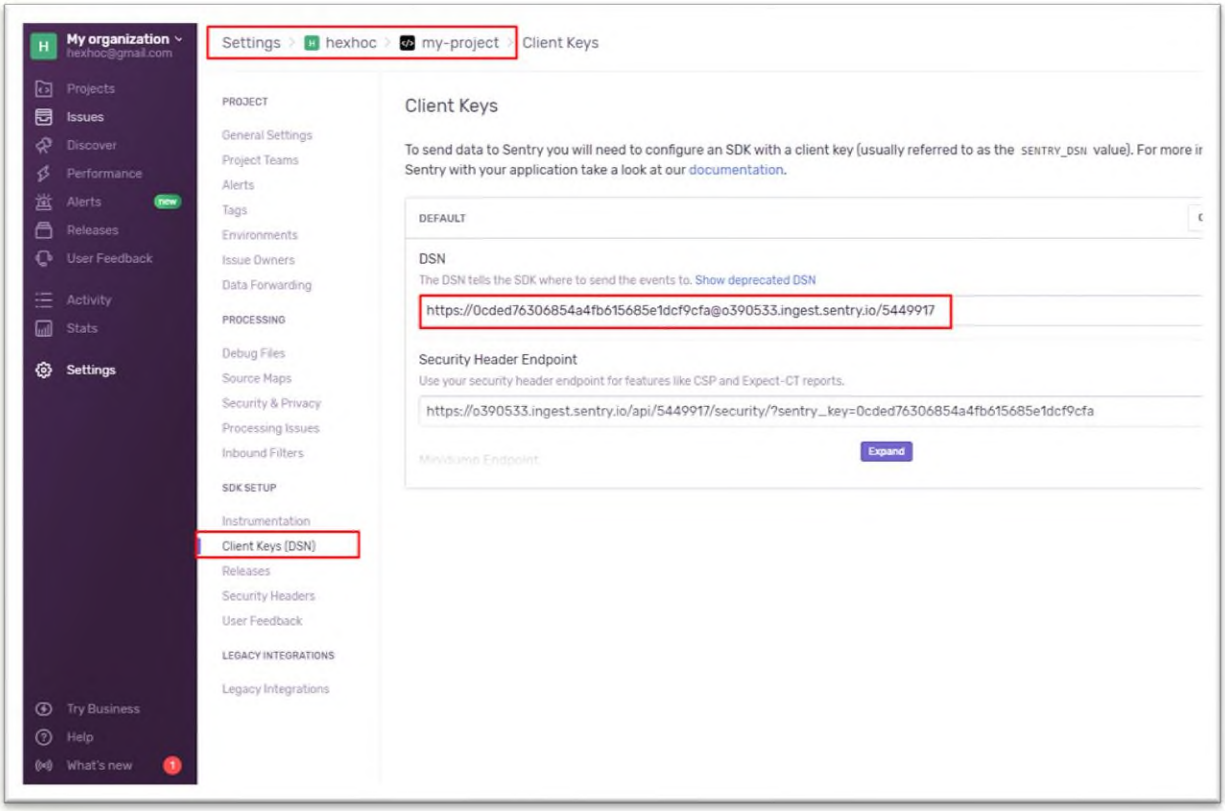


Рисунок 15. Получение ключа

Теперь внимание, там мы увидим строку DSN примерно следующего содержания:

https://0cded76306854a4fb615685e1dcf9cfa@o390533.ingest.sentry.io/5449917

В этой строке содержаться 3 нужных нам параметра:

1. 0cded76306854a4fb615685e1dcf9cfa — это public key

2. https://0cded76306854a4fb615685e1dcf9cfa@o390533.ingest.sentry.io - а это адрес сервера, на который мы будем стучаться. В случае с сервером, развернутым у себя, обычно адрес, следующий - sentry.<companydomain.ru>

3. 5449917 - номер нашего проекта

Копируем их куда-нибудь

2. Устанавливаем параметры подключения

Теперь устанавливаем http подключение к серверу sentry из 1С

Процедура ВыгрузитьЖурналРегистрацииBSentry() Экспорт

Код:

```
Процедура ВыгрузитьЖурналРегистрацииBSentry() Экспорт

    //Устанавливаем параметры подключения
    ПараметрыПодключения = Новый Соответствие;
    //ПараметрыПодключения.Вставить("User-Agent", "sentry-python/1.0");
    ПараметрыПодключения.Вставить("Content-Type", "application/json");
    ПараметрыПодключения.Вставить("X-Sentry-Auth", "Sentry sentry_version=7,sentry_key=0cded76306854a4fb615685e1dcf9cfa");

    //HTTPСоединение = Новый HTTPСоединение("sentry.companydomain.ru", , , , , ,Новый ЗащищенноеСоединениеOpenSSL);
    HTTPСоединение  = Новый HTTPСоединение("0cded76306854a4fb615685e1dcf9cfa@o390533.ingest.sentry.io", , , , , ,Новый ЗащищенноеСоединениеOpenSSL);
    HTTPЗапрос      = Новый HTTPЗапрос("/api/5449917/store/", ПараметрыПодключения);

    МассивУровнейЖурнала = Новый Массив;
    МассивУровнейЖурнала.Добавить(УровеньЖурналаРегистрации.Ошибка);
```

```
МассивУровнейЖурнала.Добавить(УровеньЖурналаРегистрации.Предупреждение);

Для каждого УровеньЖурнала из МассивУровнейЖурнала Цикл
    ОтборЖурнала = Новый Структура();
    ОтборЖурнала.Вставить("ДатаНачала", ТекущаяДата()-3600);
    ОтборЖурнала.Вставить("Уровень", УровеньЖурнала);
    УровеньЖурналаSentry = ПолучитьУровень(УровеньЖурнала);

    СобытияЖурналаТекущийДень = Новый ТаблицаЗначений;

ВыгрузитьЖурналРегистрации(СобытияЖурналаТекущийДень,ОтборЖурнала,,,);

СобытияЖурналаТекущийДень.Сортировать("ПредставлениеМетаданных, ПредставлениеСобытия");

    Для каждого СтрокаЖурнала из СобытияЖурналаТекущийДень цикл

        Тело = СформироватьТелоСообщенияSentry(СтрокаЖурнала, УровеньЖурналаSentry);

HTTPЗапрос.УстановитьТелоИзСтроки(СформироватьJSON(Тело));

        HTTPОтвет = HTTPСоединение.ОтправитьДляОбработки(HTTPЗапрос);
        Ответ = HTTPОтвет.ПолучитьТелоКакСтроку();

    КонецЦикла;

КонецЦикла;
```

КонецПроцедуры

Разберем параметры подключения:

В качестве значения для "X-Sentry-Auth" в документации нам предлагают указать следующие данные как в примере ниже:

Sentry sentry_version=7,

sentry_timestamp=1329096377,

sentry_key=b70a31b3510c4cf793964a185cfe1fd0,

sentry_secret=b7d80b520139450f903720eb7991bf3d,

sentry_client=sentry-python/1.0

Подробнее:

Sentry sentry_version - (обязательный) - текущая используемая версия sentry

sentry_timestamp - (не обязательный) - время в формате timestamp которое представляем время, в которое произошло наше событие

sentry_key - (обязательный) - тот самый public key который мы скопировали ранее

sentry_secret - (не обязательный) - Секретный ключ. Этот параметры не используется и будет удален в будущих версиях sentry

sentry_client - (необязательный) - произвольная строка в формате имя_клиента/версия_клиента которая идентифицирует платформу, на которой ведется разработка (например python/1.0)

Для работы нам достаточно указать только "Sentry sentry_version=7,sentry_key=bd945ed6d56e401da41a1f9262efc20e"

В качестве ресурса, к которому идет обращение, используется строка /api/<НомерПроекта>/store/

Номер проекта мы скопировали чуть ранее

На этом настройка параметров закончена.

3. Формируем сообщение в Sentry

Теперь приступим к разбору свойств сообщения, которое мы отправляем в Sentry

- timestamp - дата в формате timestamp, определяет в какое время событие было создано в sentry
- logger - имя логгера, который создал текущее событие
- platform - платформа, на которой возникло событие. Оставляем по умолчанию Other

- lever - аналог уровня журнала регистрации, имеет два значения - error, warning
- transaction - номер транзакции, в которой возникла ошибка
- server_name - имя рабочего сервера
- release - версия релиза (версия конфигурации)
- dist - имя дистрибутива (имя базы)
- environment – это режим совместимости. Лучше указать версию платформы, под которой был запущен сеанс.
- message - комментарий. Текстовое описание ошибки
- tags - произвольные теги, для удобства поиска и анализа ошибок
- extra - любая доп. информация о событии.
- exception - место, в котором возникла ошибка. Группировка ошибок происходит по полю exception. Если ошибка произошла в конкретной строке кода, конкретного объекта, то она будет объединена с такими ошибкам, а не выделена в отдельную ошибку.
- context - контекст, при котором возникла ошибка. Оборудование, операционная система, браузер, версия приложения, номер сеанса, соединения и т. п. Подробней тут https://develop.sentry.dev/sdk/event-payloads/contexts
- breadcrumbs - Перечень вызовов, которые предшевствовали месту возникновения ошибки. Начиная с платформы с версии 8.3.15 мы можем получать полный стек вызовов перед возникновением ошибки. В моем примере этот функционал не используется.
- https://develop.sentry.dev/sdk/event-payloads/breadcrumbs/
- user - Собственно сам пользователь, под которым произошло событие

4. Profit

Вот и все. Остается только создать регламентное задание, которое будет на постоянной основе отправлять данные в Sentry.

Теперь у нас имеется более чем полная информация по ошибкам, местам их возникновения и прочая статистика. А также уведомления о новых ошибках.

8. Инцидент-менеджмент Zabbix

Что такое Zabbix?

Zabbix — это универсальный инструмент мониторинга, способный отслеживать динамику работы серверов и сетевого оборудования, быстро реагировать на внештатные ситуации и предупреждать возможные проблемы с нагрузкой. Система мониторинга Zabbix может собирать статистику в указанной рабочей среде и действовать в определенных случаях заданным образом

В этой обзорной статье расскажем об основных принципах и ключевых инструментах, на которых построена универсальная система мониторинга Zabbix.

Архитектура Zabbix

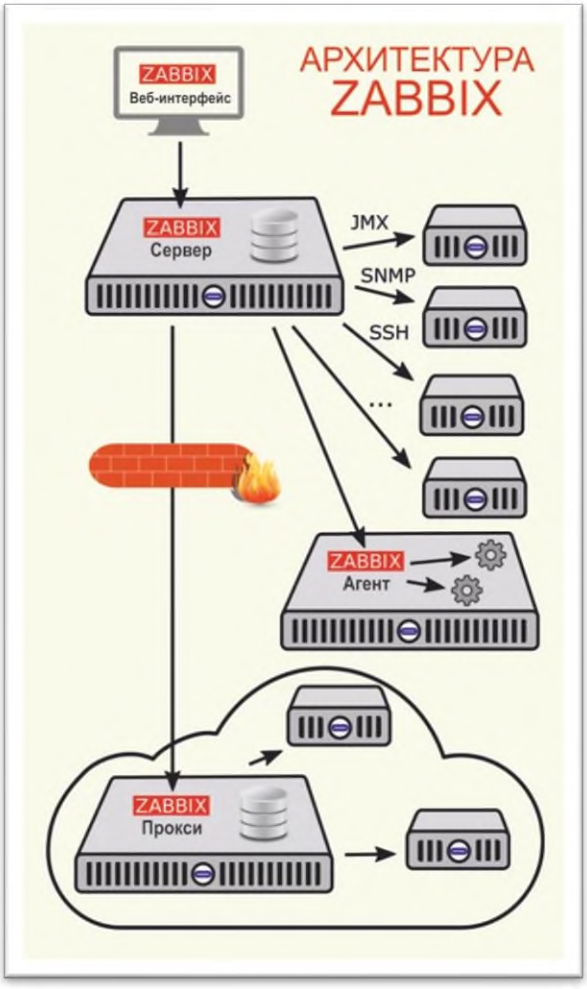


Рисунок 16. Архитектура Zabbix

У Zabbix есть 4 основных инструмента, с помощью которых можно мониторить определенную рабочую среду и собирать о ней полный пакет данных для оптимизации работы.

- Сервер — ядро, хранящее в себе все данные системы, включая статистические, оперативные и конфигурацию. Дистанционно управляет сетевыми сервисами, оповещает администратора о существующих проблемах с оборудованием, находящимся под наблюдением.
- Прокси — сервис, собирающий данные о доступности и производительности устройств, который работает от имени сервера. Все собранные данные сохраняются в буфер и загружаются на сервер. Нужен для распределения нагрузки на сервер. Благодаря этому процессу можно уменьшить нагрузку на процессор и жесткий диск. Для работы прокси Zabbix отдельно нужна база данных.
- Агент — программа (демон), которая активно мониторит и собирает статистику работы локальных ресурсов (накопители, оперативная память, процессор и др.) и приложений.
- Веб-интерфейс — является частью сервера системы и требует для работы веб-сервер. Часто запускается на том же физическом узле, что и Zabbix.

Основные возможности

Функционал включает в себя общие проверки для наиболее распространенных сервисов, в том числе СУБД, SSH, Telnet, VMware, NTP, POP, SMTP, FTP и т.д. Если стандартных настроек системы недостаточно, их можно изменить самостоятельно или же пользоваться дополнением через API.

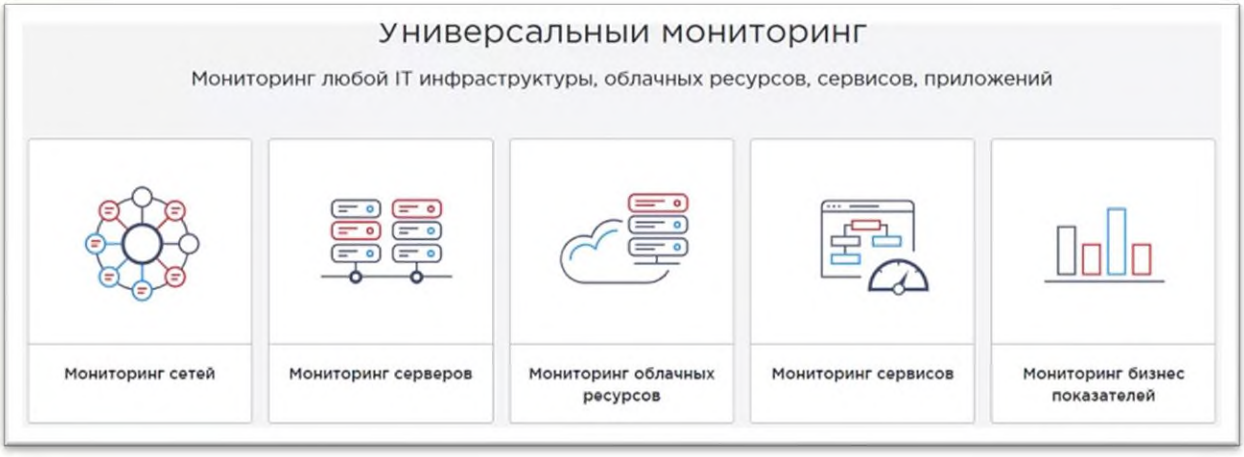


Рисунок 17. Универсальный мониторинг

Стандартные функции системы:

- Контроль нагрузки на процессор, касается и отдельных процессов.
- Сбор данных об объеме свободной оперативной и физической памяти.
- Мониторинг активности жесткого диска.
- Мониторинг сетевой активности.
- Пинг для проверки доступности узлов в сети.

Проверки

Для описания системы мониторинга Zabbix существует два ключевых понятия:

- Узлы сети — рабочие устройства и их группы (серверы, рабочие станции, коммутаторы), которые необходимо проверять. С создания и настойки узлов сети обычно начинается практическая работа с Zabbix.
- Элементы данных — набор самостоятельных метрик, по которым происходит сбор данных с узлов сети. Настройка элементов данных производится на вкладке «Элемент данных» или в автоматическом режиме — через подключение шаблона.

Сам Zabbix-агент способен отражать текущее состояние физического сервера, собирая совокупность данных. У него достаточно много метрик. С их помощью можно проверить загруженность ядра (Processor load), время ожидания ресурсов (CPU iowait time), объем системы подкачки (Total swap space) и многое другое.

В Zabbix существует целых 17 способов, дающих возможность собирать информацию. Указанные ниже, входят в число наиболее часто применяемых.

- Zabbix agent (Zabbix-агент) — сервер собирает информацию у агента самостоятельно, подключаясь по определенному интервалу.
- Simple check (Простые проверки) — простые операции, в том числе пинг.
- Zabbix trapper (Zabbix-траппер) — сбор информации с трапперов, представляющих собой мосты между используемыми сервисами и самой системой.
- Zabbix aggregate (Zabbix-комплекс) — процесс, предусматривающий сбор совокупной информации из базы данных.

- SSH agent (SSH-агент) — система подключается по SSH, использует указанные команды.
- Calculate (Вычисление) — проверки, которые система производит, сопоставляя имеющиеся данные, в том числе после предыдущих сборов.

У проверок есть заданные шаблоны (Templates), которые упрощают создание новых. Кроме обычных операций существует возможность регулярно проверять доступность веб-сервера с помощью имитации запросов браузера.

Проверка через пользовательский параметр

Чтобы выполнить проверку через агента, нужно прописать соответствующую команду в конфигурационный файл Zabbix-агента в качестве пользовательского параметра (UserParameter). Сделать это можно с помощью выражения следующего вида:

UserParameter=<ключ>,<команда>

Помимо самой команды, приведенный синтаксис содержит уникальный (в пределах узла сети) ключ элемента данных, который надо придумать самостоятельно и сохранить. В дальнейшем ключ можно использовать для ссылки на команду, внесенную в пользовательский параметр, при создании элемента данных.

Пример

UserParameter=ping,echo 1

С помощью данной команды можно настроить агент на постоянное возвращение значения «1» для элемента данных с ключем «ping».

Триггеры

Это логические выражения со значениями FALSE, TRUE и UNKNOWN, которые используются для обработки данных. Их можно создать вручную. Перед использованием триггеры возможно протестировать на произвольных значениях.

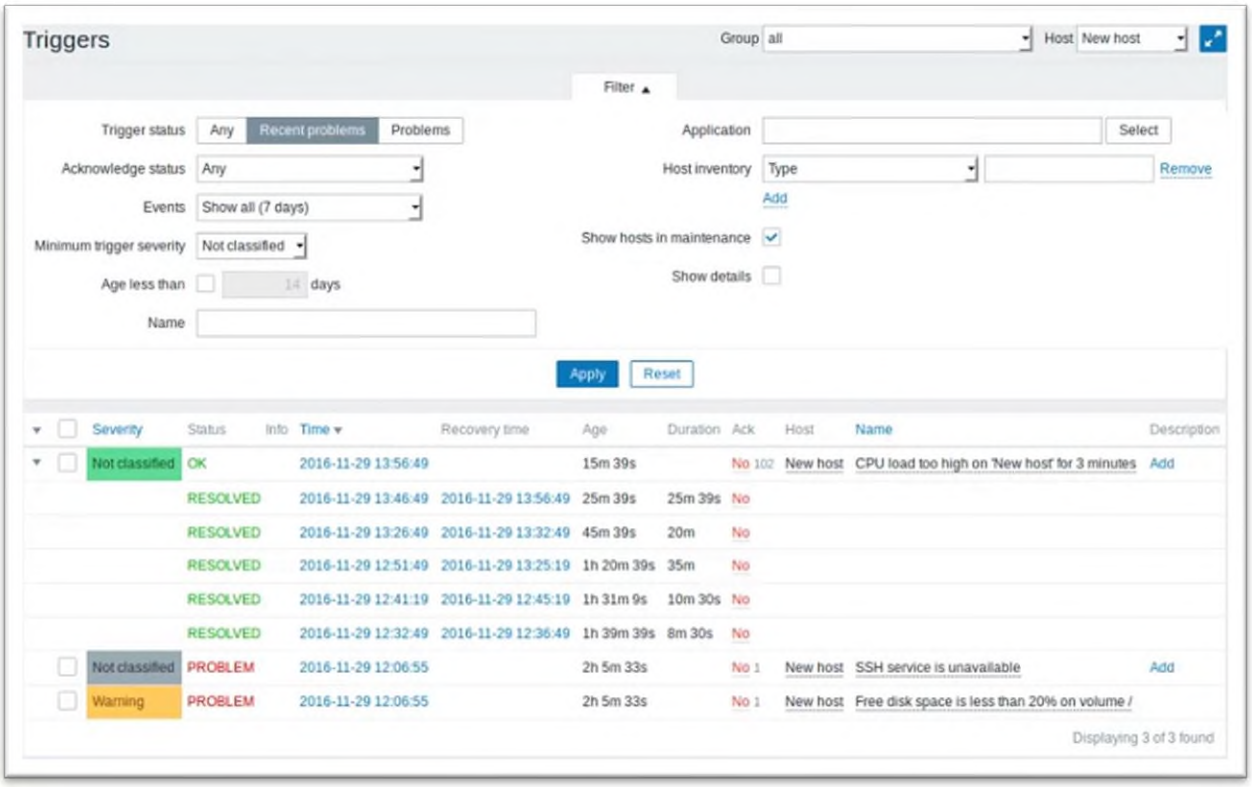


Рисунок 18. Триггеры

У каждого триггера существует уровень серьезности угрозы, который маркируется цветом и передается звуковым оповещением в веб-интерфейсе.

- Не классифицировано (Not classified) — серый.
- Информация (Information) — светло-синий.
- Предупреждение (Warning) — жёлтый.
- Средняя (Average) — оранжевый.
- Высокая (High) — светло-красный.
- Чрезвычайная (Disaster) — красный.

Некоторые функции триггеров

- abschange — абсолютная разница между последним и предпоследним значением (0 — значения равны, 1 — не равны).
- avg — среднее значение за определенный интервал в секундах или количество отсчетов.
- delta — разность между максимумом и минимумом с определенным интервалом или количеством отсчетов.
- change — разница между последним и предпоследним значением.
- count — количество отсчетов, удовлетворяющих критерию.

- date — дата.
- dayofweek — день недели от 1 до 7.
- diff — у параметра есть значения, где 0 — последнее и предпоследнее значения равны, 1 — различаются.
- last — любое (с конца) значение элемента данных.
- max\min — максимум и минимум значений за указанные интервалы или отсчеты.
- now — время в формате UNIX.
- prev — предпоследнее значение.
- sum — сумма значений за указанный интервал или количество отсчетов.
- time — текущее время в формате HHMMSS.

Триггеры обладают еще одной важной функцией для мониторинга — **прогнозированием.** Она предугадывает возможные значения и время их возникновения. Прогноз составляется на основе ранее собранных данных.

Анализируя их, триггер выявляет будущие проблемы, предупреждает администратора о возникшей вероятности. Это дает возможность предотвратить пики нагрузки на оборудование или заканчивающееся место на жестком диске.

Функционал прогнозирования добавили с обновлением системы 3.0, вышедшим в феврале 2016 года.

Действие

Действие (Action) представляет собой заданную реакцию на событие (Event). Действие может устанавливаться автоматически или вручную как для одного из событий, так и для целой группы.

Параметры действий

- Name — имя действия.
- Event source — источник события. Источниками событий служат обнаружение (Discovery Events), авторегистрация (Auto registration Events) или заданный триггер (Trigger Events).
- Enable escalations — разрешение на эскалацию событий.
- Period — период времени для шага эскалации, указывается в секундах.
- Default subject — указывается, кто извещается по умолчанию.
- Default message — стандартный текст сообщения.

- Recovery message — текст уведомления после решения проблемы.
- Recovery subject — субъект, которого извещают после операции.
- Status — статус действия, может быть «активно» и «запрещено».

Для событий, вызванных триггером или обнаружением, есть свои типы условий. Например, «Application» с операторами «=», «like» и «not like» значит, что триггер является частью указанного приложения. Или «Service type» с операторами «=», «<»и «>» предусматривает, что обнаруженный сервис совпадает с указанным.

Операции

Пользователь может указать для событий операцию или группу операций.

Параметры операций

- Step — при эскалации событий.
- Operation type — действия на определенном шаге, например, «Send message» или «Execute command».
- Event Source — источник событий.
- Send message to — отдельное сообщение (Single user) или групповое (User group).
- Default message — текст по умолчанию.
- Subject — кого оповещает система.
- Message — текст сообщения.
- Remote command — команда для удаленного управления.

Низкоуровневое обнаружение

Функция Низкоуровневого обнаружения (LLD) автоматически создает элементы и триггеры, которые позволяют отслеживать системы сервера, находящимся под наблюдением. Включение функции происходит через настройку атрибутов, которую можно сделать, пройдя по пути: «Настройка» → «Шаблоны» → «Обнаружение» (вкладка в строке с шаблоном) → вкладки «Правила обнаружения»/«Фильтры».

Что можно обнаружить

- Распространённые OID, используемые SNMP.
- Сетевые интерфейсы.
- Процессоры, их ядра.
- Файловые системы.

- Службы Windows.
- ODBC.

Дополнительные типы

Задать собственные типы низкоуровневого обнаружения возможно с применением формата JSON. Типы проверок, для которых можно указать список портов и интервал для них:

- SSH;
- LDAP;
- SMTP;
- FTP;
- HTTP;
- POP;
- NNTP;
- IMAP;
- TCP.

Если хост пропадает или обнаруживается, для события можно привязать любое действие — условия и операцию для них.

Прокси

Функция буферизации через прокси используется в том случае, когда существующая инфраструктура достаточно большая, а выделенный сервер не способен нести такую нагрузку. Прокси выступает промежуточным звеном, которое собирает информацию с агентов в буфер, а после отправляет данные на сервер.

Прокси используется еще в нескольких случаях — если агенты находятся далеко друг от друга или ограничены локальной сетью. Это сказывается на доступности агентов и величине пингов.

Для использования Zabbix прокси обязательно наличие отдельной базы данных.

Особенности веб-интерфейса

Система мониторинга Zabbix располагает удобным веб-интерфейсом, в котором сгруппированы элементы управления. Консоль предусматривает просмотр собранных данных, их настройку. Для безопасности входа и работы осуществляется автоматическое отсоединение через 30 минут пользовательского бездействия.

На главном экране всегда представлена информация о состоянии узлов сети и триггеров.

Пользователю доступны пять функциональных разделов, включая Monitoring («Мониторинг»), Inventory («Инвентарные данные»), Reports («Отчеты»), Configuration («Конфигурация») и Administration («Администрирование»).

В разделе «Конфигурации» можно найти группы хостов. По каждому элементу списка можно посмотреть более подробную информацию, например, последние события и графики данных.

Управлять шаблонами, доступными администратору, можно в соответствующем подразделе — Templates («Шаблоны»).

Установка Zabbix

Установка и настройка Zabbix-сервера на VPS начинается с подготовки рабочего окружения. Для работы потребуется минимум 2 устройства:

- Сервер. Здесь устанавливается мониторинг-система, отслеживающая сетевую и программную активность устройства. Информация записывается в базу данных.
- Клиент. Используется для управления Zabbix-сервером через веб-интерфейс.

Базовые компоненты Zabbix

- СУБД — совокупность программных модулей для управления базами данных (создания, удаления, добавления записей и т. д.). Zabbix поддерживает следующие СУБД: MySQL (InnoDB или MariaDB engine), Oracle, PostgreSQL, IBM DB2, SQLite.
- База данных (БД) — связанные между собой данные, организованные по определенной структуре в виде таблиц.
- PHP (препроцессор гипертекста) — язык программирования, используемый для построения web-приложений с использованием БД.
- Web-сервер (Apache, Nginx) — набор программных средств для работы с web-приложениями.

Сбор и передача информации с сервера на клиент выполняется по защищенному шифрованием каналу при помощи специального приложения — Zabbix-агента.

Для установки Zabbix-сервера рекомендуется использовать Linux / Unix-платформы. Для этой цели отлично подойдут операционные системы: Debian, Ubuntu и CentOS.

Готовые решения

Установка из образа

Развертывание приложения из iso-образа — готовое решение, которое значительно экономит время по сравнению с установкой вручную. Этот вариант подходит для быстрого развертывания Zabbix-сервера (MySQL/PostgreSQL) и Zabbix-прокси (MySQL/SQLite 3).

1. Для начала нужно зайти на официальный сайт приложения, где выложены архивы с готовыми решениями Zabbix под различные виртуальные платформы.
2. Выбрав нужный вариант, нужно скачать его, разархивировать и развернуть в соответствующей виртуальной машине.
3. После развертывания Zabbix, следует запустить его и залогиниться. Обычно, умолчанию логин: «root», пароль: «zabbix», либо эти параметры задаются произвольно в процессе установки.
4. Далее, нужно узнать IP-адрес, которое приложение получило командой «ip addr». Затем вписать полученный адрес в строку браузера в следующем виде: «http://<host_ip>/zabbix» (где «host_ip» — адрес установленной версии Zabbix) и нажать ввод.
5. Осуществится переход в веб-интерфейс панели управления Zabbix-сервером, где можно сделать все необходимые настройки конфигурации.

Установка Zabbix из контейнеров Docker

Zabbix интегрирован с приложением для контейнеризации Docker.

Каждый компонент Zabbix можно быстро развернуть в виде виртуального контейнера. Они доступны для скачивания на официальном сайте.

Docker-образы Zabbix поставляются для трех операционных систем

- Ubuntu;
- AlpineLinux;
- CentOS.

Установку Zabbix-сервера можно произвести несколькими основными способами.

Вариант №1

Можно сразу установить готовый образ, включающий в себя Zabbix-сервер, MySQL, Nginx. Для этого нужно выполнить команду:

docker run --name some-zabbix-appliance -p 80:80 -p 10051:10051 -d zabbix/zabbix-appliance:latest

Создастся Zabbix-экземпляр, прослушивающий 80 и 10051 порты.

Остаётся перейти по адресу: «http://<host_ip>/». В открывшемся веб-интерфейсе потребуется ввести логин – «Admin», пароль – «zabbix».

Вариант №2

Существует способ размещения Zabbix, при котором каждый компонент мониторинг-системы расположен в отдельном контейнере.

Для создаваемых контейнеров нужно создать сеть Docker:

docker network create zabbix-net

Теперь необходимо запустить контейнер с СУБД:

docker run -d --name zabbix-postgres --network zabbix-net -v /var/lib/zabbix/localtime:/etc/localtime -e POSTGRES_PASSWORD=zabbix -e POSTGRES_USER=zabbix postgres:alpine

В примере используется Postgresql.

Далее стоит разместить контейнер с Zabbix-сервером:

docker run --name zabbix-server --network zabbix-net -v /var/lib/zabbix/alertscripts:/usr/lib/zabbix/alertscripts -v /var/lib/zabbix/localtime:/etc/localtime -p 10051:10051 -e DB_SERVER_HOST="zabbix-postgres" -e POSTGRES_USER="zabbix" -e POSTGRES_PASSWORD="zabbix" -d zabbix/zabbix-server-pgsql:alpine-latest

Остаётся запустить веб-сервер Zabbix:

docker run --name zabbix-web -p 80:8080 -p 443:8443 --network zabbix-net -e DB_SERVER_HOST="zabbix-postgres" -v /var/lib/zabbix/localtime:/etc/localtime -e POSTGRES_USER="zabbix" -e POSTGRES_PASSWORD="zabbix" -e ZBX_SERVER_HOST="zabbix-server" -d zabbix/zabbix-web-nginx-pgsql:alpine-latest

В примере используется Nginx.

Остаётся перейти по адресу: «http://<host_ip>/», и войти в веб-интерфейс, воспользовавшись логином – «Admin», и паролем – «zabbix».

Ручная установка

Ниже мы покажем, как развернуть Zabbix на VDS на примере трех ОС — Debian 10, Ubuntu 20.04 и CentOS 7.

Установка на Zabbix на Debian 10

Команды, вводимые в терминале на Debian 10 (Debian 9) практически идентичны Ubuntu 20.04., но есть некоторые отличия.

Установка системы мониторинга начинается с загрузки deb-пакета Zabbix:

wget https://repo.zabbix.com/zabbix/5.0/debian/pool/main/z/zabbix-release/zabbix-release_5.0-1+buster_all.deb

Далее его необходимо установить:

dpkg -i zabbix-release_5.0-1+buster_all.deb

apt update

Далее нужно установить Zabix-сервер и агент для его мониторинга командой:

apt install zabbix-server-mysql zabbix-frontend-php zabbix-apache-conf zabbix-agent

Установка Zabbix на Ubuntu 20.04

Для начала установки Zabbix на Ubuntu загружаем и устанавливаем deb-пакет:

wget https://repo.zabbix.com/zabbix/5.0/ubuntu/pool/main/z/zabbix-release/zabbix-release_5.0-1+focal_all.deb

dpkg -i zabbix-release_5.0-1+focal_all.deb

apt update

Устанавливаем Zabix-сервер и агент:

apt install zabbix-server-mysql zabbix-frontend-php zabbix-apache-conf zabbix-agent

Установка Zabbix на CentOS 7

Установка начинается с добавления репозитория, находящегося на официальном сайте. Сделать это можно при помощи последовательности двух команд:

rpm -Uvh https://repo.zabbix.com/zabbix/5.0/rhel/7/x86_64/zabbix-release-5.0-1.el7.noarch.rpm

yum clean all

Далее нужно установить Zabix-сервер и агент:

yum install zabbix-server-mysql zabbix-agent

После этого нужно установить веб-интерфейс Zabix, последовательно выполнив ряд действий.

Установить пакет актуального софта для CentOS из репозитория Red Hat Software Collections:

yum install centos-release-scl

Отредактировать файл «/etc/yum.repos.d/zabbix.repo» и включить репозиторий «zabbix-frontend», выполнив команду:

vi /etc/yum.repos.d/zabbix.repo

Здесь следует заменить строку «enabled=0» на «enabled=1».

[zabbix-frontend]

...

enabled=1

...

Далее нужно установить пакеты веб-интерфейса Zabbix:

yum install zabbix-web-mysql-scl zabbix-apache-conf-scl



МОДУЛЬ 9. Виртуализация в DevOps

1. Основы виртуализации

2. Типы виртуализаций KVM, QEMU

3. Системы управления виртуализацией

1. Основы виртуализации

Что такое виртуализация?

Виртуализация — это предоставление набора вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию друг от друга вычислительных процессов, выполняемых на одном физическом ресурсе.

- Это механизм создания виртуального, программного, представления ресурсов без привязки к аппаратному обеспечению.
- Виртуализировать можно серверы, средства хранения данных, сетевые ресурсы, приложения и рабочие столы.

Примером использования виртуализации является возможность запуска нескольких операционных систем на одном компьютере: при том каждый из экземпляров таких гостевых операционных систем работает со своим набором логических ресурсов (процессорных, оперативной памяти, устройств хранения), предоставлением которых из общего пула, доступного на уровне оборудования, управляет хостовая операционная система — гипервизор. Также могут быть подвергнуты виртуализации сети передачи данных, сети хранения данных, платформенное и прикладное программное обеспечение.

Если говорить еще проще, то это запуск нескольких систем на базе одной вычислительной машины, при этом физические ресурсы этой машины выделяются каждой виртуальной системе независимо. Распределением ресурсов занимается хостовая операционная система — гипервизор, который выдает гостевым (виртуальным) машинам процессорную мощность, оперативную память, ресурсы хранения и сетевых коммуникаций из общего пула ресурсов.

Общие сведения о виртуализации

По своей сути, все ОС — это в общем-то и так некоторая виртуальная среда, которая предоставляется разработчику ПО, как средство реализации конечных задач. Уже давно прошло то время, когда программы писались конкретно под аппаратную часть компьютера по средствам аппаратных кодов и запросов. Сегодня же, любое приложение – это в первую очередь приложение, написанное на некотором API, который находится под управлением ОС. Задачи же ОС – предоставить данным API непосредственно доступ к аппаратным ресурсам.

Собственно, видов виртуализации существует несколько:

- Программная виртуализация;
- Аппаратная виртуализация;
- Контейнеризация или контейнерная виртуализация.

Виртуализация в свою очередь бывает полной и частичной.

Программная виртуализация – вид виртуализации, который задействует различные библиотеки ОС, транслируя вызовы виртуальной машины в вызовы ОС. (DOSBox, Virtualbox, VirtualPC). В настоящее время программная виртуализация используется не так широко. Виртуальные машины на её основе значительно менее производительные по сравнению с другими видами виртуализации.

Аппаратная виртуализация – такой вид, который предусматривает специализированную инструкцию аппаратной части, а конкретно инструкций процессора. Позволяет исполнять запросы в обход гостевой ОС, и исполнять прямо на аппаратном обеспечении. (виртуализация KVM, виртуализация XEN, Parallels, VMware, Virtualbox).

Для её работы требуется поддержка со стороны процессора. Наибольшее распространение получили технологии Intel-VT и AMD-V, в настоящее время большинство процессоров для домашних компьютеров поддерживают одну из них. Аппаратная виртуализация не получила бы такого широкого распространения, если бы не преимущества, которые обеспечивает данный подход. Эти преимущества описаны ниже.

Первое преимущество: при аппаратной виртуализации виртуальные машины управляются гипервизором напрямую, в отличие от программной виртуализации, где, например, решение о выделении памяти для виртуальной машины сначала принимает операционная система хоста, и только после подтверждения гипервизор может предоставить ей ресурсы. Благодаря этому производительность гостевых ОС значительно повышается и достигает эффективности, сравнимой с реальным компьютером с такой же конфигурацией.

Второе преимущество: так как конфигурация виртуальной машины полностью эмулируется гипервизором, установщик операционной системы не нужно модифицировать. Выбираем нужные устройства в настройках, подключаем любой стандартный установочный образ нужной операционной системы и запускаем виртуальную машину. Более того, если у вас ещё сохранился процессор с 32-битной архитектурой, с помощью аппаратной виртуализации можно настроить виртуальную машину с 64-битным процессором и установить соответствующую ОС. Независимость от

платформы хоста открывает поистине бесконечные возможности для экспериментов.

Контейнеризация или контейнерная виртуализация – это виртуализация на уровне операционной системы. Если аппаратная виртуализация полностью эмулирует оборудование и позволяет запускать любые ОС, внутри контейнера можно запустить только аналогичную хосту операционную систему. Преимуществом этого подхода является скорость, с которой создаётся контейнер — секунды, тогда как для запуска виртуальной машины счёт времени идёт на минуты. Так происходит потому, что полноценной виртуальной машине нужно сначала инициализировать всё оборудование, запустить эмуляцию и только после этого начать загружать операционную систему. При контейнеризации ОС по факту уже работает. Остаётся только создать замкнутую среду — тот самый контейнер, в котором будет запущен ещё один экземпляр операционной системы.

Контейнер представляет собой всего лишь один процесс, внутри которого выполняется операционная система. Она существует в своём собственном мире, со своей сетью, своим диском, своей файловой системой и так далее. Эту виртуализацию применяют на уровне сервисов, составляющих части программного продукта. Наиболее известные проекты: OpenVZ, Docker, LXC.

Виртуализация DevOps

DevOps сегодня является одним из ключевых факторов быстрой и последовательной разработки и развертывания программного обеспечения, поскольку он сочетает в себе новые стратегии и методологии для улучшения совместной работы команды и непрерывного предоставления качественного программного обеспечения или функций.

Кроме того, он использует технологии, которые ускоряют жизненный цикл разработки программного обеспечения и помогают одновременно повысить качество и скорость развертывания, способствуя сотрудничеству и прозрачности на протяжении всего процесса. Будь то облако, автоматизация или что-то еще, DevOps использует различные технологии и инструменты для создания ценного актива для организаций.

Еще одна известная методология, которая помогает DevOps обеспечивать быстрое, последовательное и стабильное развертывание, — это виртуализация.

Роль виртуализации в DevOps:

Основная цель DevOps — повысить скорость и качество разработки программного обеспечения, что стало возможным благодаря широкому использованию виртуализации. Виртуализация — неотъемлемая часть DevOps, затрагивающая все аспекты процесса: от разработки, тестирования и развертывания до эксплуатации, поставки и обслуживания.

С помощью этой методологии команды DevOps могут использовать виртуальные сервисы вместо производственных сервисов и проводить частое и всестороннее тестирование, даже если в архитектуре системы отсутствуют ключевые компоненты. Он имитирует поведение основных компонентов и устраняет ключевые узкие места, которые могут привести к задержке выпуска и выхода на рынок.

Другими важными областями, в которых виртуализация помогает DevOps, являются:

- Обеспечивает согласованную и стабильную среду для разработки и тестирования.
- Поскольку программное обеспечение разрабатывается в виртуальной среде, оно помогает быстрее выявлять ошибки и значительно снижать частоту сбоев при регистрации и выпуске.
- Улучшает совместимость продуктов и интероперабельность за счет одновременного тестирования на разных уровнях выпусков и исправлений, и интеграции с другими программными системами.
- Предлагает согласованную, предсказуемую и устойчивую к сбоям среду, которая повышает производительность программного обеспечения в областях, где критически важными являются контроль конфигурации, обеспечение безопасности или кибербезопасность.
- Снижает надежность после развертывания и затраты на обслуживание.

Преимущества виртуализации в DevOps:

Некоторые важные преимущества виртуализации:

- Снижает рабочую нагрузку, позволяя командам DevOps сосредоточиться на критически важных задачах.
- Экономит энергию, так как требует локального программного обеспечения или серверов, но использует виртуальные машины, что снижает потребление энергии.

- Улучшает использование оборудования.
- Помогает настроить локальную тестовую среду, используемую для различных типов тестирования, не беспокоясь о сбоях и нарушениях работы системы.
- Позволяет ускорить подготовку сервера.
- Снижение частоты сбоев благодаря возможности одновременного тестирования различных выпусков и исправлений.
- Помогает создавать надежные системы резервного копирования, автоматизируя резервное копирование данных, которое происходит каждую минуту.
- Обеспечивает эффективность использования ресурсов.
- Сокращает накладные расходы на ИТ и расходы на управление.

Проблемы виртуализации в DevOps:

Несмотря на различные преимущества DevOps, есть несколько проблем виртуализации, которые создают проблемы для команды. Имея это в виду, ниже перечислены некоторые из этих проблем виртуализации в DevOps.

- Требуется знание инфраструктуры: чтобы использовать виртуализацию в DevOps, командам требуется обширный опыт в этой технологии, что может быть проблемой для организаций. Для этого организациям придется обучать существующих сотрудников, что может занять много времени и средств.
- Отнимает много времени: в виртуализации конфигурация и приложение требуют времени, что делает процесс трудоемким.
- Угроза безопасности. Удаленный доступ и виртуализация рабочего стола и приложений подвержены риску утечки данных и взлома, что увеличивает риски безопасности.

2. Типы виртуализации KVM, QEMU

Что такое виртуализация kvm

Kernel-based Virtual Machine (KVM) — это технология виртуализации с открытым исходным кодом, встроенная в Linux®. В частности, KVM позволяет превратить Linux в гипервизор, который позволяет хост-машине запускать несколько изолированных виртуальных сред, называемых гостями или виртуальными машинами (VMS).

KVM является частью Linux. Если у вас Linux 2.6.20 или новее, у вас есть KVM. KVM был впервые анонсирован в 2006 году и объединен в основную версию ядра Linux годом позже. Поскольку KVM является частью

существующего кода Linux, он немедленно извлекает выгоду из каждой новой функции Linux, исправления и улучшения без дополнительной разработки.

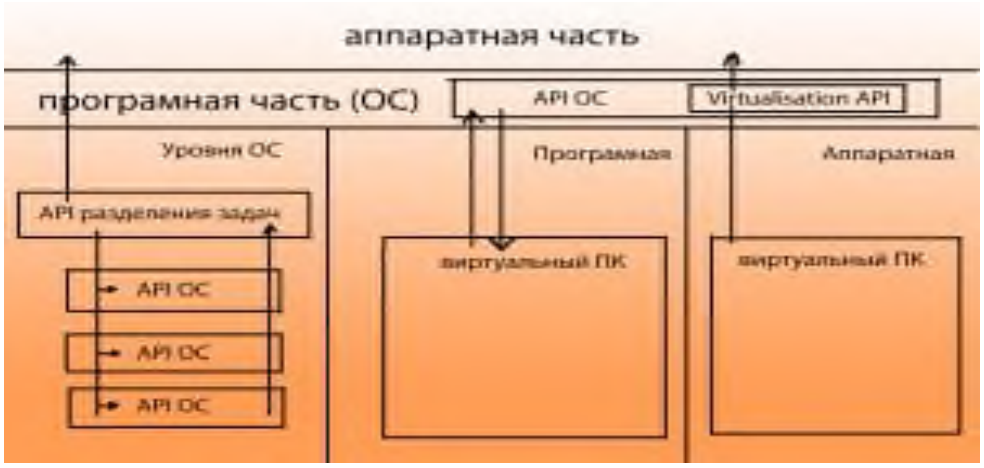


Схема 1. – Взаимодействие компонентов виртуальной машины с аппаратной частью

Особенности виртуализации для ядра linux

Для исполнения прямых аппаратных запросов в ОС должна иметься библиотека, которая направляла бы эти запросы аппаратной части напрямую. На платформах базы Linux долгое время никакой встроенной системы виртуализации (встроенного гипервизора), просто не существовало. Каждый производитель ПО для виртуализации, который поддерживало технологию аппаратной виртуализации, вынуждены были создавать собственные модули для ядра Linux (vboxdrv в Virtualbox, vmware-service в VMWare и пр.) Естественно, это не могло продолжаться вечно, и компания Qumranet, Inc, выкупленая затем Radhat создала ассоциацию Open Virtualization Alliance, которая была признана решить проблему отсутствия базового гипервизора для ядра Linux. Так и был создан гипервизор KVM или Kernel-based Virtual Machine.

Реализация

Гипервизор KVM представляет из себя загружаемый модуль ядра Linux, который предназначен для обеспечения виртуализации на платформе Linux x86. Сам модуль содержит компонент собственно виртуализации(kvm.ko), и процессорно-специфический загружаемый модуль kvm-amd.ko либо kvm-intel.ko.

Необходимым условием для использования KVM является поддержка инструкций виртуализации — Intel VT либо AMD, и ядро Linux версии 2.6.20 и выше. Существует также порт KVM под Free-BSD. Для вызова KVM традиционно используется QEMU, но также ведутся попытки добавить поддержку KVM в Virtualbox.

Сам по себе KVM не выполняет эмуляции. Вместо этого программа, работающая в пространстве пользователя, использует интерфейс /dev/kvm

для настройки адресного пространства гостя виртуальной машины, через него же эмулирует устройства ввода-вывода и видеоадаптер.

KVM позволяет виртуальным машинам использовать немодифицированные образы дисков QEMU, VMware и других, содержащие операционные системы. Каждая виртуальная машина имеет своё собственное виртуальное аппаратное обеспечение: сетевые карты, диск, видеокарту и другие устройства.

Использование

Для использования данного гипервизора существует множество реализаций. Некоторые представляют из себя целые специализированные библиотеки, другие имеют вид простых графических приложений.

Для наглядности рассматривается виртуализация KVM на базе библиотеку virt-manager.

Данная библиотека позволяет упростить вызов различных гипервизоров, предоставляя удобный интерфейс для автоматизации процесса виртуализации. Кроме того, библиотека имеет возможность работы с сетевой инфраструктурой, что иногда важно, при построении клиент-серверных рабочих мест.

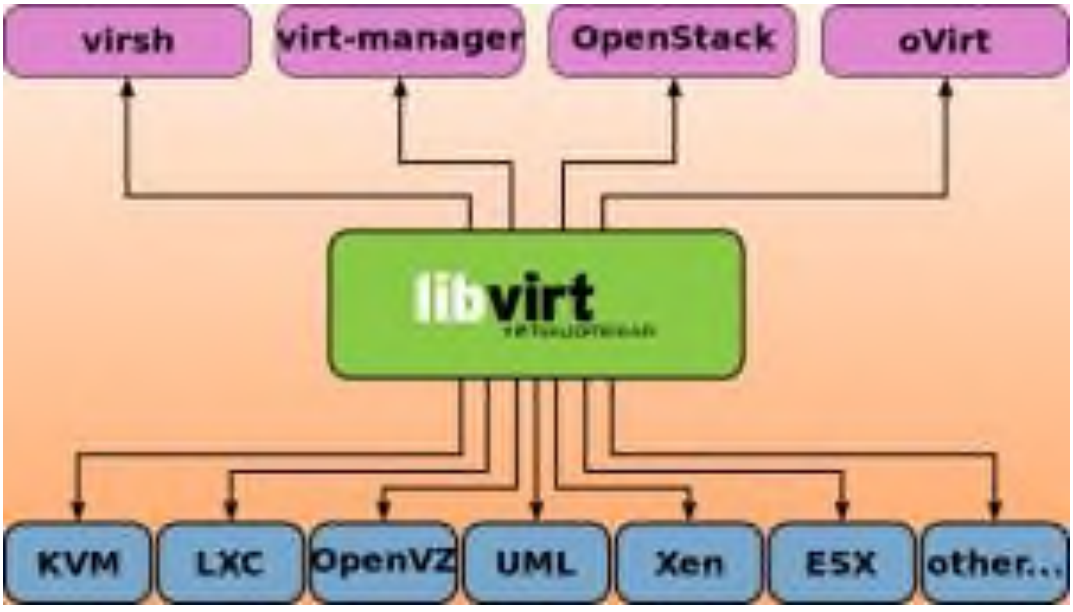


Схема 2. – Взаимодействие компонентов libvirt

QEMU позволяет создать фрейм для вызова гипервизора на клиентской системе. Данная программа настраивается аргументами вызова командной строки, является достаточно легкой и простой.

Существуют, кроме того, несколько графических оболочек, таких как Gnome-Boxes.

Выводы

Виртуализация – неотъемлемая часть современных корпоративных систем, она позволяет сэкономить колоссальные денежные и энергетические ресурсы. Развитие технологий виртуализации является приоритетным направлением многих организаций. Развиваются такие технологии как как VGAPassthrough (технология "проброса" видеокарты хост-устройства в виртуальную машину) и PCIPassthrough ("проброс" PCI устройства).

Как пользоваться qemu

Преимущество виртуализации в том, что она позволяет запустить несколько разных операционных систем на одном компьютере одновременно и при этом неважно какой они будут архитектуры. Среди домашних пользователей достаточно часто используются такие программы для эмуляции компьютера, как Virtualbox и VMware, это мощные программы с графическим интерфейсом и множеством возможностей, которые очень просто настроить.

Но существуют и другие решения, которые дают больше контроля над процессом виртуализации и при этом более сложны. Один из таких инструментов - qemu. В этой инструкции мы рассмотрим, что такое qemu и как пользоваться qemu для виртуализации, кроме того, разберем его преимущества перед другими программами.

Что такое qemu?

Qemu — это бесплатный инструмент с открытым исходным кодом для эмуляции и виртуализации работы операционных систем на компьютере. Программа может работать в Windows, Linux, MacOS и даже на Android. Ресурсы хоста, такие как процессор, жесткий диск, память, разделяются между виртуальными машинами.

Qemu использует аппаратную виртуализацию, поэтому может выполнять гостевые операционные системы почти так же быстро, как и на основном железе. Может использоваться гипервизор XEN или модуль ядра KVM в Linux. Qemu может работать в двух режимах работы:

- Полная эмуляция системы - в этом режиме qemu полностью эмулирует устройство, например компьютер, включая все его компоненты, процессор и различные периферийные устройства. Он может использоваться для запуска нескольких операционных систем без перезагрузки или отладки системного кода.
- Эмуляция пользовательского режима - работает только для Linux хоста, позволяет запускать процессы Linux, скомпилированные для одной

архитектуры в другой, например, ARM программы в x86. Полезно для разработки, кросс-компиляции и отладки.

Эмулировать можно такие архитектуры: x86 (32 и 64 бит), PowerPC (32 и 64 бит), ARM, MIPS (32 бит), Sprac (32 и 64 бит), Alpha, ColdFire(m68k), CRISv2 и MicroBlaze. Этот список уже более внушительный чем у Virtualbox.

Установка qemu

Перед тем как мы сможем использовать программу, необходимо ее установить. Если вы используете дистрибутив Linux, например, Ubuntu, то сможете найти программу в официальных репозиториях. Для Ubuntu команда будет выглядеть вот так:

```
sudo apt install qemu-kvm qemu
```

Для Fedora и других систем RedHat можно установить группу Virtualization:

```
sudo dnf install @virtualization
```

В ArchLinux используйте Pacman:

```
pacman -S qemu
```

Для Windows или MacOS вам нужно скачать исполняемый файл из официального сайта. Программа управляется только через терминал, так что вы главном меню системы ничего не появиться после установки. А теперь перейдем к тому, как настроить qemu.

Как пользоваться qemu?

Теперь, когда программа установлена и готова к использованию попытаемся разобраться как ее запустить и применять. Но сначала нужно разобраться какие команды и для чего используются. Эмулятор qemu создает много команд, но их можно разделить на группы:

- qemu-архитектура - эмуляция окружения пользователя для указанной архитектуры;
- qemu-system-архитектура - эмуляция полной системы для архитектуры;
- qemu-img - утилита для работы с дисками;
- qemu-io - утилита для работы с вводом/выводом на диск;

- qemu-user - оболочка для qemu-архитектура, позволяет запускать программы других архитектур в этой системе;
- qemu-system - оболочка для qemu-system-архитектура, позволяет полностью эмулировать систему нужной архитектуры.

Сначала разберемся с эмуляцией полной системы, поскольку для решения этой задачи виртуальная машина qemu используется чаще всего, а уже потом перейдем к режиму пользователя.

Использование qemu-system

Чтобы вы понимали, что и откуда берется, для начала рассмотрим опции утилиты qemu-system. Синтаксис команды такой:

$ qemu-system параметры

Куда сложнее здесь синтаксис каждого из параметров:

-имя_параметра имя_опции=значение: значение2

Мы рассмотрим только основные параметры, и их опции, которые нам понадобятся:

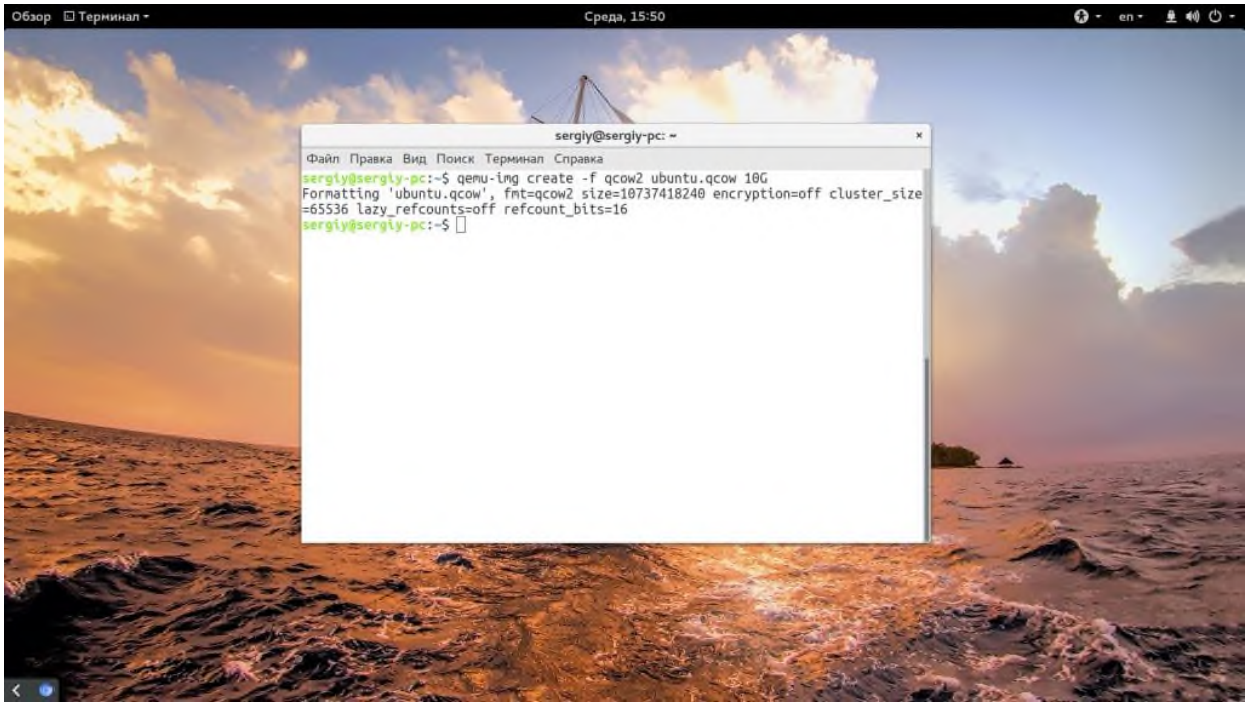
- -machine указывает тип компьютера, который вы собрались эмулировать, можно выбрать ubuntu, pc, pc-q35 и другие варианты, смотрите подробнее командой -machine help;
- -cpu - тип процессора, можно передать непосредственно тип процессора, а также дополнительные флаги;
- -smp - включает симуляцию мультипроцессорной архитектуры;
- -boot - позволяет настроить порядок загрузки устройств, a,b - дискета, c - первый жесткий диск, d - CDROM, n-p - загрузка через сеть, по умолчанию виртуальная машина загружается из жесткого диска;
- -m - указывает количество оперативной памяти, доступной машине;
- -k - раскладка клавиатуры по умолчанию;
- -soundhw - подключить звуковую карту к системе;
- -device - подключить указанное устройство к машине, нужно указать драйвер и его опции;
- -name - имя гостевой системы, будет отображаться в верху окна;
- -uuid - установить UUID для системы;

- -fda, fdb - подключить дискету из файла;
- -hda, hdb - подключить жесткий диск из файла;
- -cdrom - подключить оптический диск;
- -usb - включить поддержку USB;
- -usbdevice - подключить USB устройство;
- -display - выбрать тип отображения, доступно sdl, curses, gtk, none, vga и другие;
- -full-screen - запустить в полный экран;
- -no-acpi - отключить поддержку ACPI;
- -net - создать сетевой интерфейс;
- -realtime включить режим реального времени, опции mclock нужно передать значение on;
- -gdb - ожидать подключения отладчика gdb;
- -enable-kvm - включить поддержку kvm;
- -loadvm - загрузить сохраненное состояние;
- -daemonize - сделать процесс фоновым;
- -snapshot - записывать изменения не в файл жесткого диска, а в отдельные файлы;
- -nographic - отключить графический вывод.
- -kernel - использовать указанный образ ядра linux.
- -append - командная строка для ядра;
- -initrd - файл initrd для загрузки Linux.

Мы рассмотрели опции для qemu-system-x86-64, для других архитектур, они могут немного отличаться. А теперь разберем несколько простых примеров как использовать qemu, как создать машину qemu и настроить ее.

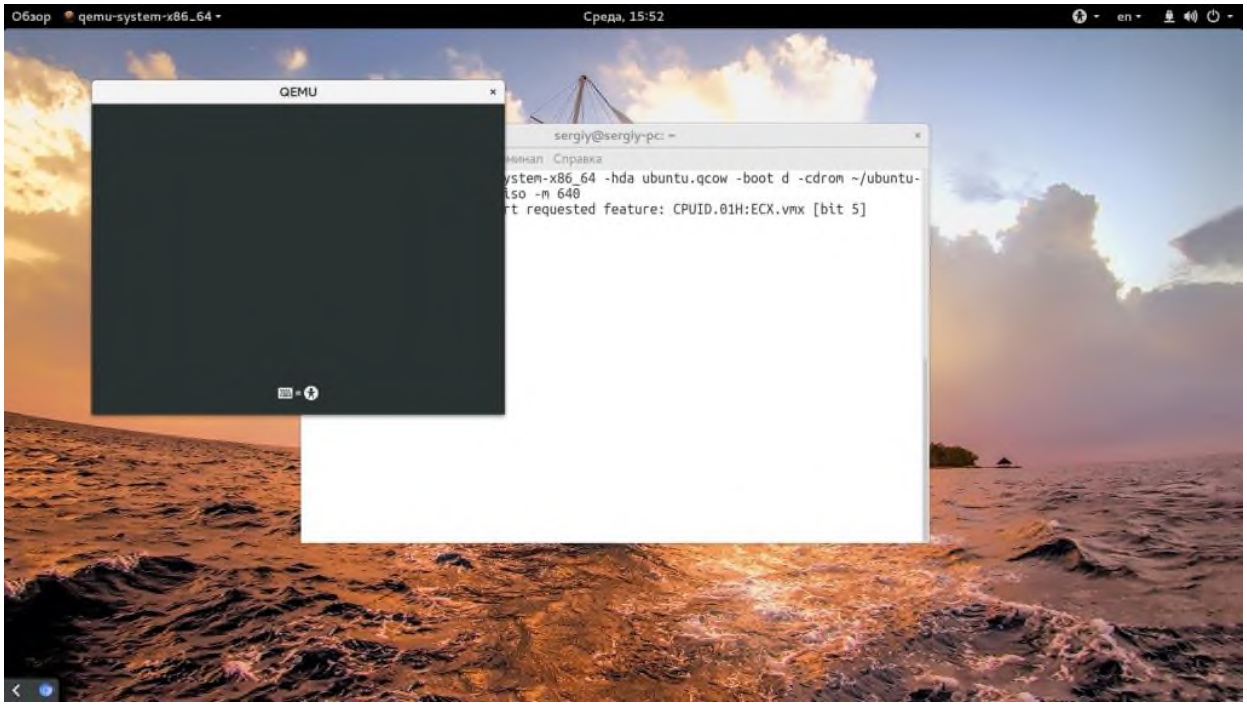
Сначала нужно создать жесткий диск для установки. Вы можете использовать реальные жесткие диски, но работать с образами намного удобнее. Можно просто создать пустой файл, заполненный нулями, а затем форматировать его в нужную файловую систему во время установки, но также можно создать файл формата qcow2, этот формат используется по умолчанию в qemu. Воспользуемся командой qemu-img:

qemu-img create -f qcow2 ubuntu.qcow 10G



Опция -f указывает формат нового файла. Теперь мы можем непосредственно перейти к запуску qemu-system. Вот так будет выглядеть самая простая команда запуска системы:

```
qemu-system-x86_64 -hda ubuntu.qcow -boot d -cdrom ~/downloads/ubuntu-17.04-server-amd64.iso -m 640
```



Здесь мы подключаем наш жесткий диск как hda, затем указываем что нужно загружаться с cdrom и подключаем образ системы ubuntu к нему.

Последний параметр указывает сколько оперативной памяти будет выделено для машины.

Дальше откроется окно, похожее на VritualBox и начнется установка системы. После того как установка будет завершена, вы сможете запускать машину командой:

```
qemu-system-x86_64 -hda ubuntu.img -m 640
```

Создавать виртуальную машину с другой архитектурой не очень сложно, достаточно изменить команду. Например, сделаем виртуальную машину ppc:

```
qemu-system-ppc -hda ubuntu.qcow -boot d -cdrom ~/downloads/ubuntu-17.04-server-amd64.iso -m 640
```

Поскольку сейчас мы не используем kvm, то система будет работать очень медленно все можно очень сильно ускорить, если его включить с помощью опции -enable-kvm:

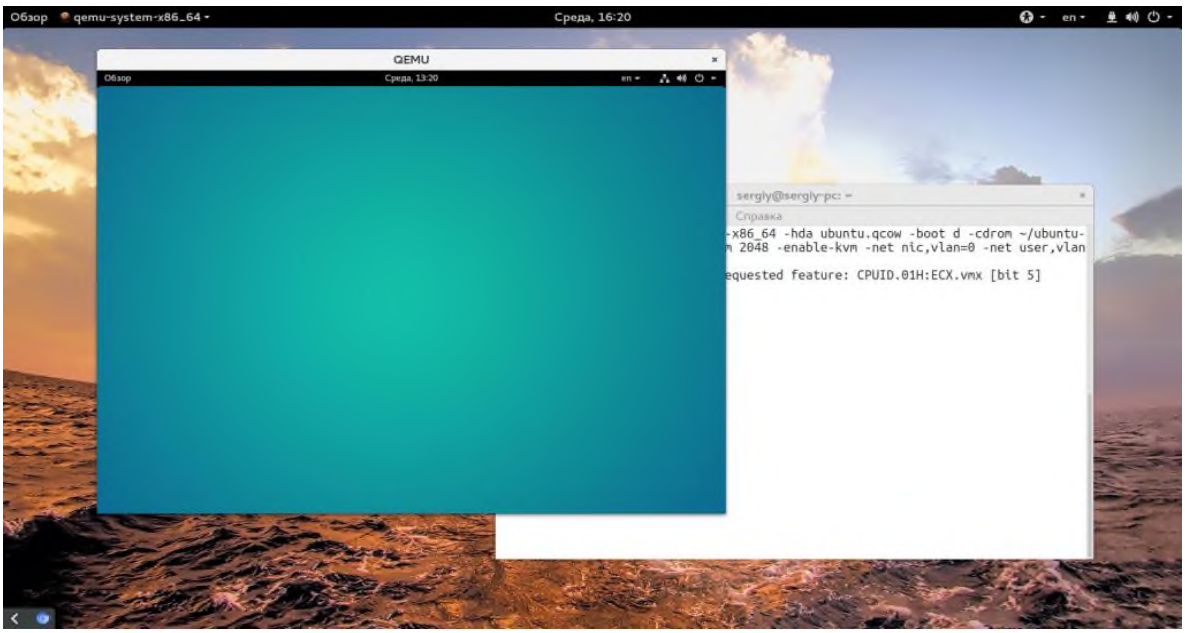
```
qemu-system-x86_64 -hda ubuntu.qcow -m 640 -enable-kvm
```

По умолчанию в гостевой системе не будет звука, но вы можете подключить туда звуковую карту:

```
qemu-system-x86_64 -hda ubuntu.qcow -m 640 -enable-kvm -soundhw ac97
```

Не всегда мы запускаем машину только в качестве сервера, иногда нужно запустить обычную десктопную систему, в таком случае нам придется настроить виртуальную видеокарту, потому что того, что идет по умолчанию хватит только на консоль. Для этого используется опция -vga:

qemu-system-x86_64 -hda ubuntu.qcow -m 640 -enable-kvm -soundhw ac97 -vga qxl.

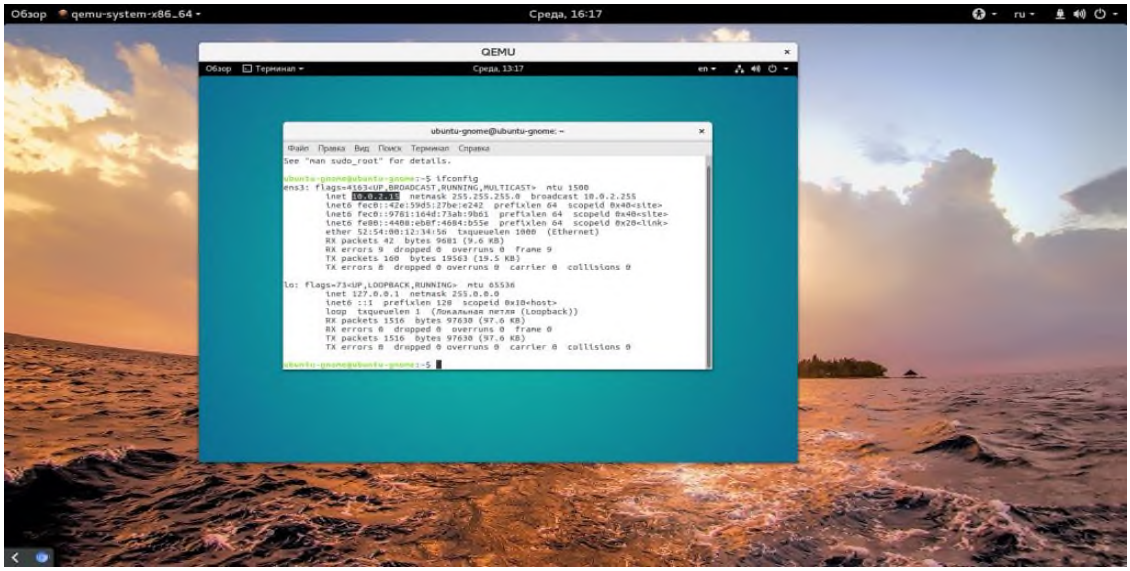


С помощью опции -global qxl-vga.vram_size можно задать объем видеопамяти, доступный машине, значение указывается в байтах, например:

```
qemu-system-x86_64 -hda ubuntu.qcow -m 640 -enable-kvm -soundhw ac97 -vga qxl
-global qxl-vga.vram_size=4294967
```

Для многих машин важно чтобы из них был доступ в сеть интернет. Настройка сети qemu — это достаточно обширная тема, которую мы не будем полностью раскрывать в этой статье. Если кратко, то, в qemu используется специальный виртуальный шлюз vlan, который позволяет соединять машины с внешней сетью или между собой. Каждый vlan имеет свой номер. Самый простой способ настроить сеть - использовать такие параметры:

```
 qemu-system-x86_64 -hda ubuntu.qcow -m 640 -enable-kvm -net nic,vlan=0 -net
user,vlan=0
```

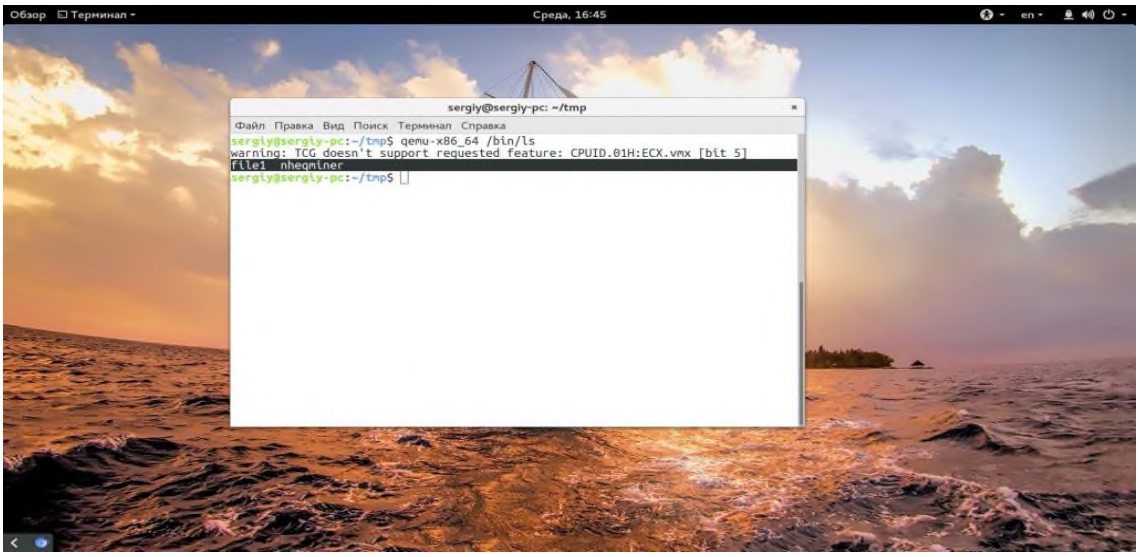


Здесь мы два раза вызываем опцию -net, в первый раз создаем виртуальную сетевую карту с помощью параметра -nic и подключаем ее к vlan с номером 0. Вторая же опция -net создает наш vlan 0 с типом конфигурации user, которая не требует дополнительной настройки и автоматически подключается к сети хоста.

Использование эмуляции окружения

Теперь рассмотрим использование qemu для эмуляции архитектуры в окружении пользователя. Команда qemu-user или qemu-архитектура позволяет выполнять программы, собранные для другой архитектуры прямо в вашей системе. Это очень часто используется для отладки программ, собранных для arm на компьютере или других подобных задач. Команде достаточно передать команду и ее параметры:

```
qemu-x86_64 /bin/ls
```



Точно так же вы можете выполнить arm программу или программу для любой из поддерживаемых архитектур.

Выводы

Мы очень кратко рассмотрели, как пользоваться qemu, основные настройки этой утилиты и опции. На самом деле там намного больше опций и возможностей. Одна только возможность эмулировать такое огромное количество архитектур чего стоит. Если для вас эмулятор qemu слишком сложен через терминал, то можно использовать графический интерфейс, например, virt-manager.

3. Системы управления виртуализацией

Что такое управление виртуализацией?

В течение многих лет, когда компании развертывали новое приложение для своих сотрудников и клиентов, оно запускалось на своем собственном компьютерном сервере. Все чаще компании запускают множество приложений одновременно на одном сервере с использованием технологии виртуализации. Это создало потребность в новом типе управления, называемом управлением виртуализацией. Управление виртуализацией помогает компаниям развертывать, обновлять, отслеживать и поддерживать виртуализированные приложения. Используя программное обеспечение для управления виртуализацией, ИТ-менеджеры могут контролировать свою виртуализированную вычислительную среду.

Потребность в управлении виртуализацией возникла из-за почти универсального внедрения виртуализации серверов и приложений. Компании поняли, что традиционный подход одного приложения на сервер заключается в неэффективном использовании вычислительных ресурсов. Довольно часто многие серверы работали с долей их номинальной производительности. Это оказалось очень дорогостоящим, так как каждый недостаточно используемый сервер все еще нуждался в операционной системе, электричестве и должен был управляться. Виртуализация позволяет нескольким приложениям совместно использовать один сервер без конфликтов. Это позволяет компаниям консолидировать количество серверов, которыми они должны владеть, что значительно сокращает расходы.

Чтобы виртуализация работала, приложение должно быть сопряжено с операционной системой в программном элементе, называемом виртуальной машиной. Программное обеспечение для управления виртуализацией часто используется для создания этих виртуальных машин. Затем виртуальная машина должна быть размещена на сервере для запуска. Программное обеспечение для управления виртуализацией используется для установки и тестирования отдельных виртуальных машин на сервере.

Виртуализированные приложения, такие как традиционные приложения, работающие на сервере, должны обновляться и проверяться с течением времени. Для этого программное обеспечение для управления виртуализацией используется для контроля производительности виртуальной машины. Если производительность ниже номинальной, администратор может использовать программное обеспечение для управления, чтобы выделить больше памяти или вычислительной мощности. Это поможет

быстрее запустить виртуальный экземпляр приложения. Аналогичным образом, если возникает проблема с приложением, программное обеспечение для управления виртуализацией используется для устранения неполадок и исправления ситуации. Кроме того, программное обеспечение используется всякий раз, когда в приложении необходимо установить новое исправление или применить обновление.

Когда виртуализация серверов и приложений впервые стала популярной, инструменты управления виртуализацией были довольно просты и предлагали лишь ограниченную функциональность. Поскольку виртуализация стала основной частью большинства ИТ-инфраструктур, инструменты управления добавили множество функций. Например, многие инструменты виртуализации позволяют ИТ-администраторам одновременно управлять своими виртуальными и физическими сетями.

Как от виртуализации перешли к облачным технологиям

Кроме того, инструменты управления виртуализацией теперь выполняют более стратегические функции. Сначала система управления использовалась для обеспечения бесперебойной работы приложений. Теперь инструменты управления все чаще используются для определения моделей использования и помощи в планировании для удовлетворения новых потребностей.

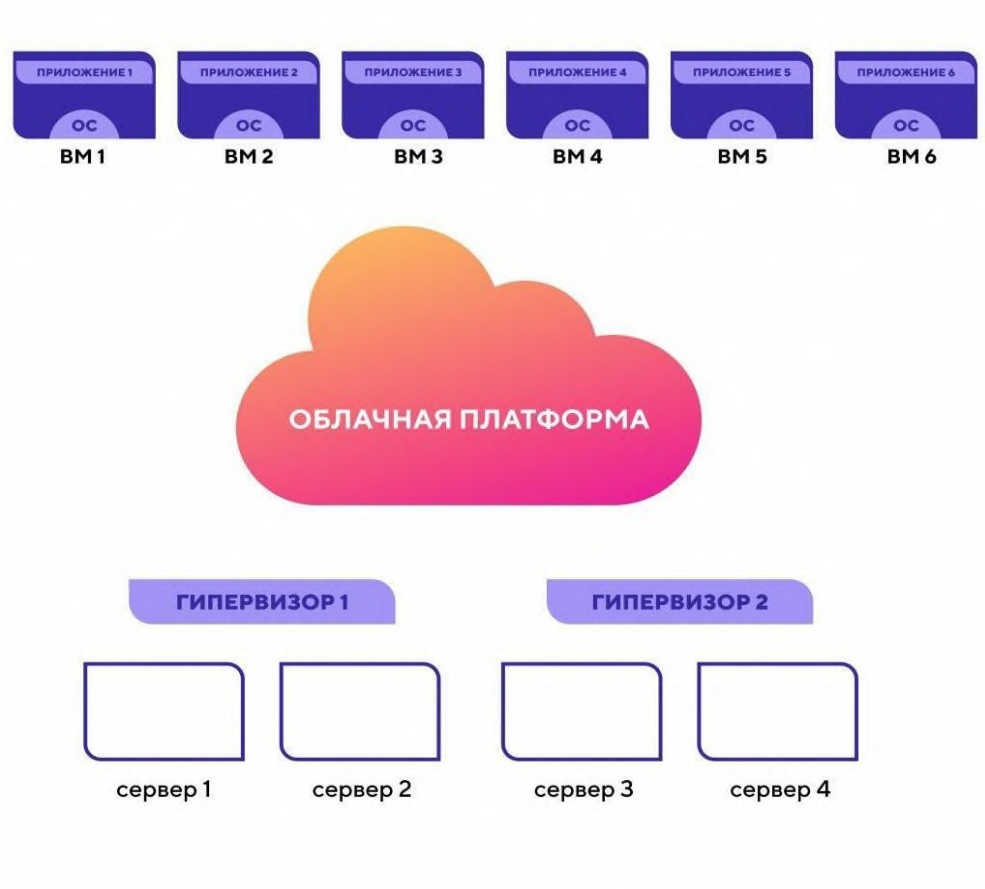
Виртуализация стала основой для появления облачных вычислений — нового способа потребления вычислительных ресурсов, в котором участвуют провайдер ресурсов и их пользователь. В частном облаке в роли «провайдера» выступает IT-департамент, в роли пользователей — все остальные.

Облачные вычисления — модель потребления вычислительных ресурсов, когда пользователь получает к ним доступ по требованию через сеть. «По требованию» (on demand) значит, что пользователь, когда ему нужно, резервирует некоторое количество ресурсов провайдера из их общего пула. Когда ресурсы больше не требуются, резервирование аннулируется, и ресурсы может использовать другой пользователь.

Само слово «облако» обозначает пул виртуализированных ресурсов, которые предоставляются потребителю провайдером в пользование.

Для технической организации облака над гипервизором и под набором виртуальных машин располагают специальное программное обеспечение (облачную платформу, например, «Базис.Digital Energy»), абстрагируясь ВМ не только от оборудования, но и от гипервизоров. Это позволяет совместно

использовать в одной IT-инфраструктуре различные гипервизоры, обычно несовместимые между собой. А также объединять все виртуальные ресурсы в единый пул с централизованным интерфейсом управления, распределения и контроля.



Структуру облачной виртуализации можно представить послойно: серверы, гипервизоры, облачная платформа, виртуальные машины с гостевыми ОС и приложениями

Получается, виртуализация — это мост для перехода к облакам. А облачные платформы открывают новые возможности использования виртуальной IT-инфраструктуры. Например, можно организовать доступ к ресурсам множеству пользователей, разместить на разных гипервизорах изолированные проекты и объединять разные гипервизоры под общим управлением. Это и есть задачи провайдера.

Облачные платформы помогают компаниям гибко управлять IT-инфраструктурой, и мгновенно ее масштабировать. По запросу пользователя выделяется столько ресурсов, сколько нужно приложениям компании. Это позволяет поддерживать бесперебойную работу даже в условиях меняющихся нагрузок.

Платформы виртуализации создают гибкую среду для тестирования различных проектов — например, в сфере разработки ПО. Виртуализация

также создает основу для внедрения облачных решений, которые повышают контроль бизнеса над критичными данными.

Сейчас на рынке представлено множество решений виртуализации: продукты VMware, vStack, платформа от Microsoft и другие. У каждого есть свои преимущества, которые подходят для решения различных задач бизнеса. Расскажем о них подробнее:

VMware

VMware — американская компания, лидер среди поставщиков решений для виртуализации. К ним относятся: vSphere (ПО для виртуализации серверов), vCenter Server (ПО для централизованного управления серверами), NSX Data Center (виртуализированные службы сети и безопасности) и Horizon 7 (платформа для виртуальных компьютеров и приложений). Компания постоянно расширяет функциональность.

Продукты могут быть слишком сложными и дорогостоящими для малого и среднего бизнеса, не разбирающегося в технологиях. VMware предлагает решения премиум-класса по более высокой цене. Однако в крупных корпоративных средах с большим бюджетом, которые охватывают множество платформ, ОС и архитектур, VMware остается лучшим выбором.

Особенности:

Поддерживает системы Windows, Linux, Solaris, FreeBSD, Netware и другие.

Решения подходят для ИТ-специалистов, разработчиков и предприятий.

Можно одновременно использовать несколько приложений в разных ОС. Перезагрузка не требуется.

Может работать с облаком.

Может быть интегрирована со сторонним программным обеспечением.

Многие инструменты vSphere обеспечивают поддержку балансировки нагрузки и динамической миграции, а также ряд других полезных функций.

vStack

vStack — платформа для внедрения виртуального дата-центра на базе обычного недорогого оборудования. Это гиперковергентное решение для предприятий, разработанное компанией ITGLOBAL.COM LABS с применением open-source-технологий. vStack является более доступной платформой, однако по производительности не уступает VMware или другим корпоративным решениям для хранения и виртуализации.

Особенности:

Нет необходимости переплачивать за дорогое оборудование, платформа может работать с устройствами потребительского класса. Нет привязки к поставщику.

Разработана на базе ОС FreeBSD (семейство UNIX), файловой системы ZFS (поддержка больших объемов данных), гипервизора bhyve (интерфейс UEFI, поддержка NVMe, быстродействие).

Поддержка Windows и Linux.

Доступная альтернатива от российского поставщика на уровне популярных западных решений.

Citrix (Xen)

Citrix предлагает набор недорогих продуктов корпоративного уровня. Их решения для виртуализации можно рассматривать в качестве альтернативы VMware. Компания отвечает за разработку кроссплатформенного гипервизора с открытым исходным кодом Xen Project.

Особенности:

Есть бесплатная версия.

Платформа с открытым исходным кодом.

Расширенные функции виртуализации. Работает с гипермасштабируемыми облаками.

Поддерживаемые ОС: GNU/Linux, FreeBSD, MiniOS, NetBSD, Solaris, Windows и другие.

OpenVZ

OpenVZ — это мощная платформа на основе метода контейнеризации с открытым исходным кодом. Платформа бесплатная и быстрая, за счет того, что построена на базе ядра Linux. Однако в роли «гостевых» систем могут выступать только дистрибутивы Linux. Ее настройка может быть не такой

простой, но это компенсируется широким спектром функций и большим количеством конфигураций.

Особенности:

Состоит из модифицированного ядра Linux и пользовательских утилит.

Поддерживает только ОС Linux.

Виртуализация происходит на основе контейнеризации для Linux. Платформа создает несколько безопасных изолированных контейнеров (также известных как VE или VPS) на одном физическом сервере, что позволяет лучше использовать ресурсы сервера и избегать конфликтов между приложениями.

Virtuozzo

Virtuozzo — компания, созданная для обеспечения коммерческой поддержки средств виртуализации Kernel-based Virtual Machine (KVM) с открытым исходным кодом. Она добавляет множество улучшений в популярный гипервизор KVM (о нем ниже). Одна из самых доступных платформ.

Особенности:

Расширенная функциональность и поддержка.

Пользовательская база состоит в основном из малых и средних компаний.

Виртуализация серверов сочестается с программно-определяемым хранилищем и контейнеризацией.

Поддерживает только ОС Linux.

KVM

Kernel-based Virtual Machine — ПО с открытым исходным кодом, обеспечивающее виртуализацию в среде Linux. Платформу можно скачать бесплатно и использовать для разработки на своей виртуальной машине. Является наиболее популярной альтернативой коммерческим платформам.

Особенности:

Обеспечивает быструю и надежную виртуализацию для устройств на Linux.

Самая низкая стоимость.

Сообщество постоянно развивает платформу, экосистему решений и функциональность.

Предоставляет множество вариантов кастомизации. Подходит для разработчиков собственных программных решений.

Microsoft Hyper-V

Возможности ПО для виртуализации Microsoft Hyper-V тесно интегрированы с широким набором продуктов Windows. Этот инструмент предоставляет множество критически важных средств виртуализации по более низкой цене, чем VMware. Hyper-V более популярен среди ИТ-специалистов, которые используют другие продукты Microsoft.

Особенности:

Платформа входит в состав Windows 10 Pro и Windows Server (2012 и 2016) без каких-либо дополнительных затрат для пользователя.

Поддержка Linux ограничена.

Hyper-V позволяет относительно неопытным пользователям создавать свою виртуальную серверную среду.

Как выбрать платформу для виртуализации

Выбор ПО для успешного внедрения виртуализации во многом зависит от того, соответствует ли платформа целям компании. Вот несколько факторов, которые следует учитывать:

ОС

Первое, что следует учитывать при выборе виртуальных решений, — это поддерживаемые операционные системы. Некоторые из решений разработаны для поддержки определенных систем, например только Windows — как Microsoft Hyper-V. Однако, если компании требуется одновременно запускать несколько типов ОС, нужно выбрать соответствующую платформу — например, решения VMware.

Масштабируемость и гибкость платформы

Для онлайн-бизнеса приоритетом может стать масштабируемость платформы — или способность поддерживать большое число одновременно работающих ВМ. Это позволяет быстро расширять ресурсы для поддержки большего числа пользователей и создает более гибкую среду для внедрения новых функций.

Потребление ресурсов и экономичность

С помощью виртуализации можно сократить потребляемые ресурсы оборудования, однако стоит выбирать платформу, которая потребляет меньше памяти. Если в системе ограничено ОЗУ, дополнительное использование ресурсов, необходимое для поддержки инструмента виртуализации, может замедлить работу системы. В итоге это может привести к увеличению расходов на обновление оборудования. И если цена является главной проблемой, доступны бесплатные варианты виртуализации.

Соответствие задачам компании

Например, для крупных компаний подходят многофункциональные enterprise-решения уровня VMware. Для небольших компаний-разработчиков ПО более важным будет гибкость платформы, возможность тестирования своих продуктов в различных средах и невысокая стоимость.



МОДУЛЬ 10. Облачные решения

1. Виды облачных сервисов (публичные, частные, интернет вещей, смешанные)
2. Организация сети
3. Вычислительные мощности или облачные вычисления
4. Контексты безопасности. Менеджеры секретов
5. Кластеры. Ресурсы под управлением облачным провайдером

1. Виды облачных сервисов (публичные, частные, интернет вещей, смешанные)

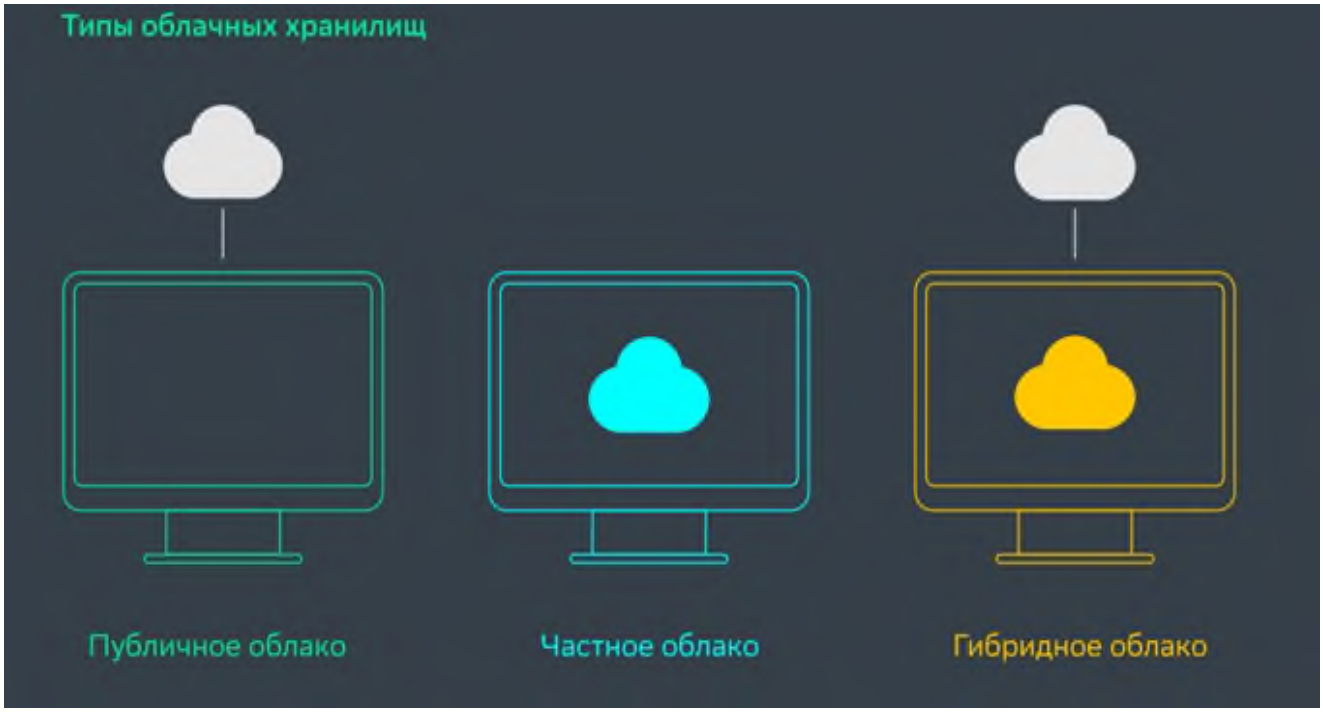
Что такое «облако»?

Облако — это виртуальные ресурсы, которые компании используют для своих ИТ-процессов: оперативная память, сетевые соединения, пространство на диске, приложения и сервисы. Они доступны на виртуальных серверах, которые работают на базе обычных. При этом мощность виртуальных серверов можно менять, а нагрузку распределять между ними в любой пропорции.

Какими бывают облачные сервисы?

Различают три основных типа облачных хостингов — частные, публичные и гибридные. Рассмотрим каждое подробнее.

Типы облачных хранилищ.



Частное облако: особенности и преимущества

Виртуальное частное облако (VPC, virtual private cloud) — инфраструктура, которая используется только одной компанией.

Частное облако можно развернуть на собственных или арендованных серверах, самостоятельно или с помощью специалистов со стороны провайдера. Но в любом случае доступ к нему будет только у одной компании.

Плюсы и минусы частного облака	
+	**-**
ИТ-ресурсы и мощности работают эффективно: нагрузка распределяется между физической и виртуальной инфраструктурой, поэтому сбоев и отказов оборудования гораздо меньше.	Высокая цена, которая складывается из стоимости покупки (или аренды) оборудования, развертывания и обслуживания.
Все данные надежно защищены, так как к серверу есть доступ только у вас.	На развертывание уходит больше времени. Необходимо создать и настроить систему индивидуально под конкретную компанию и ее внутренние стандарты хранения данных.
Можно вкладывать средства и наращивать виртуальную инфраструктуру под свои потребности, экономя на физическом ИТ-оборудовании и помещениях для него.	Масштабировать можно только в рамках выделенного облака. Если понадобится больше мощностей и сервисов, нужно докупать и настраивать их самим. В первую очередь это касается собственного ЦОДа.
Можно выделять разные по мощности и объему ресурсы для разных отделов и филиалов внутри компании — в зависимости от загрузки и задач.	Нельзя расширить мощности только на определенный срок — например, на время запуска нового продукта.
Если облако создано на базе собственного дата-центра, система доступна в любой момент по локальной сети, без интернета.	

Частные облака чаще представляют собой полноценную инфраструктуру, то есть IaaS. Этот вариант подходит крупным сетевым компаниям, которые используют большую и сложную ИТ-систему: разработчики ПО, телеком-операторы, госструктуры, банки, страховые компании, промышленные корпорации.

Некоторые компании могут работать только с частными облаками, так как по закону все данные должны храниться на их собственных серверах. Например, сотовые операторы и интернет-провайдеры.

Что такое публичное облако и каковы его преимущества?

Публичное или общедоступное облако (public cloud) — решение, при котором провайдер предоставляет доступ к сервису сразу нескольким клиентам. Каждый пользователь создает столько виртуальных серверов, сколько ему нужно, и управляет ими, но физического доступа к оборудованию у них нет.

Плюсы и минусы публичного облака	
+	**-**
Легко получить IT-инфраструктуру для старта проекта. Не нужно закупать оборудование и настраивать его — достаточно выбрать нужные сервисы и подключить их.	Невыгодно для масштабных проектов в длительной перспективе.
Гибкое ценообразование. Вы платите по принципу Pay-As-You-Go — то есть только за те ресурсы и мощности, которыми пользуетесь.	Размещение в облаке особо конфиденциальной информации (например, персональных данных) требует от облачного провайдера соответствия Федеральным законам.
Легко масштабировать. Например, в пик сезонного спроса можно подключить больше виртуального пространства и увеличить мощность серверов, а в остальное время вернуться к минимальным значениям.	Вы полностью зависите от провайдера: не можете контролировать качество оборудования и сетевого соединения.
Не нужно выделять ресурсы и сотрудников для поддержки виртуальной среды.	Защита данных потребует дополнительного внимания.
Удобно тестировать новые продукты и сервисы, которые еще не запущены. Также легко провести тест, чтобы понять потребности компании в ИТ-ресурсах.	

Гибридная модель подходит компаниям, у которых есть свое частное облако, но ее недостаточно для запуска и тестирования новых сервисов. Другой вариант — крупные холдинги, куда входят компании разных сфер, с разными требованиями к безопасности данных. Разработчики ПО с помощью

гибридного облака смогут задействовать дополнительные мощности и избежать пиковой нагрузки на сеть на стадии разработки и запуска.

Что такое мультиоблако и чем оно отличается от гибридного?

Мультиоблачные системы тоже сочетают в себе частные и публичные облака, но последние размещены не у одного провайдера, а у нескольких. Например, почта — у одного, а резервное хранилище данных — у другого.

Плюсы и минусы публичного облака	
+	-
Для каждого сервиса или базы данных можно выбирать поставщика с самыми выгодными условиями.	Приходится постоянно следить и управлять коммуникацией сразу с несколькими провайдерами. Это нагружает и менеджеров, и финансовый отдел, и ИТ-службу.
Дополнительная страховка. Если у одного провайдера произойдет сбой или утечка данных, другие части ИТ-системы не пострадают.	Есть риск потерять данные при передаче их из одного облако в другое.

Какие факторы учесть при выборе?

Чтобы разобраться, какое облако стоит выбрать, сравним частные, публичные и гибридные облака по ключевым параметрам:

	Частные	**Публичные**	**Гибридные/Мультиобл ака**
Затраты на старте	**Высокие:** нужно самим спроектироват ь и развернуть систему, купить оборудование , подобрать персонал.	**Низкие:** нужно просто зарегистрироват ься и сразу можно начинать создавать и использовать ИТ-архитектуру. Оплата производится в конце месяца только за фактически использованные ресурсы (модель	**Средние (мультиоблако) или высокие (гибридное):** нужно не только развернуть частное облако, но и настроить интеграцию с публичным.

		Pay-As-You-Go).	
Поддержка и обслуживание	**Сложно и затратно:** нужно самим поддерживать работу системы, устранять проблемы и сбои, обновлять оборудование, ПО и лицензии.	**Просто и недорого:** все расходы и технические сложности берет на себя провайдер.	**Средний уровень:** расходы и обслуживание распределяются между компанией и провайдером. С мультиоблаком немного сложнее: нужно постоянно взаимодействовать с несколькими провайдерами и обмениваться информацией.
Гибкость и масштабируемость	**Низкая:** чтобы расширить возможности, нужно закупить и настроить оборудование.	**Высокая:** дополнительные мощности можно получить мгновенно.	**Высокая:** можно быстро нарастить мощности за счет публичного облака или даже нескольких.
Безопасность	**Высокая:** доступ есть только у вашей компании, но нужно все тщательно спроектировать, настроить и контролировать.	**Требует особого внимания:** обеспечение безопасности определяется возможностями провайдера. Со стороны пользователей потребуется повышенное внимание к вопросу защиты данных.	**Средняя:** вы полностью уверены в защите данных на своих серверах, но не на стороне провайдера. Многое зависит от того, какие данные и сервисы вы размещаете у себя, а какие — в публичном облаке.

Бесперебойность	**Средняя:** получить доступ к данным и сервисам, а также сделать резервные копии достаточно легко. Но для этого лучше развернуть облако на базе 2-3 дата-центров. В случае сбоев и отказов системы нужно ждать, пока ее восстановят.	**Высокая:** в любой момент можно получить доступ к сервисам и данным, а резервные копии часто делают автоматически. Если происходят сбои, система быстро восстанавливается без потери данных по аварийной схеме.	**Средняя:** возможны трудности с частным облаком, но всегда можно перераспределить данные и мощности между несколькими публичными. У мультиоблаков здесь преимущество перед гибридными.
Инновации	Все передовые технологии приходится внедрять самостоятельно, что бывает слишком сложно и затратно для корпораций. Исключения — частные облака на базе провайдеров, где можно	Провайдеры публичных облаков быстро отслеживают и внедряют инновационные решения для своих клиентов. Например, PaaS-сервисы — главный «локомотив» трансформации облачных систем. Это помогает расширить	Можно пользоваться передовыми решениями провайдеров и внедрять их у себя в частном облаке. У тех, кто использует мультиоблака, есть дополнительный бонус: можно выбирать все лучшее сразу от нескольких поставщиков.

	получить доступ к их сервисам.	возможности, предоставить лучшее качество соединения и мгновенно подстроиться под потребности клиента.	

Кроме того, необходимо учитывать потребности компании, ее технические и финансовые возможности, требования к защите данных и планы по развитию ИТ-инфраструктуры на ближайшие 5 лет.

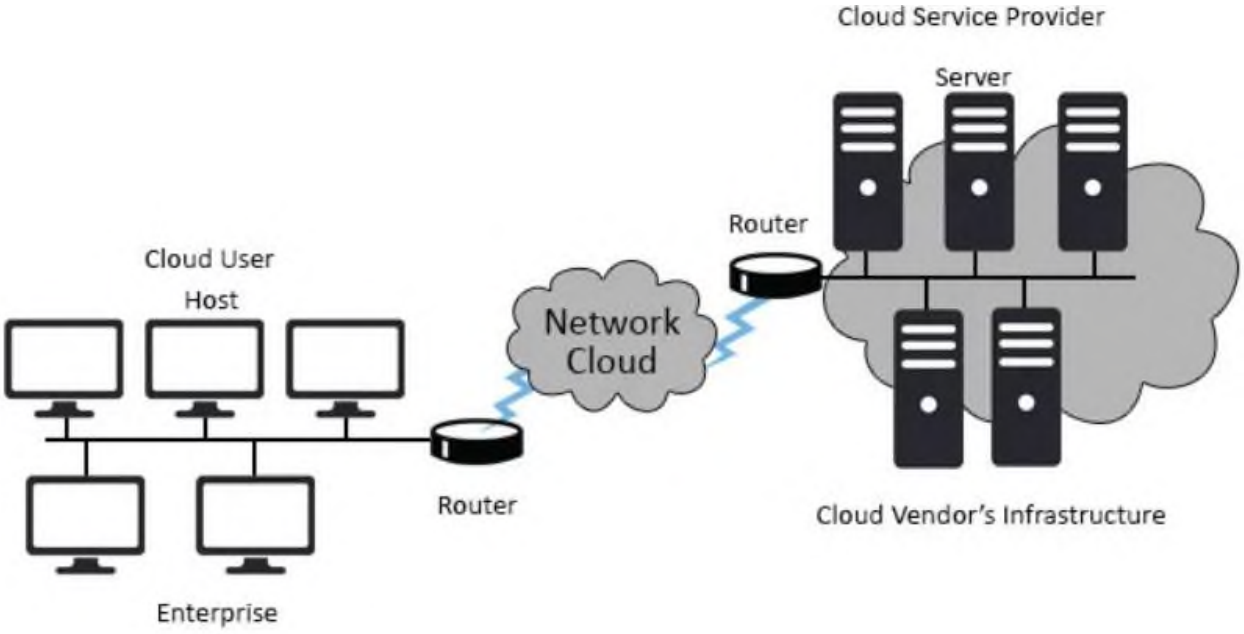
2. Организация сети

Введение в облачные сети

Cloud Networking — это услуга или технология, в которой сетевой процесс компании размещается в общедоступном или частном облаке. Облачные вычисления — это источник контроля, в котором несколько вычислительных ресурсов совместно используют одну и ту же платформу, и пользователи также могут получить доступ к этим ресурсам в определенной степени. Облачные сети аналогичным образом разделяют сети, но предоставляют более продвинутые функции и сетевые функции в облаке с подключенными серверами, созданными в киберпространстве.

Почему облачные сети?

- Облачные сети востребованы многими организациями для их быстрой и безопасной доставки, быстрой обработки, надежной передачи данных без каких-либо потерь, удобной настройки. К числу пользующихся преимуществом компаний, которые предпочитают облачные сети, относятся поставщики веб-услуг, электронная коммерция, поставщики облачных услуг, операторы сетей, поставщики облачных услуг.
- Облачные сети позволяют пользователям создавать свои сети в соответствии с требованиями облачных сервисов. Аутентичная облачная сеть предлагает высококачественный мониторинг для глобально расположенных серверов, контролирует поток трафика между взаимосвязанными серверами, защищает системы с повышенной сетевой безопасностью и обеспечивает видимость для пользователя благодаря централизованному управлению. Доступ к Интернету может быть улучшен и увеличена пропускная способность для продвижения нескольких сетевых функций в облако.



- Приведенная выше диаграмма показывает работу Cisco. Он показывает отделение ресурсов от пользователя, а затем они связывают его с облаком. Он включает управление распределенным беспроводным доступом к маршрутизаторам, а дополнение к нему обеспечивает все высокотехнологичные функции, как указано выше. Цель состоит в том, чтобы проектировать и управлять новыми облачными сетями с более защищенным частным доступом через глобальную сеть.
- Программно-определяемая глобальная сеть — это технология, в которой используется множество сетевых коммутаторов и маршрутизаторов для виртуального доступа к устройству от аппаратного обеспечения до программного обеспечения, развернутого в «белой коробке». Конфиденциальные устройства и данные устанавливаются в главном офисе филиала или клиенте и получают специальный доступ администратора для доступа к его расширенным сетевым функциям, программному обеспечению для оптимизации облачных вычислений и брандмауэрам. Облачные сети — это огромный спектр массивов с сетевыми функциями, развернутыми на облачной платформе.
- Работа в сети обеспечивает производительность и безопасность в мультиоблачной среде, поэтому информационные технологии становятся более наглядными, предоставляя конечным пользователям необходимые требования и опыт. Рабочие нагрузки распределяются между облачной средой с использованием программного обеспечения в качестве приложения-службы. Защита

предоставляется пользователю для доступа к веб-странице и инфраструктуре путем перемещения приложений в облако с использованием типичной модели безопасности. Шлюз предоставляет код контекстного доступа и многослойный межсетевой экран. Приложения и сервисы предоставляются распределенным центрам обработки данных в облачной среде.

- Программно-определяемая широкополосная сеть предоставляет традиционный метод балансировки нагрузки и объединяет все диапазоны сети для удобства пользователя. Это дает больше визуальности с помощью интеллектуальной аналитики. Предоставление решений каждому облачному пользователю может быть затруднено, но использование всех услуг и предоставление SD-WAN уникального решения для сквозных приложений.

Преимущества облачных сетей

- **Высокая масштабируемость:** запрашивается поставка ресурсов в огромных масштабах без какого-либо вмешательства человека с каждым поставщиком услуг.
- **Высокая доступность и надежность:** серверы доступны в нужное время без каких-либо задержек или разочарований.
- **Ловкость:** эффективно распределяет ресурсы среди пользователей и работает быстро.
- **Совместное использование:** благодаря распределенным вычислениям различные клиенты из разных областей совместно используют одни и те же ресурсы благодаря базовой инфраструктуре.
- **Обслуживание:** это удобно для пользователя, так как к ним легко получить доступ с места и не требует установки.
- **Низкая стоимость:** это очень рентабельно и оплачивается в зависимости от его использования.
- **Услуги по плате за использование Модель** : интерфейс прикладного программирования предоставляется клиентам для использования ресурсов и услуг и оплаты на основе услуг.
- **Самообслуживание по требованию.** Облачные вычисления предоставляют клиенту необходимые услуги и приложения. С ключом входа они могут начать использовать без какого-либо человеческого взаимодействия и поставщиков облачных услуг. Он включает в себя хранилище и виртуальные машины.

Сложности при работе с сетями, использующими облачные сервисы

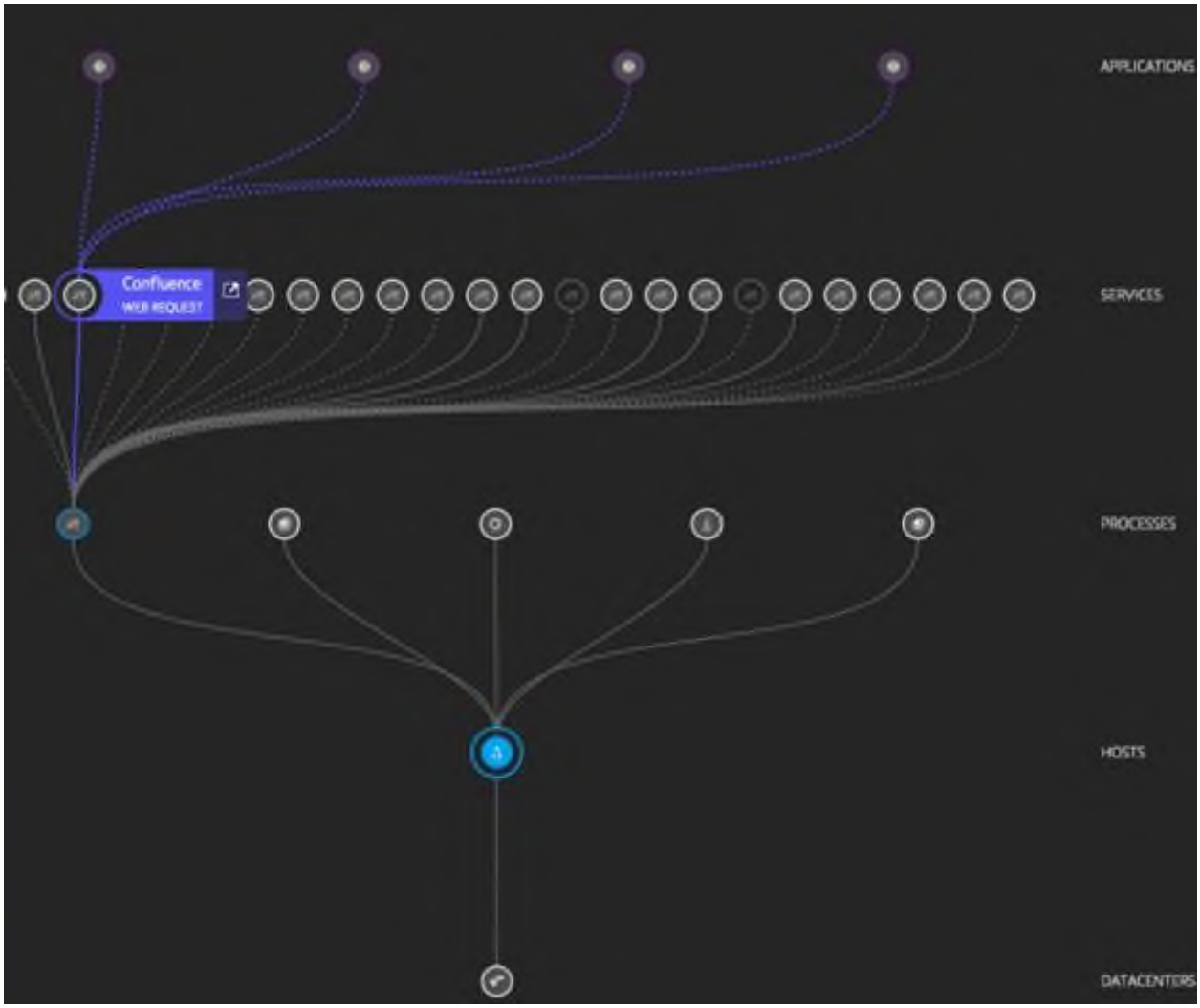
Из-за отсутствия непосредственного доступа к сети (уровни 1–2 модели OSI) командам, придерживающимся принципов DevOps, довольно сложно за ней следить. Они могут воспользоваться средствами для мониторинга, предлагаемыми облачными провайдерами, например, Cloud Watch, считывающим такие показатели работы сети, как NetworkIn и NetworkOut, однако этих показателей может быть недостаточно для выявления проблем в сети.

Ниже перечислены некоторые из основных сложностей, возникающих при использовании методологии DevOps для поддержания производительности виртуальных сетей:

- Распределение сетевых ресурсов между конкурирующими процессами (например, проблема, известная под названием TCP Incast);
- Изменяющаяся сетевая инфраструктура при наличии новых или приостановленных объектов;
- Масштабируемость сети при помощи сетевых интерфейсов ENI [*англ. Elastic Network Interfaces*];
- Качество внутренних соединений дата-центра;
- Качество соединений с частными сетями, расположенными вне дата-центра.

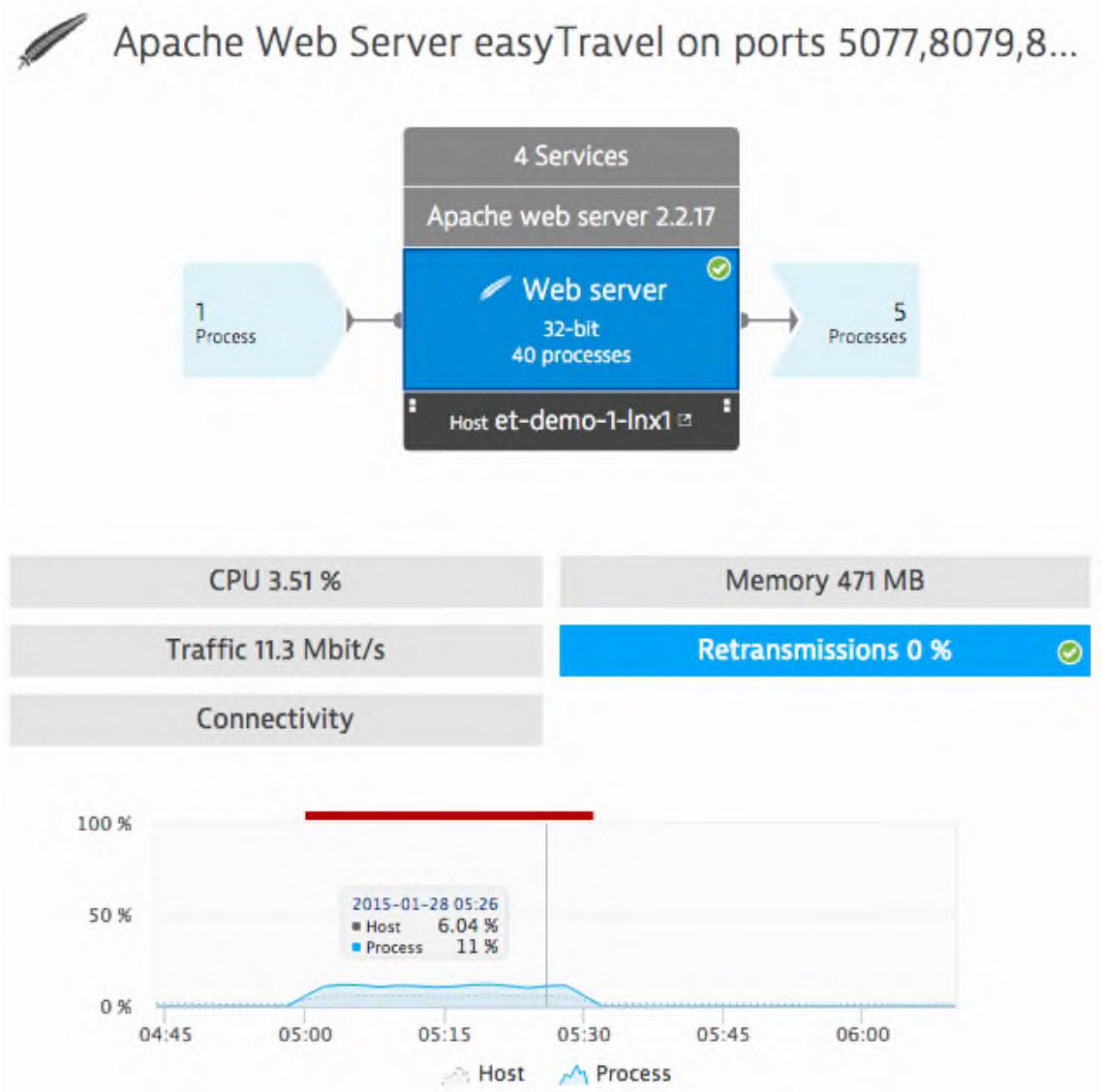
Мониторинг использования сети

При осуществлении мониторинга сети вы должны уметь адаптироваться к изменениям в инфраструктуре, упомянутым выше. В частности, нужно уметь работать с виртуальными сетевыми интерфейсами. В связи с этим мониторинг необходимо осуществлять на своих хостах, и при этом постоянно следить за изменениями виртуальной инфраструктуры. В этом случае вы можете отслеживать сетевые соединения между процессами, связанными с другими процессами и службами, таким образом осуществляя мониторинг фактического использования сети, не только сетевых устройств.



Мониторинг ресурсов очень важен… и прост!

При таком подходе к мониторингу ваша сеть не будет рассматриваться лишь как совокупность сетевых интерфейсов, таблиц маршрутизации и групп безопасности. Вместо этого ваша сеть будет рассматриваться как ограниченный ресурс, используемый процессами и приложениями. С точки зрения процессов, этот ресурс поддается мониторингу наряду с центральным процессором, оперативной и внешней памятью, и даже может быть оценен количественно. Кроме этого, проводится мониторинг производительности полностекового приложения и появляется возможность обнаружения неполадок в сети вплоть до уровня приложений.



Ниже представлены несколько основных показателей производительности сети, которые следует учитывать:

Основным показателем производительности сети является трафик данных в сети (пропускная способность).

- Показатель сетевого взаимодействия определяет долю успешно завершенных соединений по протоколу TCP и свидетельствует о доступности служб. Соединение по протоколу TCP может быть прервано или завершено с тайм-аутами, поэтому отсутствие подключения является ярким признаком возникновения проблем между отправителем и получателем в сети.
- При определении качества установленных соединений по протоколу TCP также следует обратить внимание на частоту повторной передачи данных. Назначением протокола TCP является надежное соединение и обнаружение ошибок при передаче данных. Это означает, что получатель должен дать согласие на получение пакетов данных, пересылаемых по сетевой линии связи; в противном случае они считаются потерянными и затем повторно пересылаются отправителем. Таким образом, частота повторной передачи

данных указывает на наличие слабых звеньев в сети и перегруженность ее инфраструктуры.

Виртуализированные сети не поддаются более или менее традиционным методам управления. За ними нужно следить, по крайней мере, с точки зрения ваших хостов и процессов, чтобы у вас имелись эффективные показатели производительности сети.

3. Вычислительные мощности или облачные вычисления.

Проще говоря, облачные вычисления — это предоставление вычислительных служб (в том числе серверов, хранилища, баз данных, сетей, программного обеспечения, аналитики и интеллектуального анализа) через Интернет («облако»). Такие службы ускоряют внедрение инноваций, повышают гибкость ресурсов и обеспечивают экономию благодаря высокой масштабируемости. Вы обычно платите только за облачные службы, которые позволяют сократить эксплуатационные расходы, а также повысить эффективность управления инфраструктурой и масштабирования по мере изменения потребностей бизнеса.

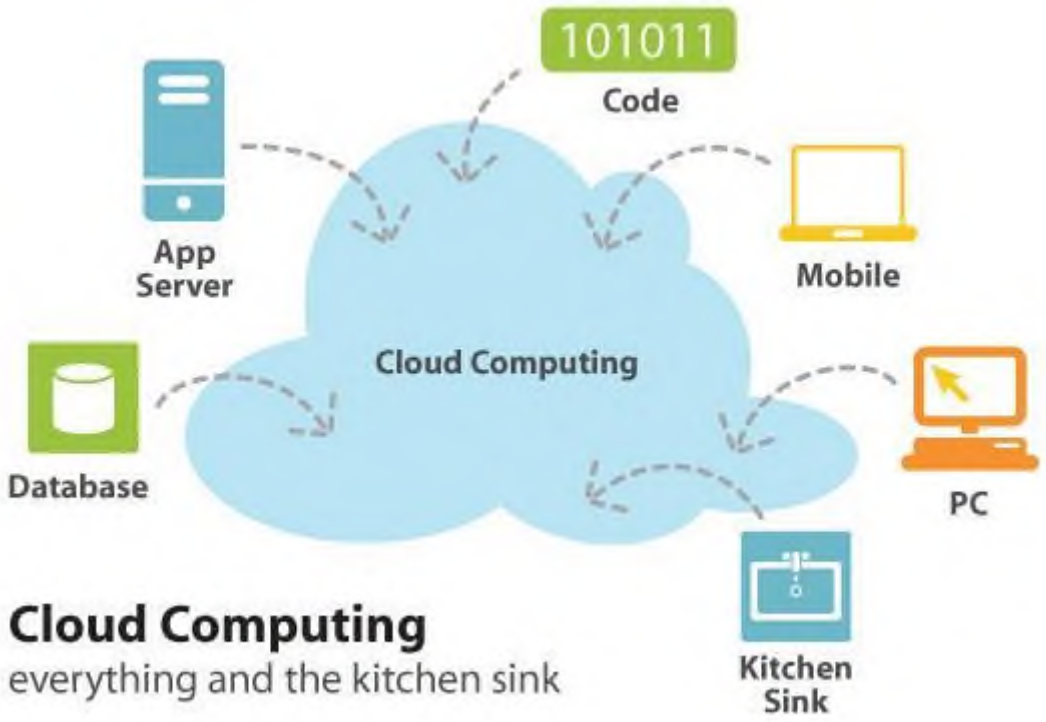
Развитие облачных вычислений

Концепция «облачных вычислений» зародилась в 1960 году, когда Джон Маккарти (Джон Маккарти — американский информатик, автор термина «искусственный интеллект», изобретатель языка Лисп, основоположник функционального программирования, лауреат премии Тьюринга за огромный вклад в область исследований искусственного интеллекта. Член Национальной академии наук США.) высказал предположение, что когда-нибудь компьютерные вычисления будут производиться с помощью «общенародных утилит».

Облачные вычисления могут показаться относительно новым явлением. Тем не менее, их история уходит корнями в начало 1950-х, когда появление мейнфреймов позволило нескольким пользователям получить доступ к центральному компьютеру. В 1960-х появились некоторые идеи, напоминающие то, что сегодня мы называем облачными вычислениями – например, концепция «межгалактической компьютерной сети» Дж. К. Р. Ликлайдера.

Идеология облачных вычислений получила популярность в 2007 году благодаря быстрому развитию каналов связи и растущей в геометрической прогрессии потребности как бизнеса, так и частных пользователей, в горизонтальном масштабировании своих информационных систем.

В 1970-е годы виртуализация подняла мейнфреймы на новый уровень, а в 1990-е телекоммуникационные компании начали предлагать подключение к виртуальной частной сети (VPN). В 1999 году Salesforce стала первой компанией, предоставляющей корпоративные приложения через Интернет. Несколько пользователей могли одновременно загружать эти приложения в браузере по невысокой цене.



Позже концепцию «вычислительного облака» начинают активно применять различные компании, например, Google. Наиболее характерный пример — служба Google Docs, позволяющая работать с офисными документами через браузер.

Современные «облака» появились в 2006 году, когда Amazon.com, в то время книжный интернет-магазин, представил Amazon Web Services (AWS), положив начало движению облачных вычислений. AWS предоставляет широкий набор сервисов, таких как вычислительные мощности и хранилища данных, по сей день оставаясь ведущей и очень надежной инфраструктурой платформ облачных веб-сервисов.

Скоро к Amazon.com присоединились Netflix, Microsoft, Google, Apple и IBM, и рынок облачных вычислений разросся.

В июле 2008 года корпорации HP, Intel, и Altaba (ранее Yahoo) объявили о создании глобальной, охватывающей множество площадок, открытой вычислительной лаборатории Cloud Computing Test Bed для развития исследований и разработок в области cloud computing. Данная лаборатория представляла собой глобально распределенную испытательную интернет-среду, которая поддерживала исследования, направленные на развитие ПО, совершенствование управления центрами обработки данных и решение аппаратных проблем, связанных с интернет-вычислениями гораздо большего масштаба, чем когда-либо раньше. Данная инициатива также должна была способствовать появлению новых интернет-приложений и услуг.

Не осталась в стороне и Microsoft: исполнительный директор корпорации Стив Баллмер сделал заявление о том, что Microsoft планирует выпустить новую операционную систему под кодовым названием «Windows Cloud», которая позволит разработчикам создавать и размещать интернет-приложения. Название «Windows Cloud» указывает на то, что новая ОС будет использовать в своей основе концепцию «вычислительного облака».

На 2014 год большинство крупнейших ИТ-вендоров на мировом рынке, включая Google, Microsoft, HP, Intel, SAP, IBM, Oracle и другие, имеют в своей линейке решения cloud computing.

Типы облачных служб: IaaS, PaaS, бессерверные службы и SaaS

Большинство служб облачных вычислений подразделяются на три общие группы: инфраструктура как услуга (IaaS), платформа как услуга (PaaS) и программное обеспечение как услуга (SaaS). Такие службы иногда называют облачным вычислительным стеком, потому что они накладываются одна на другую. Если вы узнаете, что они из себя представляют и чем друг от друга отличаются, то вам будет проще реализовать поставленные бизнес-цели.

- **Инфраструктура как услуга (IaaS)**

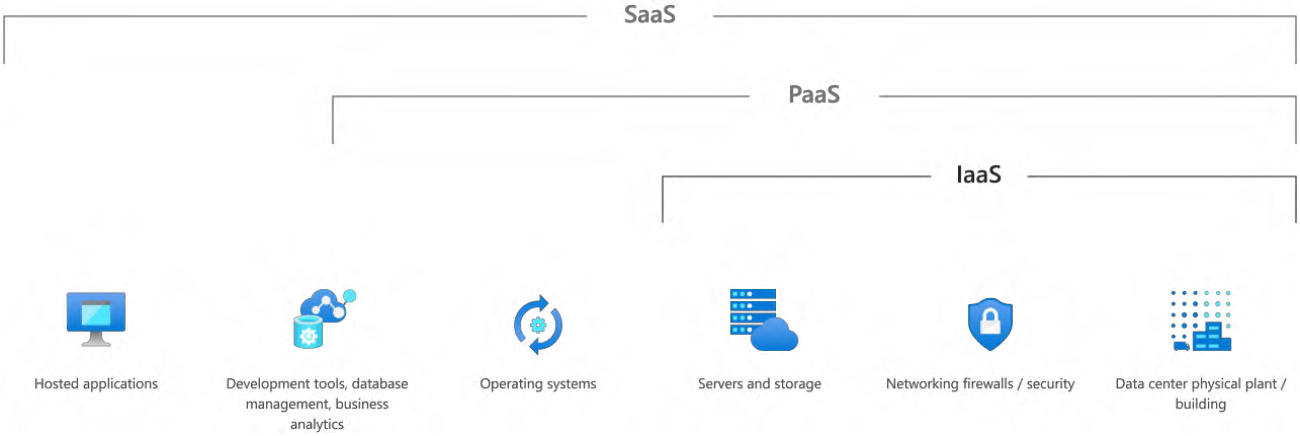
Основная группа облачных вычислительных служб. В схеме IaaS вы арендуете ИТ-инфраструктуру (серверы, виртуальные машины, хранилище, сети и операционные системы) у облачного поставщика с системой оплаты по мере использования.

Инфраструктура как услуга (IaaS) — это разновидность службы облачных вычислений, предоставляющая основные ресурсы для вычислений, хранения и сетевого подключения по запросу с оплатой по мере использования. IaaS является одним из четырех типов облачных служб, к которым также относятся программное обеспечение как услуга (SaaS), платформа как услуга (PaaS) и бессерверные вычисления.

Миграция инфраструктуры организации в решение IaaS поможет сократить затраты на обслуживание локальных центров обработки данных, сэкономить на оборудовании и получить сведения бизнес-аналитики в реальном времени. Решения IaaS позволяют гибко масштабировать ИТ-ресурсы в соответствии с увеличением и уменьшением спроса. Они также помогают быстро подготавливать новые приложения к работе и повышать надежность базовой инфраструктуры.

IaaS избавляет от затрат и сложностей, связанных с приобретением физических серверов и инфраструктуры центров обработки данных и управлением ими. Каждый ресурс предоставляется как отдельный компонент

службы, а вы платите за конкретный ресурс, только пока нуждаетесь в нем. Поставщик служб облачных вычислений, например, Azure, управляет инфраструктурой, а вы приобретаете, устанавливаете и настраиваете собственное программное обеспечение (включая операционные системы, ПО промежуточного слоя и приложения), а также управляете им.



- **Платформа как услуга (PaaS)**

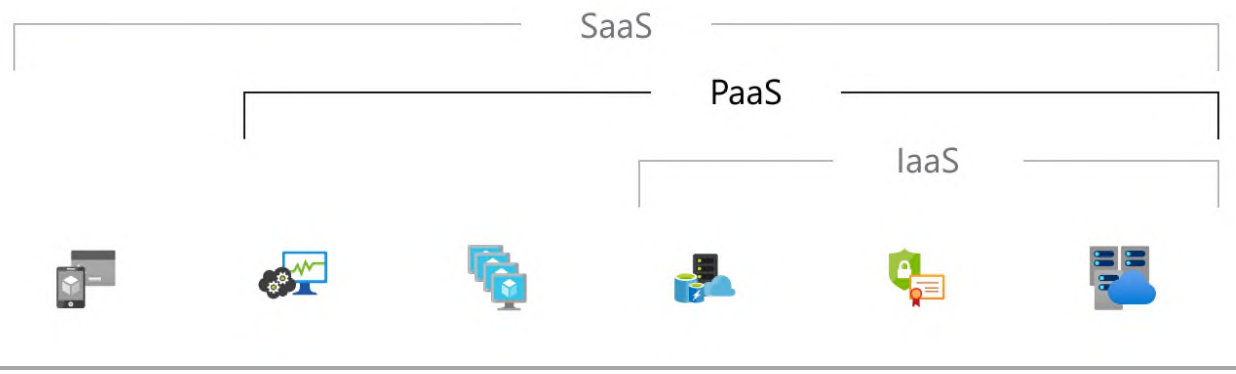
Платформа как услуга относится к облачным вычислительным службам, которые поставляют среду, доступную по запросу, для разработки, тестирования, доставки приложений программного обеспечения и управления ими. PaaS упрощает разработчикам задачу быстрого создания веб-приложений или мобильных приложений без необходимости иметь дело с базовой инфраструктурой серверов, хранилища, сети и баз данных, необходимых для разработки.

Платформа как услуга (PaaS) — это полноценная среда разработки и развертывания в облаке с ресурсами, которые позволяют предоставлять любые приложения, от простых облачных приложений до продвинутых облачных приложений промышленного класса. Вы приобретаете необходимые ресурсы у поставщика облачных служб, платите по мере использования и подключаетесь к ним по безопасному интернет-подключению.

Как и в случае с IaaS, PaaS включает инфраструктуру (серверы, хранилище и сетевое оборудование), а также ПО промежуточного слоя, средства разработки, бизнес-аналитику (BI), службы системы управления базами данных и другое. Услуга PaaS предназначена для поддержки полного жизненного цикла веб-приложения: разработки, тестирования, развертывания, управления и обновления.

PaaS также позволяет избежать затрат и трудностей, связанных с приобретением лицензий на программное обеспечение, базовой

инфраструктуры приложений, ПО промежуточного уровня, оркестраторов контейнеров, например Kubernetes, или средств разработки и других ресурсов, а также управлением ими. Вы управляете приложениями и службами, которые разрабатываете, а поставщик облачных служб обычно управляет всем остальным.



Размещенные приложения	Средства разработки, управление базами данных, бизнес-аналитика	Операционные системы	Серверы и хранилище	Сетевые брандмауэры / защита	Материальная база / здание центра обработки данных

Распространенные сценарии PaaS

Обычно организации используют PaaS в следующих случаях.

Среда для разработки. PaaS предоставляет среду, которую разработчики используют для разработки или настройки облачных приложений. Аналогично тому, как создается макрос для Excel, PaaS позволяет разработчикам создавать приложения с использованием встроенных компонентов программного обеспечения. Облачные функции, такие как масштабирование, высокая доступность и поддержка нескольких пользователей, уже включены и снижают объем кода, который необходимо разрабатывать.

Бизнес-аналитика. Средства, предоставляемые в рамках PaaS, позволяют организациям анализировать данные, находить тенденции и делать прогнозы с целью улучшения планирования, решений по продуктам, повышения возврата от инвестиций и принятия других бизнес-решений.

Дополнительные службы. Поставщики PaaS могут предлагать другие службы, которые повышают возможности приложений, такие как рабочие процессы, каталоги, безопасность и планирование.

Преимущества PaaS

Предоставляя инфраструктуру как услугу, PaaS предлагает те же преимущества, что и IaaS. Однако дополнительные компоненты (ПО промежуточного слоя, средства разработки и другие бизнес-средства) создают следующие дополнительные преимущества.

Сокращение времени программирования. Средства разработки PaaS могут сократить время, необходимое для программирования новых приложений благодаря заранее подготовленным компонентам, встроенным в платформу, включая рабочие процессы, службы каталогов, компоненты безопасности, средства поиска и т. д.

Добавление возможностей разработки без увеличения числа сотрудников. Компоненты платформы как услуги предоставляют вашей команде разработчиков новые возможности без необходимости нанимать сотрудников с соответствующими навыками.

Упрощенная разработка для нескольких платформ, включая мобильные платформы. Некоторые поставщики служб предоставляют вам варианты разработки для нескольких платформ, например компьютеров, мобильных устройств и браузеров, упрощая и ускоряя таким образом кроссплатформенную разработку приложений.

Экономичное использование продвинутых средств. Оплата по мере использования позволяет физическим и юридическим лицам использовать продвинутые средства разработки и бизнес-аналитики, которые могут быть слишком дорогими для приобретения в собственность.

Поддержка географически распределенных команд разработчиков. Поскольку доступ к среде разработки осуществляется через Интернет, команда разработчиков может работать над одними проектами, даже когда члены команды находятся в разных местах.

Эффективное управление жизненным циклом приложений. PaaS обеспечивает все возможности, которые потребуются для поддержки полноценного жизненного цикла веб-приложений: создания, тестирования, развертывания, управления и обновления внутри одной интегрированной среды.

- **Бессерверные вычисления**

Бессерверные вычислительные ресурсы применяются совместно с PaaS. Это позволяет сосредоточиться на создании функциональности приложения, не тратя время на постоянное управление необходимыми серверами и инфраструктурой. Поставщик облачных служб выполняет настройку, планирование емкости и управление сервером за вас. Бессерверные

архитектуры работают на основе событий, обладают высоким уровнем масштабируемости и используют ресурсы, только когда задействуется конкретная функция или происходит определенное событие.

Что такое бессерверные вычисления?

Бессерверные вычисления позволяют разработчикам быстрее создавать приложения, устраняя необходимость в управлении инфраструктурой. Благодаря бессерверным приложениям поставщики облачных служб могут автоматически подготовить, масштабировать и администрировать инфраструктуру, необходимую для выполнения кода.

Говоря об определении бессерверных вычислений, важно отметить, что серверы по-прежнему работают с кодом. Слово «бессерверный» в этом контексте означает, что задачи, связанные с подготовкой и администрированием инфраструктуры, выполняются незаметно для разработчика. Такой подход позволяет разработчикам сосредоточиться на бизнес-логике и повысить рентабельность решения. Бессерверные вычисления помогают командам повышать производительность и ускорять вывод продуктов на рынок. Кроме того, с помощью этих технологий организации могут лучше оптимизировать ресурсы и сосредоточиться на внедрении инноваций.

Основные преимущества бессерверных вычислений

	Забудьте об управлении инфраструктурой Полностью управляемые службы позволяют разработчикам не беспокоиться об администрировании и сосредоточиться на основной бизнес-логике. Благодаря бессерверной платформе вы просто развертываете код, и он выполняется с высоким уровнем доступности.		Динамическая масштабируемость Бессерверные вычисления позволяют динамически увеличивать и уменьшать масштаб инфраструктуры в считаные секунды в соответствии с требованиями любой рабочей нагрузки.
	Ускоренный вывод на рынок Бессерверные приложения снижают уровень зависимости операций в каждом цикле развертывания, расширяя гибкие возможности команд, чтобы они могли ускорить разработку решений и повысить их функциональность.		Более эффективное использование ресурсов Переход на бессерверные технологии помогает организациям уменьшить совокупную стоимость владения и перераспределить ресурсы, чтобы ускорить внедрение инноваций.

Шаблоны бессерверных приложений

Разработчики создают бессерверные приложения на основе разнообразных шаблонов, во многих из которых задействуются привычные подходы, чтобы обеспечить соответствие определенным требованиям и бизнес-потребностям.

Бессерверные функции

Бессерверные функции ускоряют разработку с помощью модели на основе событий. В такой модели реализованы триггеры, которые автоматически выполняют код при возникновении событий и добавлении привязок, чтобы обеспечить простую интеграцию дополнительных служб. В рамках модели оплаты каждого выполнения тарификация занимает менее секунды. При этом вы платите только за время и ресурсы, требуемые для выполнения кода.

Бессерверная служба Kubernetes

Разработчики могут использовать собственные контейнеры в полностью управляемых кластерах, которые оркестрирует Kubernetes. Масштаб таких кластеров можно автоматически увеличивать и уменьшать при внезапных изменениях объема трафика, свойственных нестабильным рабочим нагрузкам.

Бессерверные рабочие процессы

В бессерверных рабочих процессах задействуется подход с незначительным использованием кода или без его использования, чтобы упростить оркестрацию комбинированных задач. Разработчики могут интегрировать различные службы (как облачные, так и локальные) без необходимости писать код для такого взаимодействия, поддерживать код прилипания или изучать новые API-интерфейсы либо спецификации.

Среды бессерверных приложений

Бессерверная среда для приложений обеспечивает размещение интерфейсной и серверной части в полностью управляемых службах, которые отвечают за масштабирование, безопасность и соответствие требованиям.

Бессерверный шлюз API

Бессерверный шлюз API — это централизованная и полностью управляемая точка входа для бессерверных внутренних служб. Она позволяет разработчикам администрировать, защищать и анализировать API-интерфейсы в глобальном масштабе.

- **Программное обеспечение как услуга (SaaS)**

Программное обеспечение как услуга (SaaS) — это метод доставки программного обеспечения для приложений через Интернет по запросу и

обычно на основе подписки. В схеме SaaS облачные поставщики размещают программное обеспечение и базовую инфраструктуру и управляют ими, а также занимаются всем обслуживанием, включая обновление программного обеспечения и установку исправлений безопасности. Пользователи подключаются к приложению по Интернету, обычно с помощью веб-браузера на своем телефоне, планшете или ПК.

Преимущества SaaS

Получите доступ к продвинутым приложениям. Чтобы предоставить приложения SaaS пользователям, вам не нужно приобретать, устанавливать, обновлять или обслуживать оборудование, ПО промежуточного слоя или программное обеспечение. Если у организаций нет ресурсов для приобретения, развертывания необходимой инфраструктуры и программного обеспечения таких сложных промышленных приложений, как ERP или CRM, а также для управления ими, то можно сделать их доступными через SaaS.

Вы платите только за те ресурсы, которые используете. Вы экономите средства, так как SaaS автоматически изменяет масштабирование в соответствии с уровнем использования.

Используйте бесплатное клиентское программное обеспечение. Пользователи могут запускать большинство приложений SaaS напрямую через веб-браузер без необходимости загружать и устанавливать программное обеспечение, хотя для некоторых приложений требуются подключаемые модули. Это означает, что вам не нужно приобретать и устанавливать специальное программное обеспечение для ваших пользователей.

Повышайте мобильность ваших сотрудников. SaaS повышает мобильность ваших сотрудников, так как пользователи имеют доступ к приложениям и данным SaaS с любого компьютера или мобильного устройства, которые подключены к Интернету. Не нужно заботиться о разработке приложений для работы на разных типах компьютеров и устройств, потому что поставщик служб уже сделал это за вас. Кроме того, нет необходимости нанимать квалифицированных специалистов для решения проблем с безопасностью, которые могут возникнуть при использовании мобильных устройств. Правильно выбранный поставщик служб обеспечивает безопасность ваших данных, независимо от типа устройства, которое использует эти данные.

Получайте доступ к данным приложений из любого места. Когда данные хранятся в облаке, пользователи получают доступ к нужной информации с любого компьютера или мобильного устройства, подключенных к Интернету. Если вы храните данные приложения в облаке,

данные не могут быть потеряны в результате сбоя компьютера или устройства.

4. Контексты безопасности. Менеджеры секретов

Что такое секрет?

Проще говоря, секрет — это цифровые учетные данные аутентификации.

Самый известный пример секрета — это пароль.

Секреты — это индивидуально названные наборы конфиденциальной информации, которые относятся к широкому спектру защищенных данных. Существует много видов секретов, включая пароли пользователей, пароли приложений и баз данных, автоматически сгенерированные ключи шифрования, закрытые ключи шифрования, ключи API, ключи приложений, ключи SSH, токены авторизации и частные сертификаты (например, TLS, SSL). Каждый тип секрета хорошо подходит для определенного использования, будь то хранение информации в состоянии покоя или в пути, или предоставление доступа к конфиденциальным, критически важным для бизнеса ИТ-ресурсам.

Наиболее известным примером секрета является пароль, используемый для входа в приложение, веб-сайт или конечную точку. Но эти типы секретов не самые распространенные и не самые сложные для защиты.

Современные приложения, размещенные в облаке или локально, ускорили потребность во всех типах цифровых секретов. Секреты между приложениями растут в геометрической прогрессии. Они используются для шифрования данных при их передаче между приложениями, например, при отправке информации с веб-страницы, выполнении безопасного запроса к API, доступе к облачной базе данных и многих других случаях, с которыми сталкиваются современные предприятия, стремящиеся к цифровой трансформации и повышению уровня автоматизации.

При таком количестве типов секретов, используемых в стольких контекстах, легко потерять их из виду или последовательно применять на предприятии. Вот тут-то и появляется тайное управление.

Что такое управление секретами?

Управление секретами встраивает элементы управления безопасностью в наиболее рискованные уровни инфраструктуры современной организации: облака, код, данные и устройства.

Жизненный цикл секретов

- **Генерация/создание:** Секрет создается вручную пользователем или автоматически по мере необходимости. Пароли часто, но не всегда, должны соответствовать политике, регулирующей их создание и использование. Автоматическое создание учетных данных устраняет один из ключевых недостатков управления секретами: учетные данные, созданные человеком, как правило, легче взломать, чем созданные компьютером.

- **Ротация:** после использования секрет следует регулярно менять. Это часто указывается и требуется различными стандартами, который требует максимального 90-дневного цикла ротации. Это может быть автоматическая регенерация нового секрета по расписанию или запрос на создание вручную. Если секрет устарел или срок его действия истек, доступ будет запрещен до тех пор, пока он не будет обновлен.
- **Отзыв** . Возможность удалить учетные данные пользователя или приложения, тем самым лишив доступа к ресурсу, настолько важна, что закреплена во многих стандартах политики безопасности. Тайный отзыв может использоваться, когда сотрудник покидает компанию или обнаруживается аномальное поведение. Удаление ненужных, просроченных, нарушенных или слабых секретов является важным шагом в хорошей гигиене секретов.

Чтобы управлять секретами на протяжении всего их жизненного цикла, любая система, которую вы выберете, должна иметь возможность хранить и обслуживать секреты, когда это необходимо, а также обеспечивать видимость и контроль на основе политик от начала до конца.

Почему управление секретами важно?

Управление секретами позволяет безопасно хранить, передавать и проверять секреты. Это устраняет или, по крайней мере, сводит к минимуму участие людей в управлении секретами для уменьшения потенциальных точек отказа. Такой системный подход к предотвращению несанкционированного доступа к конфиденциальным данным и системам помогает избежать утечек данных, кражи или манипулирования данными и личными данными.

Строгая политика секретности помогает смягчить эти распространенные проблемы:

- Делитесь секретами с другими пользователями
- Повторное использование секретов либо из соображений целесообразности, либо потому, что пароли жестко закодированы или встроены в приложения и системы.
- Слабое хранилище секретов, например, хранение секретов в незашифрованном виде или в виде простого текста.
- Нет ротации секретов, что может быть побочным эффектом наличия жестко закодированных или встроенных секретов.
- Нет секретного отзыва

Управление секретами предотвращает разрастание секретов

По мере увеличения объема, разнообразия и сложности ваших ИТ-систем становится все труднее внедрять и управлять последовательной политикой для всех них, а также знать, где находятся секреты и как они используются. Расползание секретов — это коварное состояние, при котором

организация теряет след своих учетных данных, поддавшись лоскутному одеялу систем управления, каждая из которых имеет свою собственную политику управления. Существует множество способов управления секретами, и когда у каждого приложения, облачного провайдера или подразделения организации есть собственная модель безопасности, организация в целом теряет видимость.

Политики секретов должны устанавливаться и управляться централизованно, чтобы обеспечить их последовательное применение, определяя правила обработки секретов на каждом этапе их жизненного цикла.

Управление секретами и управление ключами

Secret Manager позволяет хранить, управлять и получать доступ к секретам в виде двоичных объектов или текстовых строк. Имея соответствующие разрешения, вы можете просматривать содержимое секрета.

Secret Manager хорошо подходит для хранения информации о конфигурации, такой как пароли базы данных, ключи API или сертификаты TLS, необходимые приложению во время выполнения.

Система управления ключами, позволяет управлять криптографическими ключами и использовать их для шифрования или расшифровки данных. Однако вы не можете просматривать, извлекать или экспортировать сам ключевой материал.

Точно так же вы можете использовать систему управления ключами для шифрования конфиденциальных данных перед их передачей или сохранением. Затем вы можете расшифровать конфиденциальные данные перед их использованием. Использование системы управления ключами для защиты секрета таким образом более сложно и менее эффективно, чем использование Secret Manager.

Секрет

Секрет — это глобальный объект проекта, который содержит набор метаданных и версий секрета. Метаданные могут включать места репликации, метки, аннотации и разрешения. В секретных версиях хранятся фактические секретные данные, такие как ключ API или учетные данные.

Версия

Секретная версия хранит фактические секретные данные, такие как ключи API, пароли или сертификаты.

Вы можете обращаться к отдельным версиям секрета. Вы не можете изменить версию, но можете удалить ее.

Ротация

Вы меняете секрет, добавляя к секрету новую версию секрета. Можно получить доступ к любой версии данного секрета, если эта версия включена. Чтобы предотвратить использование секретной версии, вы можете отключить эту версию.

Невозможно запланировать автоматическую ротацию секрета.

Контент безопасности на примере технологической платформы ПАО «Ростелеком»

Основные понятия

Ниже перечислены основные понятия с указанием соответствующих им параметров в модуле decs_vm.

1. Контроллер облачной инфраструктуры DECS – управляющее приложение, которое обеспечивает авторизацию пользователей и оркестрацию облачных ресурсов.
   - Адрес контроллера задается в обязательном параметре controller_url. Например, controller_url: https://ds1.digitalenergy.online
2. Авторизационный провайдер – приложение, работающее по протоколу Oauth2, предназначенное для выпуска и валидации токенов доступа к контроллеру облачной инфраструктуры.
   - Адрес авторизационного провайдера задается в параметре oauth2_url. Например oauth2_url: https://sso.digitalenergy.online
3. Подписчик (*tenant*) – сущность, которая используется для группирования облачных ресурсов по принадлежности к определенному клиенту для целей учета потребления и биллинга.
   - Имя подписчика задается параметром tenant.
4. Пользователь (*user*) – пользователь облачной инфраструктуры, представленный учетной записью. Чтобы получить возможность управлять облачными ресурсами (например, создавать виртуальные сервера) пользователь должен быть ассоциирован с одним или несколькими подписчиками и иметь соответствующие права, определяемые ролевой моделью, принятой в облачной платформе DECS. Для доступа к платформе пользователь должен авторизоваться одним из способов, описанных в разделе «Способы авторизации».
5. Виртуальный сетевой сегмент – защищенный сетевой сегмент, служащий для сетевого подключения виртуальных серверов и группирования их по функциональному признаку. Данный сегмент можно рассматривать как небольшой персональный дата-центр (virtual data center, VDC), в котором размещаются группы виртуальных серверов. Виртуальный сетевой сегмент идентифицируется одним из двух способов:

- По уникальному в рамках каждого контроллера DECS цифровому идентификатору (параметр vdc_id).
- По комбинации параметров tenant и vdc_name. Обратите внимание, что имя vdc_name уникально только в рамках одного и того же tenant.

6. Виртуальный сервер – виртуальная машина, которая работает в облаке DECS и доступна по сети. Виртуальный сервер характеризуется количеством выделенных ему CPU (параметр cpu), объемом ОЗУ (ram), размером загрузочного диска (bootdisk.size). При создании виртуального сервера на загрузочный диск устанавливается образ операционной системы, заданный в параметре image_name. Помимо загрузочного диска к виртуальному серверу можно подключить несколько дисков для хранения прикладных данных, конфигурация которых задается параметром data_disks. Виртуальный сервер идентифицируется одним из двух способов:
   - По уникальному в рамках каждого контроллера DECS цифровому идентификатору сервера (параметр id).
   - По комбинации параметров name, tenant и vdc_name (или vdc_id). Обратите внимание, что имя виртуального сервера name уникально только в рамках одного и того же виртуального сетевого сегмента.
7. Сетевой доступ к виртуальному серверу возможен через настройку правил трансляции сетевых портов на периметре соответствующего виртуального сетевого сегмента (параметр port_forwarding), а также посредством назначения серверу прямого внешнего IP адреса (параметры ext_network и ext_network_id).

Способы авторизации

На текущий момент облачная платформа DECS поддерживает два базовых типа авторизации:

1. С использованием авторизационного провайдера, работающего по протоколу Oauth2. Данный способ является предпочтительным, так как обеспечивает бОльшую гибкость и безопасность. Для авторизации в этом режиме в task необходимо указать параметры oauth2_url и controller_url, а также предоставить одно из нижеперечисленного:
   - Комбинация Application ID & Application secret, соответствующих пользователю, от имени которого будет осуществляться управление облачными ресурсами в текущей сессии. В процессе проверки предоставленных модуль получает от авторизационного провайдера токен (JSON Web Token), который затем используется для доступа к указанному DECS контроллеру. Чтобы модуль decs_vm авторизовался по данному варианту, в task следует установить параметр authenticator: oauth2 и задать параметры app_id и app_secret..

- JSON Web Token – заранее полученный от авторизационного провайдера токен доступа, ассоциированный с определенным пользователем, от имени которого будет осуществляться управление облачными ресурсами в текущей сессии. Чтобы модуль decs_vm авторизовался по данному варианту, в task следует установить параметр authenticator: jwt и задать параметр jwt..

2. С использованием комбинации user:password. Данный режим не использует внешних авторизационных провайдеров и подразумевает, что пользователь с такой комбинацией user:password зарегистрирован непосредственно на указанном в параметре controller_url контроллере облачной инфраструктуры DECS.

- Чтобы модуль decs_vm авторизовался по данному варианту, в task следует установить параметр authenticator: legacy и задать параметры user и password.

После успешной авторизации пользователь (*user*) получает доступ к ресурсам, находящимся под управлением соответствующего DECS контроллера. Доступ предоставляется в рамках подписчиков (*tenant*), с которыми ассоциирован данный пользователь, и в соответствии с присвоенными ему ролями.

Как выглядит task для разных режимов авторизации

1 Режим *oauth2* с использованием *Application ID* & *Application Secret*

Данный режим задается установкой параметра authenticator: oauth2

В этом режиме необходимо указать параметры app_id и app_secret (либо задать соответствующие значения в переменных окружения *DECS_APP_ID* и *DECS_APP_SECRET*).

Чтобы воспользоваться этим режимом, вы должны заранее создать *Application ID* & *Application Secret* под своей учетной записью на OAuth2 сервере, который используется для авторизации в вашем DECS контроллере. Вы можете сделать это самостоятельно в соответствующем разделе личного кабинета на OAuth2 сервере.

Помимо вышеуказанных параметров также необходимо задать oauth2_url и controller_url. В итоге *task* будет выглядеть следующим образом:

```
---
- hosts: ansible_master
  tasks:
  - name: manage VM in DECS cloud
    decs_vm:
      authenticator: oauth2
      app_id: "{{ my_app_id }}"
```

```
      app_secret: "{{ my_app_secret }}"
      oauth2_url: "https://sso.decs.online"
      controller_url: "https://cloud.digitalenergy.online"
    <<<дальнейшие детали опущены>>>
...
```

2 Режим *jwt* с использованием заранее полученного JSON Web Token (JWT)

Данный режим задается установкой параметра authenticator: jwt

Режим *jwt* может оказаться полезным, когда вы планируете создавать множество виртуальных машин (например, в асинхнронном режиме вызова модуля *decs_vm*, так как дает экономию по количеству API вызовов в адрес OAuth2 провайдера и контроллера облачной инфраструктуры.

Чтобы воспользоваться этим режимом вам сначала потребуется получить действительный JWT, например, от OAuth2 сервера, который используется для авторизации в вашем DECS контроллере. Для получения JWT можно воспользоваться вспомогательным модулем *decs_jwt*.

Помимо задания параметра jwt (либо сохранения JWT в переменной окружения *DECS_JWT*) также необходимо задать oauth2_url и controller_url. В итоге *task* будет выглядеть следующим образом:

```
---
- hosts: ansible_master
  tasks:
  - name: manage VM in DECS cloud
    decs_vm:
      authenticator: jwt
      jwt: "{{ my_jwt }}"
      oauth2_url: "https://sso.decs.online"
      controller_url: "https://cloud.digitalenergy.online"
    <<<дальнейшие детали опущены>>>
...
```

Режим *legacy* с использованием имени пользователя и пароля

Данный режим задается установкой параметра authenticator: legacy

В этом режиме необходимо указать параметры user и password (либо задать соответствующие значения в переменных окружения *DECS_USER* и *DECS_PASSWORD*).

Чтобы воспользоваться этим режимом вам потребуется учетная запись (*legacy user*), заданная непосредственно на вашем DECS контроллере. Для

создания такой учетной записи обратитесь к администратору облачной платформы.

Помимо вышеуказанных параметров также необходимо задать controller_url. В итоге *task* будет выглядеть следующим образом:

```
---
- hosts: ansible_master
  tasks:
  - name: manage VM in DECS cloud
    decs_vm:
      authenticator: legacy
      user: "{{ legacy_user_name }}"
      password: "{{ legacy_user_password }}"
      controller_url: "https://cloud.digitalenergy.online"
    <<<дальнейшие детали опущены>>>
...
```

5. Кластеры. Ресурсы под управлением облачным провайдером.

Что такое кластер?

На высоком уровне компьютерный кластер представляет собой группу из двух или более компьютеров или узлов, которые работают параллельно для достижения общей цели. Это позволяет распределять рабочие нагрузки, состоящие из большого количества отдельных, распараллеливаемых задач между узлами кластера. В результате эти задачи могут использовать объединенную память и вычислительную мощность каждого компьютера для повышения общей производительности.

Чтобы построить компьютерный кластер, отдельные узлы должны быть подключены к сети, чтобы обеспечить межкомпонентную связь. Программное обеспечение кластера компьютеров затем можно использовать для объединения узлов вместе и формирования кластера. Он может иметь общее устройство хранения и/или локальное хранилище на каждом узле. Как правило, по меньшей мере один узел обозначается как ведущий узел и выступает в качестве точки входа в кластер. Ведущий узел может отвечать за делегирование входящей работы другим узлам и, при необходимости, агрегирование результатов и возврат ответа пользователю.

В идеале кластер функционирует так, как если бы это была единая система. Пользователю, обращающемуся к кластеру, не нужно знать, является ли система кластером или отдельной машиной. Кроме того, кластер должен быть спроектирован таким образом, чтобы свести к минимуму задержку и предотвратить узкие места в обмене данными между узлами.

Типы кластерных вычислений

Компьютерные кластеры обычно можно разделить на три типа:

- Высокая доступность или отработка отказа
- Балансировка нагрузки
- Высокопроизводительные вычисления

Как вы увидите в следующем разделе, три типа кластеров тесно связаны с потенциальными преимуществами, которые предлагают кластеры. Если применимо, я сошлюсь на связанный тип кластера после объяснения конкретного преимущества и того, как кластер его предоставляет. Также важно отметить, что кластер может быть более одного из этих трех типов. Например, кластер, в котором размещен веб-сервер, скорее всего, будет как кластером высокой доступности, так и кластером балансировки нагрузки.

Четыре ключевых преимущества кластерных вычислений

Кластерные вычисления обеспечивают ряд преимуществ: высокая доступность благодаря отказоустойчивости и отказоустойчивости,

возможности балансировки нагрузки и масштабирования, а также повышение производительности. Давайте подробнее рассмотрим каждую из этих функций и рассмотрим, как кластеры обеспечивают их.

1. Высокая доступность

Есть несколько важных терминов, которые следует помнить при обсуждении надежности системы:

Доступность - доступность системы или услуги в течение определенного периода времени, обычно выражаемая в процентах от времени безотказной работы в течение данного года (например, доступность 99,999% или пять 9-х)

Устойчивость - насколько хорошо система восстанавливается после сбоя

Отказоустойчивость - способность системы продолжать предоставлять услугу в случае сбоя

Надежность - вероятность того, что система будет функционировать должным образом

Избыточность - дублирование критически важных ресурсов для повышения надежности системы

Приложение, работающее на одной машине, имеет единую точку отказа, что приводит к снижению надежности системы. Если машина, на которой размещено приложение, выходит из строя, почти всегда будет время простоя во время восстановления инфраструктуры. Поддержание уровня избыточности, который помогает повысить надежность, может сократить время недоступности приложения. Это может быть достигнуто путем упреждающего запуска приложения на второй системе (которая может или не может получать трафик) или наличия холодной системы (как в настоящее время не запущенной), предварительно настроенной с приложением. Эти конфигурации называются соответственно активно-активными и активно-пассивными конфигурациями. При обнаружении сбоя активно-активная система может немедленно переключиться на вторую машину, в то время как активно-пассивная система выполнит отработку отказа после того, как вторая машина будет работать.

Компьютерные кластеры состоят из нескольких узлов, выполняющих один и тот же процесс одновременно, и поэтому являются активно-активными системами. Активно-активные системы обычно отказоустойчивы, потому что система по своей сути предназначена для обработки потерь узла. В случае сбоя узла остальные узлы готовы взять на себя рабочую нагрузку неисправного узла. С учетом сказанного, кластер, который требует лидерного узла, должен запускать как минимум два ведущих узла в конфигурации «активный-активный». Это может предотвратить недоступность кластера в случае сбоя ведущего узла.

Помимо большей отказоустойчивости, кластеры могут повысить отказоустойчивость, упрощая повторное присоединение восстановленных узлов к системе и возврат кластера к оптимальному размеру. Любое количество простоев системы дорого обходится организации и может привести к ухудшению качества обслуживания клиентов, поэтому крайне важно, чтобы система была устойчивой и отказоустойчивой в случае сбоя. Использование кластера может повысить отказоустойчивость и отказоустойчивость системы, обеспечивая более высокую доступность. Кластеры с такими характеристиками называются кластерами с высокой доступностью или отказоустойчивостью.

2. Балансировка нагрузки

Балансировка нагрузки — это акт распределения трафика между узлами кластера для оптимизации производительности и предотвращения получения каким-либо одним узлом непропорционально большого объема работы. Подсистема балансировки нагрузки может быть установлена на ведущих узлах или подготовлена отдельно от кластера. Выполняя периодические проверки работоспособности каждого узла в кластере, подсистема балансировки нагрузки может определить, произошел ли сбой узла, и если да, то он будет направлять входящий трафик на другие узлы кластера.

Хотя кластер компьютеров изначально не балансирует нагрузку, он позволяет выполнять балансировку нагрузки на своих узлах. Эта конфигурация называется кластером балансировки нагрузки и часто одновременно является кластером высокой доступности.

3. Масштабирование

Существует две классификации масштабирования: вертикальное и горизонтальное. Вертикальное масштабирование (также называемое масштабированием вверх/вниз) включает в себя увеличение или уменьшение ресурсов, выделяемых процессу, таких как объем памяти, количество ядер процессора или доступное хранилище. Горизонтальное масштабирование (горизонтальное масштабирование), с другой стороны, это когда в системе выполняются дополнительные параллельные задания.

При обслуживании кластера важно отслеживать использование ресурсов и масштабирование, чтобы обеспечить надлежащее использование ресурсов кластера. К счастью, сама природа кластера делает его тривиальным для горизонтального масштабирования — администратору просто нужно добавлять или удалять узлы по мере необходимости, помня о минимальном уровне избыточности, чтобы обеспечить высокую доступность кластера.

4. Производительность

Когда дело доходит до распараллеливания, кластеры могут достигать более высоких уровней производительности, чем одна машина. Это связано с

тем, что они не ограничены определенным количеством процессорных ядер или другого оборудования. Кроме того, горизонтальное масштабирование может максимизировать производительность, предотвращая нехватку ресурсов в системе.

Кластеры высокопроизводительных вычислений (HPC) используют распараллеливаемость компьютерных кластеров для достижения максимально возможного уровня производительности. Суперкомпьютер является распространенным примером кластера HPC.

Проблемы кластеризации

Наиболее очевидной проблемой кластеризации является повышенная сложность установки и обслуживания. Операционная система, приложение и его зависимости должны быть установлены и обновлены на каждом узле. Это становится еще более сложным, если узлы в кластере не однородны. Использование ресурсов для каждого узла также должно тщательно контролироваться, а журналы должны быть агрегированы, чтобы убедиться, что программное обеспечение работает правильно. Кроме того, хранилищем становится сложнее управлять; общее запоминающее устройство должно предотвращать перезапись узлов друг другом, а распределенные хранилища данных должны синхронизироваться.

Контейнеризация приложения и размещение кластера в облаке, как я объясню далее, могут помочь облегчить некоторые из этих проблем.

Кластеры в облаке

До появления общедоступного облака компьютерные кластеры состояли из набора физических машин, взаимодействующих через локальную сеть. Создание компьютерного кластера потребовало продуманного планирования, чтобы обеспечить его соответствие нынешним и будущим требованиям, поскольку масштабирование физического кластера может занять недели или даже месяцы. Кроме того, локальные или самоуправляемые кластеры не были устойчивыми в случае региональных бедствий, поэтому для обеспечения избыточности должны были быть приняты другие меры безопасности. Например, использование второго поставщика энергии и размещение узлов в двух физических местах предотвратит отключение электроэнергии в зависимости от поставщика или региона.

Итак, что такое кластер в облачных вычислениях? Проще говоря, это группа узлов, размещенных на виртуальных машинах и подключенных в виртуальном частном облаке. Использование облака позволяет полностью обойти большую часть накладных расходов, связанных с настройкой кластера. Виртуальные машины могут быть подготовлены по требованию, что позволяет кластерам масштабироваться за считанные минуты. Инфраструктура может

быть быстро обновлена, обеспечивая гибкость, необходимую кластеру для адаптации к изменяющимся потребностям. И, наконец, развертывание узлов в нескольких зонах доступности и регионах может повысить задержку пользователей и устойчивость кластера.

Проще говоря, кластеризация в облаке может значительно сократить время и усилия, необходимые для запуска и работы, а также предоставить длинный список услуг для повышения доступности, безопасности и удобства обслуживания кластера.

Стратегия управления ресурсами

Управление ресурсами — это основная функция, необходимая для любой облачной системы, и неэффективное управление ресурсами оказывает прямое негативное влияние на производительность и стоимость, а также может косвенно влиять на функциональность системы, становясь слишком дорогим или неэффективным из-за низкой производительности.

Стратегии управления облачными ресурсами, связанные с тремя моделями предоставления облачных услуг — Iaas, PaaS, SaaS — отличаются друг от друга. В некоторых случаях, когда поставщики облачных услуг могут предсказать всплеск, они могут заранее выделить ресурсы (например, в случае сезонных веб-сервисов).

Однако для незапланированного всплеска ситуация может осложниться. Вы можете использовать Auto Scaling для незапланированных пиковых нагрузок, но для этого вам нужен пул ресурсов, который вы можете высвобождать или выделять по запросу, и система мониторинга, которая позволяет вам в режиме реального времени принимать решения о перераспределении ресурсов. Имейте в виду, что Auto Scaling поддерживается сервисами PaaS, но для IaaS это сложнее из-за отсутствия стандартов.

С технической точки зрения, облако — это часть ресурсов кластера, способная увеличиваться и уменьшаться в соответствии с изменениями нагрузки. Также облачные ресурсы контролируются на трех независимых уровнях:

- **Уровень** кластера. Кластерный уровень управления питанием представлен диспетчером ресурсов кластера, программным комплексом, управляющим ресурсами и задачами в кластере с целью поддержания его эффективности. По сути, CRM отвечает за создание и удаление облаков.
- **Уровень** узла. Управление питанием на уровне узла осуществляется операционной системой (ОС), поэтому ОС контролирует высокоуровневое состояние оборудования. Например, для экономии энергии ОС может переводить процессор (ЦП) в спящий режим или останавливать диски.

- **Аппаратный уровень**. Современные процессоры состоят из множества модулей, которые не всегда могут быть задействованы в работе. Поэтому неиспользуемые модули можно отключить. Это делается специальной схемой, отвечающей за внутреннее управление питанием ЦП. Итак, все управление осуществляется на аппаратном уровне без участия какой-либо ОС.

Управление облаком

Методы распределения в компьютерных облаках должны основываться на дисциплинированном подходе, а не на специальных методах. Итак, вот четыре основных механизма реализации политик управления ресурсами в облачных вычислениях:

- **Теория** управления. Теория управления использует обратную связь, чтобы гарантировать стабильность системы и прогнозировать переходное поведение, но она может предсказывать только локальное поведение.
- **Машинное обучение**. Основным преимуществом методов машинного обучения является то, что им не нужна модель производительности системы. Вы можете применить эту технику для координации нескольких менеджеров вегетативной системы.
- **На основе** полезности. Подходы, основанные на полезности, требуют модели производительности и механизма для сопоставления производительности на уровне пользователя с затратами.
- **Ориентация на рынок**. Такие механизмы не требуют системной модели, такой как объединение аукционов для пакетов ресурсов.

Инфраструктура облачных вычислений представляет собой сложную систему с большим количеством общих ресурсов. Они подвержены непредсказуемым запросам и могут быть затронуты внешними событиями, не зависящими от вас. Управление облачными ресурсами требует сложных политик и решений для многоцелевой оптимизации. Вот почему заблаговременное планирование того, как вы собираетесь управлять этими ресурсами, поможет обеспечить плавный переход к работе с облаком.



МОДУЛЬ 11. Работа с Terraform: как управлять облачной инфраструктурой

1. Облачные провайдеры IaaS, отечественные и зарубежные
2. Синтаксис и принцип работы Terraform
3. Введение в Golang
4. Написание собственных провайдеров для Terraform
5. Развертывание инфраструктуры на платформе Базис.DigitalEnergy с применением Terraform

1. Облачные провайдеры IaaS, отечественные и зарубежные

В последнее время появилось множество ресурсов, меняющих привычный уклад IT-отделов в сфере коммерции, финансов и пр. Одними, из таких являются **IaaS-провайдеры**, предоставляющие для бизнеса IT-инфраструктуру как услугу.

Что это такое и как это использовать

IaaS-инфраструктура – это виртуальные серверы с масштабируемыми ресурсами (ядра процессоров, жесткие диски, оперативная память), объединенные в сеть, на которых устанавливаются любые необходимые программы и приложения. Такие серверы называются облачными.

Используя в работе **IaaS-инфраструктуру**, можно при помощи готовых инструментов, создавать и настраивать все необходимое, в любом количестве, в любой момент времени, для установки программного продукта, его разработки и запуска.

Различают ряд подкатегорий **IaaS услуг**, а именно *публичное*, *частное* или *гибридное* облако. О различии между этими терминами мы поговорим немного ниже.

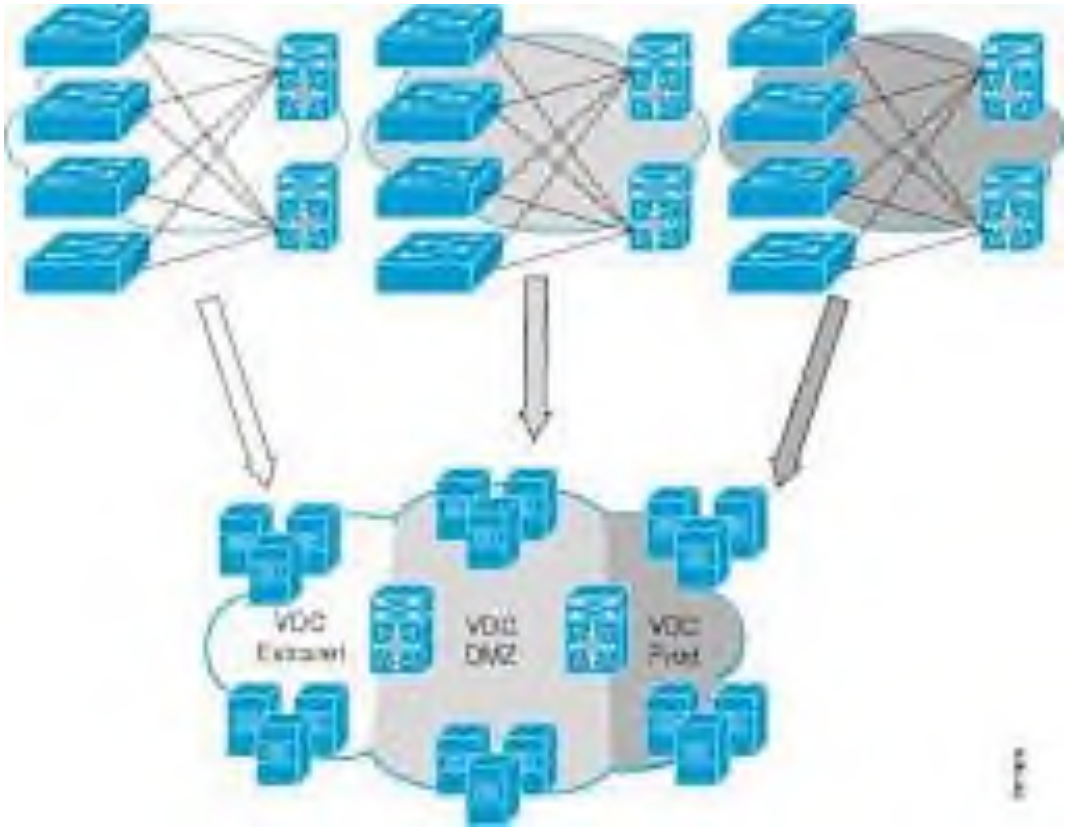
IaaS-провайдер обеспечивает:

- Техническое обслуживание оборудования и его обновление
- Обеспечение виртуализации
- Администрирование сети и серверов на физическом уровне

Услуги IaaS

VDC – Виртуальный дата-центр

При решении арендовать IaaS облако, можно воспользоваться услугами виртуального дата центра (VDC).

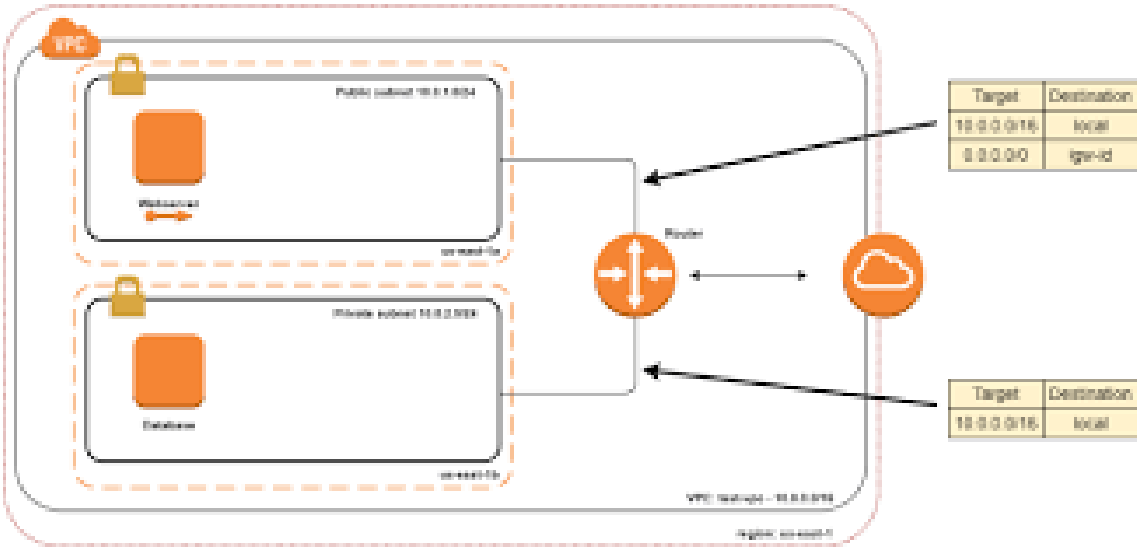


VDC – виртаульный Data-центр – это специально разработанная объединенная система облачной инфраструктуры для бизнеса. Основой являются: ядра процессора, оперативная память, дисковое пространство. Это виртуальная реализацией реального дата-центра, который имеет в работе серверы и сетевые компоненты, находящиеся в определенном виртуальном пространстве. Данные виртуальные пространства находятся на одном или нескольких реальных дата–центрах.

Преимущество в применении облачной обработки данных состоит в том, что она дает доступ к нужным ресурсам в виде собственного дата-центра, при этом организациям не приходиться тратить огромные суммы на создание реального ЦОДа. Каждая организация выбирает именно те ресурсы, которые необходимы ей для работы и платит только за них.

Все действия по регулировке операционных систем выполняются самостоятельно. Сюда входит: установка программ, конфигурирование брандмауэра и т. д.

VDC дает возможность выбрать раздел облака, чтобы настроить в этих рамках ресурсы для создания виртуальной сети. В этом случае у пользователя под контролем находится вся выбранная среда виртуальной сети. При этом возможно проводить выбор собственных диапазонов IP-адресов, настройку таблиц маршрутизации.



Сеть VDC настраивается исходя из пожеланий клиента. Данная система обеспечивает многоступенчатую систему безопасности. Она состоит из ряда групп безопасности и списков по контролю доступа в сеть. Это дает возможность проводить контроль по доступу к инстантам в каждой подсети.

На любом этапе работы в вашем распоряжении служба поддержки, которая даст необходимую консультацию по всем возникающим вопросам. Часть сервисов предлагает даже произвести ряд работ, касающихся настройки сервисов за клиента. Но это будет стоить дополнительных затрат.

Зарубежные облачные провайдеры

93% компаний используют несколько облачных сервисов — это обусловлено стремлением к конфиденциальности, зависимостью от устойчивости провайдера и необходимостью постоянного “бэкапа”. Рассказываем, какие компании популярны в мире.

США

Глобальный, но прежде всего американский лидер — Amazon Web Services (AWS).

Спектр облачных продуктов у Amazon очень широкий. Они предлагают клиентам вычислительные сервисы, хранилища, базы данных, инструменты аналитики, сетевые технологии, мобильные сервисы, инструменты для разработчиков, инструменты управления, технологии Интернета вещей, средства обеспечения безопасности и корпоративные приложения. По итогам III квартала 2021 г. под контролем AWS находится 32% мирового облачного рынка. В топ-3 компаний также входят Microsoft Azure и Google.

Однако подобное “господство” имеет минусы – если происходит сбой, работа встает у сотен популярных сайтов, сервисов и приложений, а сама компания несет огромные финансовые потери. Сбои у AWS последнее время

происходят часто — три случая только за декабрь, но компания по-прежнему остается одной из самых надежных в мире.

Европа

Помимо компаний из “большой тройки” (да, в Европе преобладают все те же AWS, Microsoft и Google), местные пользуются услугами компаний IBM, Salesforce и, к примеру, OVHcloud.

Последняя из перечисленных компаний родом из Франции. Она считается одним из крупнейших облачных провайдеров в Европе, а также известным поставщиком хостинг-услуг. У OVHcloud есть решения для публичного и частного облака, VPS, выделенного сервера, а также пул корпоративных продуктов, связанных с BigData, искусственным интеллектом и машинным обучением.

30 центров обработки данных компании охватывают 183 страны на пяти континентах. Она также проектирует и строит собственную волоконно-оптическую сеть.

Отечественные облачные провайдеры

В настоящее время активно развивается отечественный рынок облачных решений. Существует большое количество российских облачных провайдеров, которые ничуть не уступают, а во многом даже превосходят возможности зарубежных корпораций. Представляем некоторые облачные платформы в России.

RTCloud

RTCloud - облачная платформа компании «Ростелеком», реализуемая «РТКомм». Она обеспечивает доступ к мощнейшим вычислительным ресурсам, сервисам и программному обеспечению сети современных центров обработки данных на всей территории России. Концепция облачных вычислений заключается в предоставлении конечным пользователям удаленного доступа (через интернет) к динамично масштабируемым вычислительным ресурсам (включая операционные системы и инфраструктуру), поддержку и обновление которых осуществляет провайдер услуг.

Timeweb Cloud

Timeweb Cloud – современная облачная платформа, входящая в экосистему Timeweb.

Платформа позволяет создавать отказоустойчивые серверы и базы данных в облаке, использовать S3-совместимое объектное хранилище, балансировщики нагрузки и другие облачные решения.

Для бесперебойной работы облака компания использует технологические решения от ведущих мировых вендоров Intel, AMD и Juniper Networks, а также резервирует архитектуру по схеме N+1. Это обеспечивает круглосуточную доступность проектов любого масштаба и сложности.

Размещает оборудование в дата-центрах уровня TIER III, отвечающих всем требованиям безопасности, соответствует ФЗ-152 и использует самые высокоскоростные линии связи.

Из других плюсов платформы:

Бесплатный перенос проектов в облако.

Бесплатное администрирование.

Бесплатная техническая и информационная поддержка в мессенджерах, по телефону, в чате и тикетах.

VK Cloud Solutions

ИТ-платформа для бизнеса от VK. Среди решений VK Cloud Solutions есть виртуальные серверы и сети, объектное хранилище S3, аренда VPS/VDS, управляемые базы данных, кластеры Kubernetes и другие возможности. Платформа предлагает любые варианты размещения (как Public, так и Private) и полностью соответствует требованиям 152-ФЗ.

Особенности:

Тройная репликация данных.

Защита соединений с облаком.

Послеаварийное восстановление.

Yandex.Cloud

Облачная система от Яндекса. Yandex.Cloud предоставляет вычислительные мощности для размещения, тестирования и прототипирования проектов. В сервисах заявлен широкий ряд возможностей, например: запуск моделей машинного обучения, контейнерная разработка и масштабируемое хранилище данных.

Особенности:

Безопасная разработка и многоуровневая защита.

Партнерская программа страхования рисков.

Техническая поддержка – есть бесплатный пул услуг, платные по индивидуальному запросу.

SberCloud

Облачная платформа, которая входит в экосистему Сбера. Предлагает более 50 облачных сервисов (работа с БД, хранение информации, аналитика данных, обеспечение безопасности и т. д.). Также SberCloud – это разработчик суперкомпьютеров Christofari и Christofari Neo.

Особенности:

Платформа для разработки machine learning полного цикла.

Лицензии ФСТЭК и ФСБ на оказание услуг в области защиты информации и информационных систем.

Виртуальный ЦОД на базе VMware.

#Cloud MTS

Облачная система от МТС, которая включает в себя решения в формате частных, публичных и гибридных облаков. #CloudMTS предлагает клиентам ресурсы для создания ИТ-инфраструктуры с помощью 25 сервисов.

Особенности:

Гибкая модель тарификации и прозрачная стоимость услуг.

Круглосуточная поддержка, выделенная команда экспертов.

Кастомизация решений.

2. Синтаксис и принцип работы Terraform

Terraform — это инструмент «инфраструктура как код» (IaC), который позволяет безопасно и эффективно создавать, изменять и версионировать инфраструктуру. Сюда входят как компоненты низкого уровня, такие как вычислительные экземпляры, хранилище и сеть, так и компоненты высокого уровня, такие как записи DNS и функции SaaS.

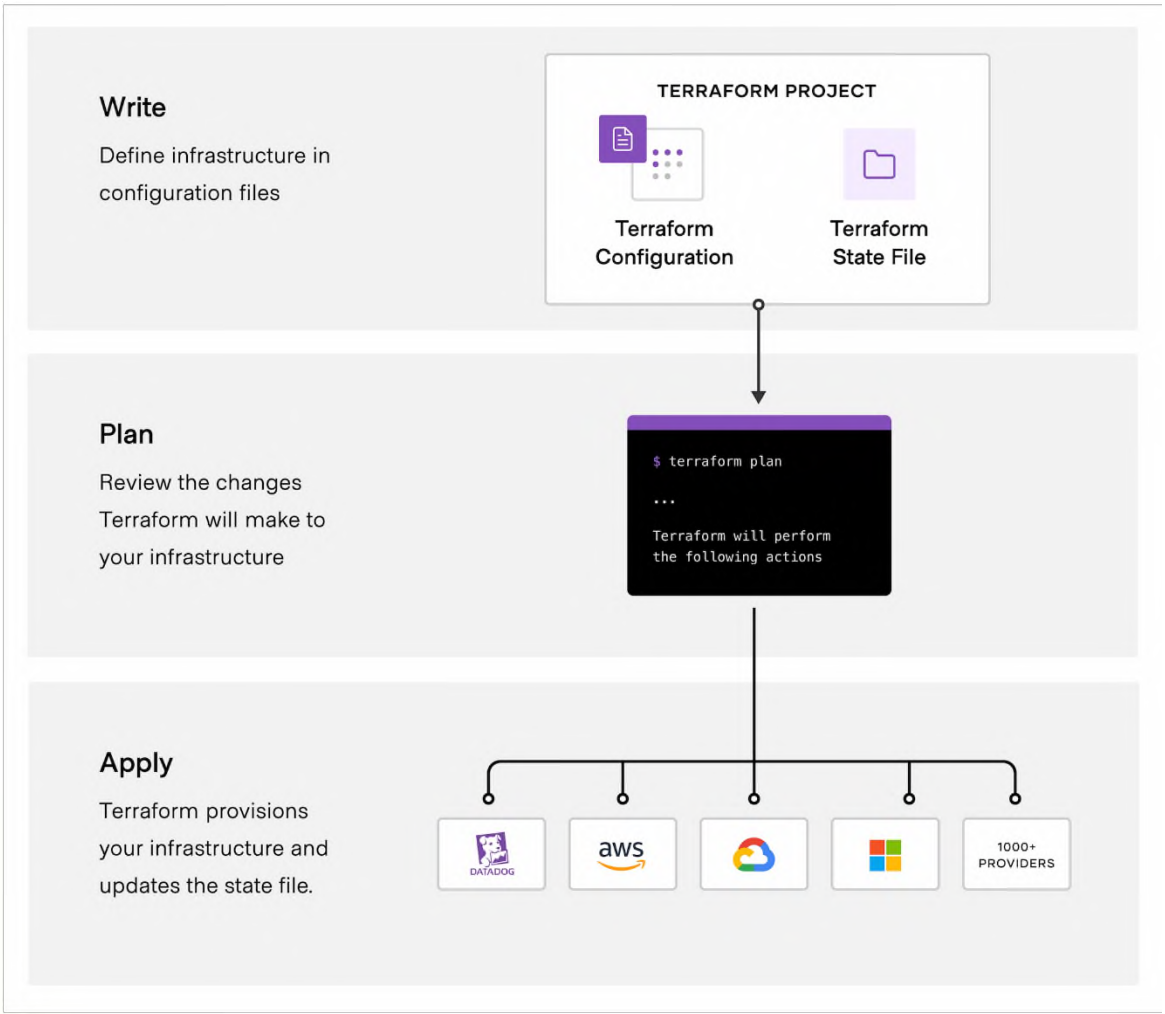
Как работает Терраформ?

Terraform создает и управляет ресурсами на облачных платформах и других сервисах через свои интерфейсы прикладного программирования (API). Провайдеры позволяют Terraform работать практически с любой платформой или сервисом с помощью доступного API.



Основной рабочий процесс Terraform состоит из трех этапов:

- **Write:** вы определяете ресурсы, которые могут быть у нескольких облачных провайдеров и служб. Например, вы можете создать конфигурацию для развертывания приложения на виртуальных машинах в сети виртуального частного облака (VPC) с группами безопасности и балансировщиком нагрузки.
- **Plan:** Terraform создает план выполнения, описывающий инфраструктуру, которую он создаст, обновит или уничтожит на основе существующей инфраструктуры и вашей конфигурации.
- **Apply:** после утверждения Terraform выполняет предложенные операции в правильном порядке, соблюдая любые зависимости ресурсов. Например, если вы обновите свойства VPC и измените количество виртуальных машин в этом VPC, Terraform воссоздаст VPC перед масштабированием виртуальных машин.



Преимущества и составляющие

В работе с инструментом мы отметили несколько преимуществ:

- **Скорость развертывания новых тенантов** (пользовательских виртуальных сред). Обычно, чем больше новых клиентов, тем больше «кликов» требуется сделать сотрудникам технической поддержки для публикации новых ресурсов. С Terraform пользователи могут изменять параметры виртуальных машин (например, автоматически выключать ОС и увеличивать раздел виртуального диска) без участия технической поддержки и выключения самой машины.
- **Моментальная проверка плана активации** нового теннанта. С помощью описания кода инфраструктуры мы можем сразу же проверить, что и в каком порядке будет добавлено, а также в каком конечном состоянии будет та или иная виртуальная машина или виртуальная сеть с подключениями к виртуальным машинам.
- **Возможность описывать большинство популярных облачных платформ.** Вы можете использовать инструмент от Amazon и Google Cloud, до

частных платформ на базе VMware vCloud Director, предлагающих услуги в рамках IaaS, SaaS и PaaS решений.

- **Управлять несколькими облачными провайдерами** и распространять инфраструктуру между ними для повышения отказоустойчивости, используя одну конфигурацию для создания, диагностики и управления облачными ресурсами.
- **Удобное использование для создания демо-стендов** под тестирование и отладку программного обеспечения. Вы можете создавать и передавать стенды для отдела тестирования, параллельно проверять ПО в разных средах, а также моментально изменять и удалять ресурсы, создав всего лишь один план построения ресурсов

«Террариум» Terraform

Кратко рассказали про преимущества инструмента, теперь разберем его на составляющие

Providers (провайдеры).

В Terraform практически любой тип инфраструктуры можно представить в качестве ресурса. Связь между ресурсами и платформой API обеспечивается providers модулями, которые позволяют создавать ресурсы в рамках определённой платформы, например, Azure или VMware vCloud Director.

В рамках проекта вы можете взаимодействовать с разными провайдерами на разных платформах.

Resources (описание ресурсов).

Описание ресурсов позволяет управлять компонентами платформы, например виртуальными машинами или сетями.

Вы можете самостоятельно создать описание ресурсов для провайдера VMware vCloud Director и использовать это описание для создания ресурсов у любого хостинг-провайдера, который использует vCloud Director. Вам потребуется лишь заменить параметры аутентификации и параметры сетевого подключения к необходимому хостинг-провайдеру

Provisioners.

Эта составляющая дает возможность выполнять операции по первоначальной установке и обслуживанию операционной системы после создания виртуальных машин. После того, как вы создали ресурс виртуальной машины, с помощью provisioners вы можете настроить и подключиться по SSH, выполнить обновление операционной системы, а также загрузить и выполнить скрипт.

Переменные Input и Output.

Input переменные — входные переменные для любых типов блоков.

Output переменные позволяют сохранить значения после создания ресурсов и могут быть использованы, как входные переменные в других модулях, например в блоке Provisioners.

States (состояния).

States-файлы хранят информацию о конфигурации ресурсов платформы провайдера. При первом создании платформы никаких сведений о ресурсах нет и перед любой операцией Terraform обновляет состояние с реальной инфраструктурой уже описанных ресурсов.

Основная цель состояний, это сохранение связки объектов уже созданных ресурсов для сравнения конфигурации добавляемых ресурсов и объектов, чтобы избежать повторного создания и изменений платформы.

Информация о состоянии по умолчанию хранится в локальном файле terraform.tfstate, но при необходимости есть возможность использовать удаленное хранение для работы в команде.

Также вы можете импортировать текущие ресурсы платформы в состояние, чтобы далее взаимодействовать с другими ресурсами, которые в свою очередь были созданы без помощи Terraform.

Конфигурирование

Terraform использует текстовые файлы для описания инфраструктуры и установки переменных. Эти текстовые файлы называются *конфигурациями* Terraform и имеют `.tf` . В этом разделе рассказывается о формате этих файлов, а также о том, как они загружаются.

Формат файлов конфигурации может быть двух форматов: формат Terraform и JSON. Формат Terraform более удобен для чтения, поддерживает комментарии и является обычно рекомендуемым форматом для большинства файлов Terraform. Формат JSON предназначен для машин, которые могут создавать, изменять и обновлять, но при желании также может выполняться операторами Terraform. Формат Terraform заканчивается на `.tf` , а формат JSON - на `.tf.json` .

Синтаксис конфигурации

Синтаксис конфигураций Terraform называется *HashiCorp Configuration Language (HCL)* . Он предназначен для обеспечения баланса между удобочитаемостью и редактированием, а также удобством для машин. Для удобства использования Terraform может также считывать конфигурации JSON.

Синтаксис терраформ

Вот пример синтаксиса HCL компании Terraform:

```
# AMI
```

```
variable "ami" {
  description = "the AMI to use"
}

/* A multi
   line comment. */
resource "aws_instance" "web" {
  ami               = "${var.ami}"
  count             = 2
  source_dest_check = false

  connection {
    user = "root"
  }
}
```

Базовая точка отсчета:

- Однострочные комментарии начинаются с #
- Многострочные комментарии заключаются в символы /* и */
- Значения присваиваются с помощью синтаксиса key = value (пробелы не имеют значения). Значение может быть любым примитивом (строка, число, логическое значение), списком или картой.
- Строки в двойных кавычках.
- Строки могут интерполировать другие значения, используя синтаксис, заключенный в ${} , например ${var.foo} .
- Многострочные строки могут использовать синтаксис «здесь, документ» в стиле оболочки, при этом строка начинается с маркера типа <<EOF , а затем строка заканчивается на EOF в отдельной строке. Строки строки и конечный маркер *не* должны иметь отступа.
- Предполагается, что числа имеют основание 10. Если вы поставите перед числом префикс 0x , оно будет рассматриваться как шестнадцатеричное число.
- Логические значения: true , false .
- Списки примитивных типов можно составлять с помощью квадратных скобок ([]). Пример: ["foo", "bar", "baz"] .
- Карты могут быть сделаны с фигурными скобками ({}) и двоеточие (:): { "foo": "bar", "bar": "baz" } . Кавычки могут быть опущены на ключах, если ключ не начинается с числа, и в этом случае кавычки требуются. Для однострочных карт необходимы запятые между парами ключ / значение. В

многострочных картах достаточно новой строки между парами ключ / значение.

В дополнение к основам синтаксис поддерживает иерархии разделов, такие как "ресурс" и "переменная" в примере выше. Эти секции похожи на карты, но визуально выглядят лучше. Например, они почти эквивалентны:

```
variable "ami" {
 description = "the AMI to use"
}
приравнивается:
variable = [{
 "ami": {
  "description": "the AMI to use",
 }
}]
```

Заметили, как визуально верхняя строфа выглядит намного лучше? Повторяя несколько разделов variable , он создает список variable . По возможности используйте разделы, так как они визуально понятнее и читабельнее.

Синтаксис JSON

Terraform также поддерживает чтение конфигурационных файлов в формате JSON.Приведенный выше пример конвертирован в JSON:

```
{
 "variable": {
  "ami": {
   "description": "the AMI to use"
  }
 },

 "resource": {
  "aws_instance": {
   "web": {
    "ami": "${var.ami}",
    "count": 2,
    "source_dest_check": false,

    "connection": {
     "user": "root"
```

```
      }
     }
    }
   }
  }
```

Преобразование должно быть довольно простым и документально подтвержденным.

Недостатками СПСН являются меньшая читабельность и отсутствие комментариев. В противном случае эти два аспекта полностью взаимозаменяемы.

Конфигурация источника данных

Источники данных позволяют извлекать или вычислять данные для использования где-нибудь в конфигурации Terraform. Использование источников данных позволяет конфигурации Terraform основываться на информации, определенной вне Terraform или определенной другой отдельной конфигурацией Terraform.

В Terraform поставщики несут ответственность за определение и реализацию источников данных. В то время как ресурс заставляет Terraform создавать и управлять новым компонентом инфраструктуры, источники данных представляют представления только для чтения в уже существующие данные или вычисляют новые значения на лету внутри самого Terraform.

Например,источник данных может получить информацию об артефактах от Terraform Enterprise, информацию о конфигурации от Consul, или просмотреть уже существующий ресурс AWS путем фильтрации по его атрибутам и тегам.

Каждый источник данных в Terraform сопоставляется с поставщиком на основе совпадения самого длинного префикса. Например, `aws_ami` данных aws_ami будет сопоставляться с поставщиком `aws` (если он существует).

Пример

Конфигурация источника данных выглядит следующим образом:

```
# Найдите последний доступный AMI с тегом Component = web
data "aws_ami" "web" {
 filter {
   name   = "state"
   values = ["available"]
 }
```

```
  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

Описание

Блок data создает экземпляр данных с заданным TYPE (первый параметр) и NAME (второй параметр). Сочетание типа и имени должно быть уникальным.

Внутри блока ({ }) находится конфигурация для экземпляра данных. Конфигурация зависит от типа и документируется для каждого источника данных.

Каждый экземпляр данных будет экспортировать один или несколько атрибутов, которые можно интерполировать в другие ресурсы с помощью переменных формы data.TYPE.NAME.ATTR . Например:

```
resource "aws_instance" "web" {
  ami           = "${data.aws_ami.web.id}"
  instance_type = "t1.micro"
}
```

Meta-parameters

Поскольку источники данных по сути представляют собой подмножество ресурсов только для чтения, они также поддерживают те же мета-параметры ресурсов, за исключением блока конфигурации lifecycle .

Многочисленные обстоятельства поставщика

Подобно ресурсам , мета-параметр provider может использоваться, когда конфигурация имеет несколько экземпляров одного и того же поставщика с псевдонимами:

```
data "aws_ami" "web" {
  provider = "aws.west"

  # ...
}
```

Жизненный цикл источника данных

Если аргументы экземпляра данных не содержат ссылок на вычисленные значения, такие как атрибуты ресурсов, которые еще не

созданы, то экземпляр данных будет прочитан и его состояние обновлено на этапе "обновления" Terraform, который по умолчанию запускается перед созданием плана. Это гарантирует, что полученные данные будут доступны для использования во время планирования, а разница будет показывать реальные полученные значения.

Аргументы экземпляра данных могут относиться к вычисленным значениям, в этом случае атрибуты самого экземпляра не могут быть разрешены до тех пор, пока не будут определены все его аргументы. В этом случае обновление экземпляра данных будет отложено до фазы "применения", и все интерполяции атрибутов экземпляра данных будут отображаться как "вычисленные" в плане, так как значения еще не известны.

Переменные среды

TF_LOG

Если установлено любое значение, то позволяет отображать подробные журналы на stderr, что полезно при отладке. Например:

```
export TF_LOG=TRACE
```

Чтобы отключить, либо снимите его, либо установите на пустое место. При снятии этого параметра по умолчанию будет использоваться stderr. Например:

```
export TF_LOG=
```

Дополнительные сведения об отладке Terraform см. В разделе « Отладка» .

TF_LOG_PATH

Это указывает, где журнал должен сохранять свои выходные данные. Обратите внимание, что даже если `TF_LOG_PATH` установлен, `TF_LOG` должен быть установлен для того, чтобы любой лесозаготовок должен быть включен. Например, чтобы всегда записывать журнал в каталог, из которого вы сейчас запускаете terraform:

```
export TF_LOG_PATH=./terraform.log
```

TF_INPUT

Если установлено значение «false» или «0», заставляет команды terraform вести себя так, как если бы был указан флаг -input=false . Это используется, когда вы хотите отключить запросы для переменных, для которых не указаны их значения. Например:

```
export TF_INPUT=0
```

TF_MODULE_DEPTH

При задании значения заставляет команды терраформ вести себя так, как если бы был указан флаг -module-depth=VALUE . Например, установив для него значение 0, вы активируете такие команды, как план и график, для отображения более сжатой информации.

```
export TF_MODULE_DEPTH=0
```

TF_VAR_name

Переменные среды можно использовать для установки переменных. Переменные среды должны быть в формате TF_VAR_name , и это значение будет проверяться в последнюю очередь. Например:

```
export TF_VAR_region=us-west-1
export TF_VAR_ami=ami-049d8641
export TF_VAR_alist='[1,2,3]'
export TF_VAR_amap='{foo = "bar", baz = "qux" }'
```

TF_CLI_ARGS и TF_CLI_ARGS_name

Значение TF_CLI_ARGS будет указывать дополнительные аргументы для командной строки. Это позволяет упростить автоматизацию в средах CI, а также изменить поведение Terraform по умолчанию в вашей собственной системе.

Эти аргументы вставляются непосредственно *после* подкоманды (например, plan) и *перед* любыми флагами, указанными непосредственно в командной строке. Такое поведение гарантирует, что флаги в командной строке имеют приоритет над переменными среды.

Например, следующая команда: TF_CLI_ARGS="-input=false" terraform apply -force эквивалентна вводу вручную: terraform apply -input=false -force .

Флаг TF_CLI_ARGS влияет на все команды Terraform. Если вы укажете именованную команду в форме TF_CLI_ARGS_name , то это повлияет только на эту команду. В качестве примера, чтобы указать, что только планы никогда не TF_CLI_ARGS_plan="-refresh=false" , вы можете установить TF_CLI_ARGS_plan = "- refresh = false" .

Значение флага анализируется так,как если бы вы напечатали его непосредственно в оболочке.Двойные и одинарные кавычки позволяют захватывать строки,а аргументы будут разделены пробелами в противном случае.

TF_DATA_DIR

TF_DATA_DIR изменяет место, где Terraform хранит свои данные для каждого рабочего каталога, например текущую конфигурацию удаленного сервера.

По умолчанию эти данные записываются в подкаталог .terraform текущего каталога, но вместо TF_DATA_DIR будет использоваться путь, указанный в TF_DATA_DIR , если он не пустой.

В большинстве случаев нет необходимости устанавливать эту переменную, но это может быть полезно, если, например, рабочая директория не доступна для записи.

Каталог данных используется для хранения данных, которые должны сохраняться от одной команды к другой, поэтому важно, чтобы эта переменная была установлена последовательно во всех командах рабочего процесса Terraform (начиная с terraform init), иначе Terraform может не найти поставщиков, модули и другие артефакты.

TF_SKIP_REMOTE_TESTS

Это можно настроить перед запуском модульных тестов, чтобы отказаться от любых тестов, требующих удаленного подключения к сети. Модульные тесты пытаются автоматически определить, когда соединение недоступно, и пропустить соответствующие тесты, но установив эту переменную, вы можете заставить пропустить эти тесты.

```
export TF_SKIP_REMOTE_TESTS=1
make test
```

Синтаксис интерполяции

Встроенный в строки в Terraform, независимо от того, используете ли вы синтаксис Terraform или синтаксис JSON, вы можете интерполировать другие значения. Эти интерполяции заключены в ${} , например ${var.foo} .

Синтаксис интерполяции является мощным и позволяет ссылаться на переменные, атрибуты ресурсов, функции вызова и т. д.

Вы можете выполнять математические операции при интерполяции, позволяя писать такие выражения, как ${count.index + 1} . И вы также можете использовать условные выражения для определения значения на основе некоторой логики.

Вы можете избежать интерполяции с помощью двойных знаков доллара: $${foo} будет отображаться как буквальный ${foo} .

Доступные переменные

Существует множество доступных ссылок на переменные, которые вы можете использовать.

Строковые переменные

Используйте var. префикс, за которым следует имя переменной. Например, ${var.foo} интерполирует значение переменной foo .

Ассоциативный массив или map

Синтаксис: `var.MAP["KEY"]` . Например, `${var.amis["us-east-1"]}` получит значение ключа `us-east-1` в переменной карты `amis` .

Списки

Синтаксис: `"${var.LIST}"` . Например, `"${var.subnets}"` получит значение списка `subnets` в виде списка. Вы также можете возвращать элементы списка по индексу: `${var.subnets[idx]}` .

Атрибуты вашего собственного ресурса

Синтаксис: `self.ATTRIBUTE` . Например, `${self.private_ip}` интерполирует частный IP-адрес этого ресурса.

Примечание. Синтаксис `self.ATTRIBUTE` разрешен и действителен только в средствах обеспечения.

Атрибуты прочих ресурсов

Синтаксис: `TYPE.NAME.ATTRIBUTE` . Например, `${aws_instance.web.id}` интерполирует атрибут ID из ресурса `aws_instance` с именем `web` . Если у ресурса установлен атрибут `count` , вы можете получить доступ к отдельным атрибутам с индексом с отсчетом от нуля, например `${aws_instance.web.0.id}` . Вы также можете использовать синтаксис splat, чтобы получить список всех атрибутов: `${aws_instance.web.*.id}` .

Атрибуты источника данных

Синтаксис: `data.TYPE.NAME.ATTRIBUTE` . Например. `${data.aws_ami.ubuntu.id}` интерполирует атрибут `id` из `aws_ami` данных aws_ami с именем `ubuntu` . Если для источника данных установлен атрибут `count` , вы можете получить доступ к отдельным атрибутам с помощью индекса с отсчетом от нуля, например `${data.aws_subnet.example.0.cidr_block}` . Вы также можете использовать синтаксис splat, чтобы получить список всех атрибутов: `${data.aws_subnet.example.*.cidr_block}` .

Выходы из модуля

Синтаксис: `MODULE.NAME.OUTPUT` . Например, `${module.foo.bar}` будет интерполировать вывод `bar` из модуля `foo` .

Информация для подсчёта

Синтаксис: `count.FIELD` . Например, `${count.index}` интерполирует текущий индекс в ресурсе с несколькими счетчиками.

Информация о путях

Синтаксис: `path.TYPE` . ТИП может быть `cwd` , `module` или `root` . `cwd` интерполирует текущий рабочий каталог. `module` интерполирует путь к текущему модулю. `root` интерполирует путь к корневому модулю. В общем, вам, вероятно, понадобится переменная `path.module` .

Метаинформация о терраформах

Синтаксис: `terraform.FIELD` . Этот тип переменной содержит метаданные о текущем выполнении Terraform. FIELD в настоящее время может быть `env` только для ссылки на текущую среду активного состояния .

Условия

Интерполяции могут содержать условия для ответвления на конечное значение.

```
resource "aws_instance" "web" {
  subnet = "${var.env == "production"? var.prod_subnet : var.dev_subnet}"
}
```

Условный синтаксис — это известная троичная операция:

```
CONDITION? TRUEVAL: FALSEVAL
```

Условием может быть любой корректный синтаксис интерполяции, например доступ к переменным, вызов функции или даже другое условие. Значение true и false также может быть любым допустимым синтаксисом интерполяции. Возвращаемые типы со стороны true и false должны быть одинаковыми.

Поддерживаемые операторы:

- Равенство: == и !=
- Численное сравнение: > , < , >= , <=
- Логическая логика: && , || , одинарный !

Обычным случаем использования условных значений является включение/выключение ресурса путем условной установки счетчика:

```
resource "aws_instance" "vpn" {
  count = "${var.something ? 1 : 0}"
}
```

В приведенном выше примере ресурс "vpn" будет включен только в том случае,если "var.something" вычислит true. В противном случае VPN-ресурс не будет создан вообще.

Встроенные функции

Terraform поставляется со встроенными функциями. Функции вызываются с `name(arg, arg2, ...)` синтаксиса (arg, arg2, ...) . Например, чтобы прочитать файл: `${file("path.txt")}` .

ПРИМЕЧАНИЕ. Правильное экранирование требуется для значений полей JSON, содержащих кавычки ("), таких как значения `environment` . Если JSON устанавливается напрямую, они должны быть экранированы как `\"` в JSON, например, `"value": "I \"love\" escaped quotes"` . При использовании

значения переменной Terraform они должны быть экранированы как \\\" в переменной, например, value = "I \\\"love\\\" escaped quotes" в переменной и "value": "${var.myvariable}" в формате JSON.

Поддерживаемые встроенные функции

Поддерживаются встроенные функции:

- abs(float) - возвращает абсолютное значение заданного числа с плавающей запятой. Пример: abs(1) возвращает 1 , а abs(-1) также вернет 1 , тогда как abs(-3.14) вернет 3.14 . См. Такжефункцию signum .
- basename(path) - возвращает последний элемент пути.
- base64decode(string) - учитывая строку в кодировке base64, декодирует ее и возвращает исходную строку.
- base64encode(string) - возвращает представление данной строки в кодировке base64.
- base64gzip(string) - сжимает заданную строку с помощью gzip, а затем кодирует результат в base64. Это можно использовать с определенными аргументами ресурсов, которые позволяют передавать двоичные данные с кодировкой base64, поскольку строки Terraform должны быть действительными UTF-8.
- base64sha256(string) - возвращает представление в кодировке base64 необработанной суммы SHA-256 данной строки. **Это не является эквивалентом** из base64encode(sha256(string)) , так как sha256() возвращает шестнадцатеричное представление.
- base64sha512(string) - возвращает представление в кодировке base64 необработанной суммы SHA-512 данной строки. **Это не является эквивалентом** из base64encode(sha512(string)) , так как sha512() возвращает шестнадцатеричное представление.
- bcrypt(password, cost) - возвращает зашифрованный с помощью Blowfish хеш строки по заданной стоимости. Если не указано иное, будет использована cost поумолчанию10.
- ceil(float) - возвращает наименьшее целое значение, большее или равное аргументу.
- chomp (string) - удаляет завершающие символы новой строки из данной строки.
- chunklist(list, size) - возвращаетэлементы list , разбитые по size . Примеры:
  - ✓ chunklist (aws_subnet.foo.*.id, 1) : выводит [["id1"], ["id2"], ["id3"]]

- ✓ chunklist (var.list_of_strings, 2) : выводит [["id1", "id2"], ["id3", "id4"], ["id5"]]
- cidrhost (iprange, hostnum) - принимает диапазон IP-адресов в нотации CIDR и создает IP-адрес с заданным номером хоста. Если заданный номер хоста отрицательный, счет начинается с конца диапазона. Например, cidrhost("10.0.0.0/8", 2) возвращает 10.0.0.2 , а cidrhost("10.0.0.0/8", -2) возвращает 10.255.255.254 .
- cidrnetmask(iprange) - принимает диапазон IP-адресов в нотации CIDR и возвращает формат маски подсети в формате адреса, который некоторые системы ожидают для интерфейсов IPv4. Например, cidrnetmask("10.0.0.0/8") возвращает 255.0.0.0 . Не применимо к сетям IPv6, поскольку нотация CIDR является единственной допустимой нотацией для IPv6.
- cidrsubnet(iprange, newbits, netnum) - принимает диапазон IP-адресов в нотации CIDR (например, 10.0.0.0/8) и расширяет его префикс, чтобы включить дополнительный номер подсети. Например, cidrsubnet("10.0.0.0/8", 8, 2) возвращает 10.2.0.0/16 ; cidrsubnet("2607:f298:6051:516c::/64", 8, 2) возвращает 2607:f298:6051:516c:200::/72 .
- coalesce(string1, string2, ...) - возвращает первое непустое значение из заданных аргументов. Необходимо указать как минимум два аргумента.
- Coalescelist (list1, list2, ...) - возвращает первый непустой список из заданных аргументов. Необходимо указать как минимум два аргумента.
- Compact (list) - удаляет пустые строковые элементы из списка. Это может быть полезно в некоторых случаях, например, при передаче объединенных списков в качестве переменных модуля или при анализе выходных данных модуля. Пример: compact(module.my_asg.load_balancer_names)
- concat (list1, list2, ...) - объединяет два или более списков в один. Пример: concat (aws_instance.db.*.tags.Name, aws_instance.web.*.tags.Name)
- contains (list, element) - возвращает *true,* если список содержит данный элемент, и возвращает *false в* противном случае. Примеры: contains(var.list_of_strings, "an_element")
- dirname(path) - возвращает все, кроме последнего элемента пути, обычно это каталог пути.

- distinct(list) - удаляет повторяющиеся элементы из списка. Сохраняет первое вхождение каждого элемента и удаляет последующие вхождения. Эта функция действительна только для плоских списков. Пример: distinct(var.usernames)
- element(list, index) - возвращает единственный элемент из списка по заданному индексу. Если индекс больше количества элементов, эта функция будет завершена с использованием стандартного алгоритма модификации. Эта функция работает только с плоскими списками. Примеры:
  - ✓ element(aws_subnet.foo.*.id, count.index)
  - ✓ element(var.list_of_strings, 2)
- file(path) - считывает содержимое файла в строку. Переменные в этом файле *не* интерполируются. Содержимое файла читается как есть. path интерпретируется относительно рабочего каталога. Переменные пути можно использовать для ссылки на пути относительно других базовых местоположений. Например, при использовании file() изнутри модуля вы обычно хотите file("${path.module}/file") путь относительно базы модуля, например: file ("$ {path.module} / file") .
- floor(float) - возвращает наибольшее целое значение, меньшее или равное аргументу.
- flatten(list of lists) - сглаживает списки списков до плоского списка примитивных значений, рекурсивно удаляя все вложенные списки. Примеры:
  - ✓ flatten(data.github_user.user.*.gpg_keys)
- format(format, args, ...) - форматирует строку в соответствии с заданным форматом. Синтаксис формата - стандартныйсинтаксис sprintf . Хорошую документацию по синтаксису можно найти здесь . Пример нулевого префикса счетчика, обычно используемый для именования серверов: format("web-%03d", count.index + 1) .
- formatlist(format, args, ...) - форматирует каждый элемент списка в соответствии с заданным форматом, аналогично format , и возвращает список. Аргументы, не входящие в список, повторяются для каждого элемента списка. Например, чтобы преобразовать список адресов DNS в список URL-адресов, вы можете использовать: formatlist("https://%s:%s/", aws_instance.foo.*.public_dns, var.port) . Если несколько аргументов являются списками и имеют одинаковое количество элементов, то форматирование применяется к элементам

списков параллельно. Пример: formatlist("instance %v has private ip %v", aws_instance.foo.*.id, aws_instance.foo.*.private_ip) . Передача списков разной длины в список форматирования приводит к ошибке.

- indent(numspaces, string) - добавляет указанное количество пробелов перед всеми, кроме первой строки данной многострочной строки. Может быть полезно при вставке многострочной строки в контекст с уже имеющимся отступом. Первая строка не имеет отступа, чтобы строка с отступом могла быть размещена после некоторого вида преамбулы с уже имеющимся отступом. Пример: " \"items\": ${ indent(4, "[\n \"item1\"\n]") },"
- index(list, elem) - Находит индекс заданного элемента в списке. Эта функция работает только с плоскими списками. Пример: index(aws_instance.foo.*.tags.Name, "foo-test")
- join(delim, list) - объединяет список с разделителем для результирующей строки. Эта функция работает только с плоскими списками. Примеры:
  - ✓ join(",", aws_instance.foo.*.id)
  - ✓ join(",", var.ami_list)
- jsonencode(value) - возвращает представление данного значения в кодировке JSON, которое может содержать произвольно вложенные списки и карты. Обратите внимание: если значение является строкой, его значение будет заключено в кавычки.
- keys(map) - возвращает лексически отсортированный список ключей карты.
- Length (list) - возвращает количество элементов в данном списке или карте или количество символов в данной строке.
  - ✓ ${length (split (",", "a,b,c"))} = 3
  - ✓ ${length("a,b,c")} = 5
  - ✓ ${length (map("key", "val"))} = 1
- list(items, ...) - возвращает список, состоящий из аргументов функции. Эта функция обеспечивает способ представления литералов списка в интерполяции.
  - ✓ ${list("a", "b", "c")} возвращает список "a", "b", "c" .
  - ✓ ${list()} возвращает пустой список.
- log(x, base) - возвращает логарифм x .
- lookup(map, key, [default]) - Выполняет динамический поиск в переменной карты. Параметр map должен быть другой переменной, например var.amis . Если key не существует в map , интерполяция не

удастся, если вы не укажете третий аргумент, default , который должен быть строковым значением, возвращаемым, еслив map ненайден key . Эта функция работает только на плоских картах и возвращает ошибку для карт, содержащих вложенные списки или карты.

- lower(string) - возвращает копию строки со всеми буквами Unicode, сопоставленными с их нижним регистром.
- map (key, value, ...) - возвращает карту, состоящую из пар ключ / значение, указанных в качестве аргументов. Каждый нечетный аргумент должен быть строковым ключом, а каждый четный аргумент должен иметь тот же тип, что и другие указанные значения. Повторяющиеся ключи не допускаются. Примеры:
  - ✓ map ("hello", "world")
  - ✓ map ("us-east", list ("a", "b", "c"), "us-west", list ("b", "c", "d"))
- matchkeys (values, keys, searchset) - для values двух спискови keys одинаковой длины возвращает все элементы из values которых вписке searchset существует соответствующий элемент из keys. Например, matchkeys (aws_instance.example.*.id, aws_instance.example.*.availability_zone, list("us-west-2a")) вернет список идентификаторов экземпляров экземпляров aws_instance.example в "us-west-2a" . Отсутствие совпадения приведет к пустому списку. Элементы keys обрабатываются последовательно, поэтому порядок возвращаемых values сохраняется.
- max(float1, float2, ...) - возвращает наибольшее из чисел с плавающей запятой.
- merge (map1, map2, ...) - возвращает объединение 2 или более карт. Карты используются в указанном порядке, и повторяющиеся ключи заменяют предыдущие записи.
  - ✓ ${merge (map ("a", "b"), map ("c", "d"))} возвращает {"a": "b", "c": "d"}
- min(float1, float2, ...) - возвращает наименьшее из чисел с плавающей запятой.
- md5(string) - возвращает (обычное) шестнадцатеричное представление хеша MD5 заданной строки.
- pathexpand(string) - возвращает строку пути к файлу срасширением ~ до домашнего каталога. Примечание: это создаст разницу планов между двумя разными хостами, если пути к файлам не совпадают.

- pow(x, y) - возвращает основание x экспоненты y в виде числа с плавающей запятой.

Пример:

- ✓ ${pow(3,2)} = 9
- ✓ ${pow(4,0)} = 1

- replace(string, search, replace) - выполняет поиск и замену в заданной строке. Все экземпляры search заменяются значением replace . Если search заключен в косую черту, он рассматривается как регулярное выражение. При использовании регулярного выражения replace может ссылаться на вложенные захваты в регулярном выражении с помощью $n, где n — это индекс или имя вложенного захвата. При использовании регулярного выражения синтаксис соответствует синтаксису регулярного выражения re2 .
- rsadecrypt (string, key) - Расшифровывает string с помощью RSA. Используется схема заполнения PKCS # 1 v1.5. string должна бытькодировке base64. key должен быть закрытым ключом RSA в формате PEM. Вы можете использовать file() чтобы загрузить его из файла.
- sha1(string) - возвращает (обычное) шестнадцатеричное представление хэша SHA-1 данной строки. Пример: "${sha1("${aws_vpc.default.tags.customer}-s3-bucket")}"
- sha256(string) - возвращает (обычное) шестнадцатеричное представление хэша SHA-256 данной строки. Пример: "${sha256("${aws_vpc.default.tags.customer}-s3-bucket")}"
- sha512(string) - возвращает (обычное) шестнадцатеричное представление хэша SHA-512 заданной строки. Пример: "${sha512("${aws_vpc.default.tags.customer}-s3-bucket")}"
- signum (integer) – возвращает -1 для отрицательных чисел, 0 для 0 и 1 для положительных чисел. Эта функция полезна, когда вам нужно установить значение для первого ресурса и другое значение для остальных ресурсов. Пример: element (split(",", var.r53_failover_policy), signum(count.index)) где 0-й индекс указывает на PRIMARY , а 1-й - на FAILOVER
- slice(list, from, to) - возвращает часть list между from (включительно) и to (включительно). Пример: slice(var.list_of_strings, 0, length(var.list_of_strings) - 1)
- sort(list) - возвращает лексографически отсортированный список строк, содержащихся в списке, переданном в качестве

аргумента. Сортировку можно использовать только со списками, которые содержат только строки. Примеры: sort(aws_instance.foo.*.id) , sort(var.list_of_strings)

- split(delim, string) - разделяет строку, созданную ранее join обратно в список. Это полезно для проталкивания списков через выходы модуля, поскольку в настоящее время они поддерживают только строковые значения. В зависимости от использования строку, в которой это выполняется, может потребоваться заключить в квадратные скобки, чтобы указать, что вывод фактически является списком, например, a_resource_param = ["${split(",", var.CSV_STRING)}"] . Пример: split(",", module.amod.server_ids)
- substr(string, offset, length) - извлекает подстроку из входной строки. Отрицательное смещение интерпретируется как эквивалент положительного смещения, измеренного назад от конца строки. Длина -1 интерпретируется как «до конца строки».
- timestamp() - возвращает строку отметки времени UTC в формате RFC 3339. Эта строка будет изменяться при каждом вызове функции, поэтому для предотвращения различий в каждом плане и применении ее необходимо использовать с ignore_changes жизненного цикла ignore_changes .
- timeadd(time, duration) - возвращает строку отметки time в формателUTC, соответствующую добавлению заданной duration ко времени в формате RFC 3339. Например, timeadd("2017-11-22T00:00:00Z", "10m") "2017-11-22T00:10:00Z" значение «2017-11-22T00: 10: 00Z» .
- title(string) - возвращает копию строки с заглавными буквами первых символов всех слов.
- transpose(map) - меняет местами ключи и значения списка на карте списков строк. Например, транспонирование (map ("a", list ("1", "2"), "b", list ("2", "3")) дает значение, эквивалентное map ("1", list («a»), «2», список («a», «б»), «3», список («б»)).
- trimspace(string) - возвращает копию строки с удаленными начальными и конечными пробелами.
- upper(string) - возвращает копию строки со всеми буквами Unicode, сопоставленными с их верхним регистром.
- urlencode(string) - возвращает URL-безопасную копию строки.

- uuid() - возвращает строку UUID в формате RFC 4122 v4. Эта строка будет изменяться при каждом вызове функции, поэтому для предотвращения различий в каждом плане и применении ее необходимо использовать с ignore_changes жизненного цикла ignore_changes .
- values(map) - возвращает список значений карты в порядке ключей, возвращаемыхфункцией keys . Эта функция работает только на плоских картах и возвращает ошибку для карт, содержащих вложенные списки или карты.
- zipmap(list, list) - создает карту из списка ключей и списка значений. Все ключи должны быть строкового типа, а длина списков должна быть одинаковой. Например, чтобы вывести сопоставление имен пользователей AWS IAM с отпечатком ключа, который использовался для шифрования их первоначального пароля, можно использовать: zipmap(aws_iam_user.users.*.name, aws_iam_user_login_profile.users.*.key_fingerprint) .

Шаблоны

Длинными строками можно управлять с помощью шаблонов. Шаблоны — это источники данных, определяемые именем файла и некоторыми переменными, которые используются во время интерполяции. У них есть вычисляемый атрибут rendered содержащий результат.

Похоже на источник данных шаблона:

```
data "template_file" "example" {
 template = "$${hello} $${world}!"
 vars {
  hello = "goodnight"
  world = "moon"
 }
}

output "rendered" {
 value = "${data.template_file.example.rendered}"
}
```

Тогда отображаемое значение будет goodnight moon! .

Вы можете использовать любую из встроенных функций в вашем шаблоне.

Использование шаблонов с графами

Вот пример, который объединяет возможности шаблонов с интерполяцией по count чтобы получить параметризованный шаблон, уникальный для каждого экземпляра ресурса:

```
variable "count" {
 default = 2
}

variable "hostnames" {
 default = {
  "0" = "example1.org"
  "1" = "example2.net"
 }
}

data "template_file" "web_init" {
 # Рендеринг шаблона один раз для каждого экземпляра
 count    = "${length(var.hostnames)}"
 template = "${file("templates/web_init.tpl")}"
 vars {
  # count.index сообщает нам индекс экземпляра, который мы обрабатываем.
  hostname = "${var.hostnames[count.index]}"
 }
}

resource "aws_instance" "web" {
 # Создайте по одному экземпляру для каждого имени хоста
 count     = "${length (var. hostnames)}"

 # Передаем каждому экземпляру соответствующий файл template_file
 user_data = "${data.template_file.web_init.*. rendered[count.index]}"
}
```

Таким образом, мы aws_instance.web список ресурсов данных template_file.web_init , который мы можем использовать в сочетании с нашим списком ресурсов aws_instance.web .

Математические операции

Математические операции могут быть выполнены в интерполяции:

```
variable "count" {
 default = 2
}

resource "aws_instance" "web" {
 # ...

 count = "${var. count}"

 # Отметьте экземпляр счетчиком, начинающимся с 1, т. е. web-001
 tags {
  Name = "${format ("web-%03d", count.index + 1)}"
 }
}
```

Поддерживаемые операции:

- *Сложение* (+), *вычитание* (-), *умножение* (*) и *деление* (/) для типов с **плавающей запятой**
- *Сложение* (+), *Вычитание* (-), *Умножение* (*), *Разделение* (/) и По *модулю* (%) для **целочисленных** типов

Приоритеты операторов — это стандартный математический порядок операций: *умножение* (*), *деление* (/) и *модуль* (%) имеют приоритет над *сложением* (+) и *вычитанием* (-). Круглые скобки могут использоваться для принудительного упорядочивания.

```
"${2 * 4 + 3 * 3}" # вычисляет до 17
"${3 * 3 + 2 * 4}" # вычисляет до 17
"${2 * (4 + 3) * 3}" # вычисляет до 42
```

Порядок загрузки и семантика

При вызове любой команды, которая загружает конфигурацию Terraform, Terraform загружает все конфигурационные файлы в каталоге, указанном в алфавитном порядке.

Загружаемые файлы должны иметь `.tf` или `.tf.json`, чтобы указать используемый формат. В противном случае файлы игнорируются. В одном каталоге могут находиться файлы разных форматов; это нормально, если один файл конфигурации Terraform будет иметь синтаксис Terraform, а другой - JSON.

Файлы переопределения являются исключением, поскольку они загружаются после всех файлов без переопределения в алфавитном порядке.

Конфигурации в загруженных файлах добавляются друг к другу. Это в отличие от слияния. Это означает, что два ресурса с одинаковым именем не объединяются, а вместо этого вызовут ошибку проверки. Это отличается от переопределений, которые действительно объединяются.

Порядок переменных, ресурсов и т. д., Определенных в конфигурации, не имеет значения. Конфигурации Terraform декларативны, поэтому ссылки на другие ресурсы и переменные не зависят от порядка, в котором они определены.

Локальные значение

Локальные значения присваивают имя выражению, которое затем может быть использовано несколько раз внутри модуля.

При сравнении модулей с функциями в традиционном языке программирования, если переменные аналогичны аргументам функции, а выходы аналогичны значениям, возвращаемым функцией, тогда *локальные значения* сопоставимы с локальными переменными функции.

Примеры

Локальные значения определены в блоках locals

```
# Идентификаторы нескольких наборов экземпляров EC2, объединенных вместе
locals {
  instance_ids = "${concat(aws_instance.blue.*.id, aws_instance.green.*.id)}"
}

# Вычисляемый префикс имени по умолчанию
locals {
  default_name_prefix = "${var.project_name}-web"
  name_prefix         = "${var.name_prefix != "" ? var.name_prefix : local.default_name_prefix}"
}

# Локальные значения могут быть интерполированы в другом месте с помощью "local". префикс.
resource "aws_s3_bucket" "files" {
  bucket = "${local.name_prefix}-files"
  # ...
```

```
}
```

Именованные локальные карты могут быть объединены с локальными картами для реализации общих или значений по умолчанию:

```
# Определить общие теги для всех ресурсов
locals {
 common_tags = {
  Component   = "awesome-app"
  Environment = "production"
 }
}

# Создайте ресурс, который сочетает общие теги с тегами, специфичными для экземпляра.
resource "aws_instance" "server" {
 ami          = "ami-123456"
 instance_type = "t2. micro"

 tags = "${merge (
  local.common_tags,
  map (
   "Name", "awesome-app-server",
   "Role", "server"
  )
 )}"
}
```

Описание

Блок locals определяет одну или несколько локальных переменных внутри модуля. Каждый блок locals может иметь столько локальных переменных, сколько необходимо, и в модуле может быть любое количество блоков locals.

Имена, данные элементам в блоке locals , должны быть уникальными во всем модуле. Данное значение может быть любым выражением, допустимым в текущем модуле.

Выражение локального значения может относиться к другим местным жителям, но, как обычно, циклы ссылок не допускаются. То есть локальное значение не может ссылаться на себя или на переменную, которая ссылается (прямо или косвенно) на него.

Рекомендуется сгруппировать логически связанные локальные значения в один блок, особенно если они зависят друг от друга. Это поможет читателю понять взаимосвязь между переменными. И наоборот, предпочитает, чтобы определить, *не связанные* локальные значения в *отдельных* блоках, и рассмотрят аннотирование каждого блока с комментарием, описывающим любым контекстом, общим для всех вложенных местных жителей.

Модули

Модули используются в Terraform для создания модулей и инкапсуляции групп ресурсов в вашей инфраструктуре.

Пример

```
module "consul" {
  source  = "hashicorp/consul/aws"
  servers = 5
}
```

Описание

Блок `module` инструктирует Terraform создать экземпляр модуля и, в свою очередь, создать экземпляры любых ресурсов, определенных в нем.

Имя, указанное в заголовке блока, используется для ссылки на конкретный экземпляр модуля из выражений внутри вызывающего модуля, а также для ссылки на модуль в командной строке.Оно не имеет значения за пределами конкретной конфигурации Terraform.

Внутри корпуса блока находится конфигурация модуля. Все атрибуты в блоке должны соответствовать переменным в модуле, за исключением следующих, которые Terraform рассматривает как особые:

- Source - (Обязательный). Строка источника модуля, определяющая расположение исходного кода дочернего модуля.
- Version - (Необязательно)Строка ограничения версии , указывающая, какие версии указанного модуля являются приемлемыми. Будет использоваться самая новая версия, соответствующая ограничению. version поддерживается только для модулей, полученных из реестров модулей.
- providers - (Необязательно) Карта, ключи которой являются именами конфигурации провайдера, ожидаемыми дочерним модулем, а значения - соответствующими именами провайдеров в вызывающем модуле. Это позволяет явно передавать конфигурации поставщика дочерним модулям . Если не указано иное, дочерний модуль наследует все конфигурации поставщика по умолчанию (без псевдонима) от вызывающего модуля.

Конфигурация вывода

Выходы определяют значения, которые будут выделены пользователю при применении Terraform, и их можно легко запросить с помощью команды output . Использование вывода более подробно описано в руководстве по началу работы . На этой странице описывается синтаксис конфигурации для выходов.

Компания Terraform знает много об инфраструктуре, которой она управляет. Большинство ресурсов имеют атрибуты, связанные с ними, а выходные данные позволяют легко извлекать и запрашивать эту информацию.

Пример

Простая конфигурация вывода выглядит следующим образом:

```
output "address" {
  value = "${aws_instance.db.public_dns}"
}
```

В результате будет выведено строковое значение, соответствующее публичному DNS адресу экземпляра AWS, определенного Terraform, с именем "db".Также возможно экспортировать сложные типы данных, такие как карты и списки:

```
output "addresses" {
  value = ["${aws_instance.web.*.public_dns}"]
}
```

Описание

Блок output настраивает одну выходную переменную. Несколько выходных переменных можно настроить с несколькими выходными блоками. NAME дается к выходному блоку это имя используется для ссылки на выходную переменную. Он должен соответствовать соглашениям об именах переменных Terraform, если он будет использоваться в качестве входных данных для других модулей.

Внутри блока ({ }) находится конфигурация вывода. Это параметры, которые можно установить:

- value (обязательно) - значение вывода. Это может быть строка, список или карта. Обычно это включает интерполяцию, поскольку статические выходные данные обычно бесполезны.
- description (необязательно) - понятное для человека описание вывода. Это в первую очередь для документации для пользователей, использующих вашу конфигурацию Terraform. В будущей версии Terraform эти описания будут отображаться как часть некоторой команды CLI Terraform.

- depends_on (список строк) - явные зависимости, которые имеет этот вывод. Эти зависимости будут созданы до обработки этого выходного значения. Зависимости имеют формат TYPE.NAME , например aws_instance.web.
- sensitive (необязательный, логический)

Syntax

Полный синтаксис:

```
output NAME {
  value = VALUE
}
```

Чувствительные выходы

Выходы можно пометить как содержащие конфиденциальный материал, установив для атрибута sensitive значение true , например:

```
output "sensitive" {
  sensitive = true
  value     = VALUE
}
```

Когда выходные данные отображаются на экране после применения terraform apply или terraform refresh , чувствительные выходные данные редактируются, а вместо их значения отображается <sensitive> .

Ограничения чувствительных выходов

- Значения чувствительных выходов по-прежнему хранятся в состоянии Terraform и доступны с помощью команды terraform output , поэтому на них нельзя полагаться как на единственное средство защиты значений.
- Чувствительность не отслеживается внутренне, поэтому, если выход в другом модуле интерполирован в ресурс, то значение будет отображено на экране.

Overrides

Terraform загружает все файлы конфигурации в каталог и складывает их вместе. Terraform также имеет концепцию переопределений , способ создания файлов, которые загружаются последними и объединяются в вашу конфигурацию, а не добавляются.

Замедления имеют несколько вариантов использования:

Машины (инструменты)могут создавать переопределения для изменения поведения Terraform без необходимости редактирования конфигурации Terraform с учетом человеческой читабельности.

Временные изменения могут быть внесены в конфигурации Terraform без необходимости изменения самой конфигурации.

Имена override должны быть переопределены или заканчиваться на _override , за исключением расширения. Примеры допустимых файлов переопределения: override.tf , override.tf.json , temp_override.tf .

Файлы переопределения загружаются последними в алфавитном порядке.

Файлы с переопределением могут быть в синтаксисе Terraform или в JSON,так же, как и непереопределяемые конфигурации Terraform.

Example

Если у вас есть файл конфигурации Terraform example.tf с содержимым:

```
resource "aws_instance" "web" {
  ami = "ami-408c7f28"
}
```

И вы создали файл override.tf с содержимым:

```
resource "aws_instance" "web" {
  ami = "foo"
}
```

Затем AMI для одного ресурса будет заменен на "foo".Обратите внимание, что синтаксис переопределения может быть синтаксисом Terraform или JSON. Вы можете смешивать и сопоставлять синтаксис без проблем.

Конфигурация ресурсов

Самое главное, что вы настроите с Terraform — это ресурсы. Ресурсы — это компонент вашей инфраструктуры. Это может быть какой-то низкоуровневый компонент, например, физический сервер, виртуальная машина или контейнер. Или это может быть компонент более высокого уровня, такой как провайдер электронной почты, DNS записи или базы данных.

На этой странице предполагается, что вы уже знакомы с синтаксисом конфигурации .

Example

Конфигурация ресурсов выглядит следующим образом:

```
resource "aws_instance" "web" {
  ami           = "ami-408c7f28"
  instance_type = "t1. micro"
}
```

Description

Блок resource создает ресурс данного TYPE (первый параметр) и NAME (второй параметр). Сочетание типа и имени должно быть уникальным.

Внутри блока ({ }) находится конфигурация ресурса. Конфигурация зависит от типа и документируется для каждого типа ресурса в разделе поставщиков .

Meta-parameters

Для всех ресурсов доступны **мета-параметры** :

- count int) - количество идентичных ресурсов для создания. Это относится не ко всем ресурсам.

В настоящее время модули не поддерживают параметр count .

- depends_on (список строк) - явные зависимости, которые имеет этот ресурс. Эти зависимости будут созданы перед этим ресурсом.
- provider (строка) - имя конкретного поставщика, который будет использоваться для этого ресурса. Имя имеет формат TYPE.ALIAS , например aws.west . Где west устанавливается с помощью атрибута alias в поставщике.
- lifecycle (блок конфигурации) - настраивает поведение жизненного цикла ресурса. Конкретные параметры описаны ниже.

Блок lifecycle позволяет установить следующие ключи:

✓ create_before_destroy (bool) - этот флаг используется, чтобы гарантировать, что замена ресурса будет создана до того, как исходный экземпляр будет уничтожен. Например, это можно использовать для создания новой записи DNS перед удалением старой записи.

✓ prevent_destroy (bool) - этот флаг обеспечивает дополнительную защиту от разрушения данного ресурса. Если для него установлено значение true , любой план, включающий уничтожение этого ресурса, вернет сообщение об ошибке.

✓ ignore_changes (список строк) - настраивает способ оценки различий для ресурсов, позволяя игнорировать отдельные атрибуты при изменении. Например, это можно использовать для игнорирования динамических изменений ресурса из внешних ресурсов. Остальные мета-параметры игнорировать нельзя.

Имена игнорируемых атрибутов могут совпадать по их имени, а не по идентификатору состояния. Например, если в aws_route_table определены два маршрута, а список ignore_changes содержит «маршрут», оба маршрута будут проигнорированы. Кроме того, вы также можете использовать одну запись с подстановочным знаком (например, "*"), которая будет соответствовать всем именам атрибутов. Использование частичной строки вместе с подстановочным знаком (например, "rout*") **не** поддерживается.

Интерполяции в настоящее время не поддерживаются в блоке конфигурации lifecycle (см. Проблему № 3116)

Timeouts

Отдельные ресурсы могут предоставлять блок timeouts позволяющий пользователям настраивать время, в течение которого определенная операция может пройти, прежде чем она будет считаться ошибкой. Например, ресурс aws_db_instance предоставляет настраиваемые тайм-ауты для операций create , update и delete . Любой ресурс, который предоставляет тайм-ауты, будет задокументировать значения по умолчанию для этой операции, и пользователи могут перезаписать их в своей конфигурации.

Пример перезаписи тайм-аутов create и delete :

```
resource "aws_db_instance" "timeout_example" {
  allocated_storage = 10
  engine            = "mysql"
  engine_version    = "5.6.17"
  instance_class    = "db.t1. micro"
  name              = "mydb"

  # ...

  timeouts {
    create = "60m"
    delete = "2h"
  }
}
```

Отдельные ресурсы должны дать согласие на предоставление настраиваемых тайм-аутов, и попытка настроить тайм-аут для ресурса, который не поддерживает тайм-ауты, или перезапись определенного действия, которое ресурс не указывает в качестве опции, приведет к ошибке. Допустимые единицы времени: s , m , h .

Явные зависимости

Терраформа обеспечивает успешное создание зависимостей до создания ресурса. Во время операции по уничтожению терраформа гарантирует, что этот ресурс будет уничтожен до того, как будут созданы его зависимости.

Ресурс автоматически зависит от всего, на что он ссылается посредством интерполяции . Автоматически определяемые зависимости — это все, что

требуется большую часть времени. Вы также можете использовать параметр depends_on , чтобы явно определить список дополнительных зависимостей.

Первичный вариант использования явного параметра depends_on — это зависимость от *побочного эффекта* другой операции. Например: если провайдер создает файл, и ваш ресурс читает этот файл, то для Terraform нет ссылки на интерполяцию для автоматического соединения двух ресурсов. Однако необходимо представить причинный порядок. Это идеальный случай для depends_on . Тем не менее, в большинстве случаев следует избегать использования depends_on и разрешить Terraform автоматически определять зависимости.

Синтаксис depends_on — это список ресурсов и модулей:

- Ресурсы - этоTYPE.NAME, например aws_instance.web .
- Модули имеют имя module.NAME , например module.foo .

Когда ресурс зависит от модуля, *все* в этом модуле должно быть создано до создания ресурса.

Пример ресурса, зависящего как от модуля, так и от ресурса, показан ниже. Обратите внимание, что depends_on может содержать любое количество зависимостей:

```
resource "aws_instance" "web" {
  depends_on = ["aws_instance.leader", "module.vpc"]
}
```

Используйте экономно! depends_on требуется редко. Почти в каждом случае автоматическая система зависимостей Terraform - лучший сценарий, когда ваши ресурсы зависят только от того, что они явно используют. Пожалуйста, подумайте, прежде чем использовать depends_on , чтобы определить, может ли Terraform автоматически сделать это лучше.

Блок соединения

Внутри ресурса вы можете дополнительно иметь **блок подключения** . Блоки подключения описывают Terraform, как подключиться к ресурсу для обеспечения . Этот блок не должен присутствовать, если вы используете только локальные средства обеспечения или если вы не выполняете подготовку вообще.

Ресурсы сами по себе предоставляют некоторые данные, например,IP-адрес, но другие данные должны быть указаны пользователем.

Полный список параметров, которые можно указать, указан на странице подключения провайдера .

Provisioners

Внутри ресурса вы можете указать ноль или более **блоков обеспечения** . Provisioner блоки настройки provisioners .

Внутри блока провайдеров находится конфигурация, специфичная для провайдера, как и конфигурация, специфичная для ресурса.

Блоки провайдеров могут также содержать соединительный блок (задокументированный выше).Этот блок подключения может использоваться для предоставления более конкретной информации о подключении для конкретного провайдера. Примером использования может быть использование другого пользователя для входа в систему для одного провайдера.

Использование переменных со count

При объявлении нескольких экземпляров ресурса с помощью count обычно требуется, чтобы каждый экземпляр имел разное значение для данного атрибута.

Для этого вы можете использовать интерполяцию ${count.index} вместе с переменной карты .

Например, вот как можно создать три инстанса AWS, каждый со своим статическим IP-адресом:

```
variable "instance_ips" {
  default = {
    "0" = "10.11.12.100"
    "1" = "10.11.12.101"
    "2" = "10.11.12.102"
  }
}

resource "aws_instance" "app" {
  count = "3"
  private_ip = "${lookup (var.instance_ips, count.index)}"
  # ...
}
```

Для ссылки на конкретный экземпляр ресурса вы можете использовать resource.foo.*.id[#] где # - порядковый номер экземпляра.

Например, чтобы создать список всех идентификаторов подсетей AWS и указать конкретную подсеть в списке, вы можете использовать следующий синтаксис:

```
resource "aws_vpc" "foo" {
  cidr_block = "198.18.0.0/16"
```

```
}

resource "aws_subnet" "bar" {
  count      = 2
  vpc_id     = "${aws_vpc.foo.id}"
  cidr_block = "${cidrsubnet(aws_vpc.foo.cidr_block, 8, count.index)}"
}

output "vpc_id" {
  value = "${aws_vpc.foo.id}"
}
output "all_subnet_ids" {
  value = "${aws_subnet.bar.*.id}"
}
output "subnet_id_0" {
  value = "${aws_subnet.bar.*.id[0]}"
}
output "subnet_id_1" {
  value = "${aws_subnet.bar.*.id[1]}"
}
```

Многочисленные обстоятельства поставщика

По умолчанию ресурс нацелен на поставщика в зависимости от его типа. Например, ресурс aws_instance будет нацелен на поставщика "aws". Начиная с Terraform 0.5.0, ресурс может нацеливаться на любого поставщика по имени.

Основным случаем использования для этого является целевая конфигурация провайдера, многократно настроенная на поддержку нескольких регионов и т. д.

Чтобы настроить таргетинг на другого поставщика, установите поле provider :

```
resource "aws_instance" "foo" {
   provider = "aws.west"

   # ...
}
```

Значение поля должно быть TYPE или TYPE.ALIAS. Значение ALIAS берется из значения поля alias при настройке поставщика.

```
provider "aws" {
  alias = "west"
```

```
  # ...
}
```

Если поле provider не указано, используется поставщик по умолчанию.

Syntax

Полный синтаксис:

```
resource TYPE NAME {
   CONFIG ...
   [count = COUNT]
   [depends_on = [NAME, ...]]
   [provider = PROVIDER]
   [LIFECYCLE]

   [CONNECTION]
   [PROVISIONER ...]
}
```

где CONFIG :

```
KEY = VALUE

KEY {
   CONFIG
}
```

где LIFECYCLE :

```
lifecycle {
   [create_before_destroy = true|false]
   [prevent_destroy = true|false]
   [ignore_changes = [ATTRIBUTE NAME, ...]]
}
```

где CONNECTION :

```
connection {
   KEY = VALUE
   ...
}
```

где PROVISIONER :

```
provisioner NAME {
   CONFIG ...
   [when = "create"|"destroy"]
   [on_failure = "continue"|"fail"]
```

[CONNECTION]

}

Конфигурация терраформа

Раздел конфигурации terraform используется для настройки самого Terraform, например, для выполнения конфигурации требуется минимальная версия Terraform.

На этой странице предполагается, что вы уже знакомы с синтаксисом конфигурации .

Example

Конфигурация терраформы выглядит следующим образом:

terraform {

required_version = "> 0.7.0"

}

Description

Блок terraform настраивает поведение самого Terraform.

В настоящее время в этом блоке разрешены только конфигурации required_version и backend .

required_version указывает набор ограничений версии, которые должны соблюдаться для выполнения операций с этой конфигурацией. Если запущенная версия Terraform не соответствует этим ограничениям, отображается ошибка. См. Раздел ниже, посвященный этой опции.

См движков для получения дополнительной информации о backend конфигурации.

Никакое значение в блоке terraform не может использовать интерполяцию. Блок terraform загружается очень рано при выполнении Terraform, и интерполяции еще не доступны.

Указание требуемой версии терраформы

Параметр required_version может использоваться для требования определенной версии Terraform. Если запущенная версия Terraform не соответствует указанным ограничениям, Terraform покажет ошибку и завершит работу.

Когда модули используются все требования Terraform версии , указанные в полном дереве модуля, должны отрабатываться неудовлетворительно. Это означает, что параметр required_version может использоваться модулем, чтобы требовать, чтобы все потребители модуля также использовали определенную версию.

Значение этой конфигурации представляет собой список ограничений, разделенных запятыми. Ограничение — это оператор, за которым следует версия, например > 0.7.0 . Ограничения поддерживают следующие операции:

- = (или без оператора): точное равенство версий
- != : версия не равна
- > , >= , < , <= : сравнение версий, где «больше» — это больший номер версии.
- ~> : оператор пессимистического ограничения. Пример: для ~> 0.9 это означает >= 0.9, < 1.0 . Пример: для ~> 0.8.4 это означает >= 0.8.4, < 0.9

Для модулей рекомендуется минимальная версия, например > 0.8.0 . Эта минимальная версия гарантирует, что модуль работает должным образом, но дает потребителю возможность использовать более новые версии.

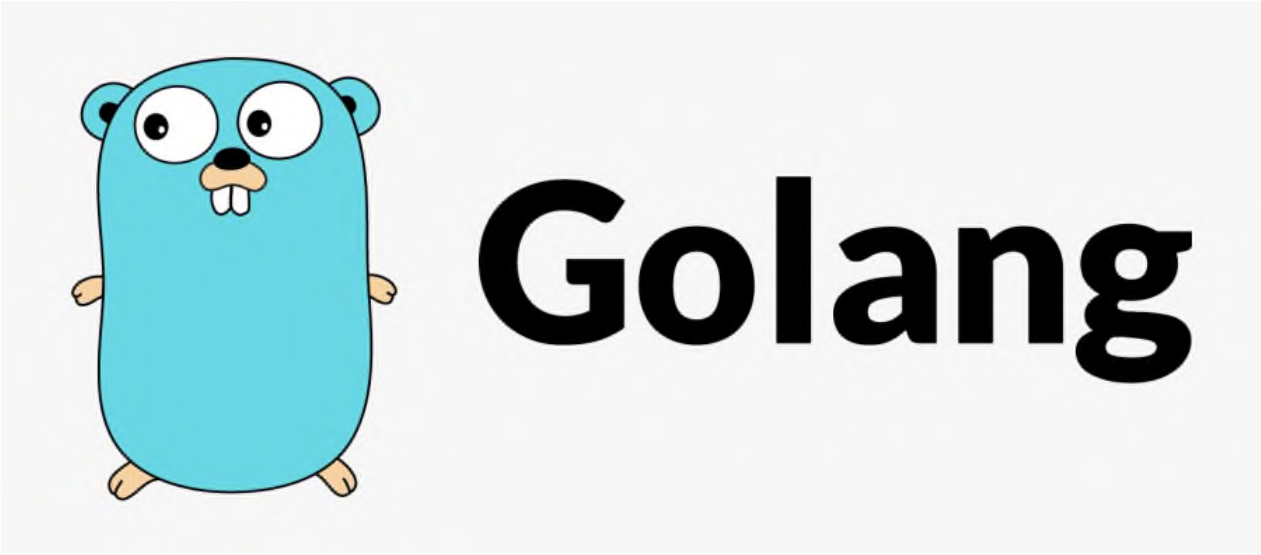
Syntax

Полный синтаксис:

```
terraform {
  required_version = VALUE
}
```

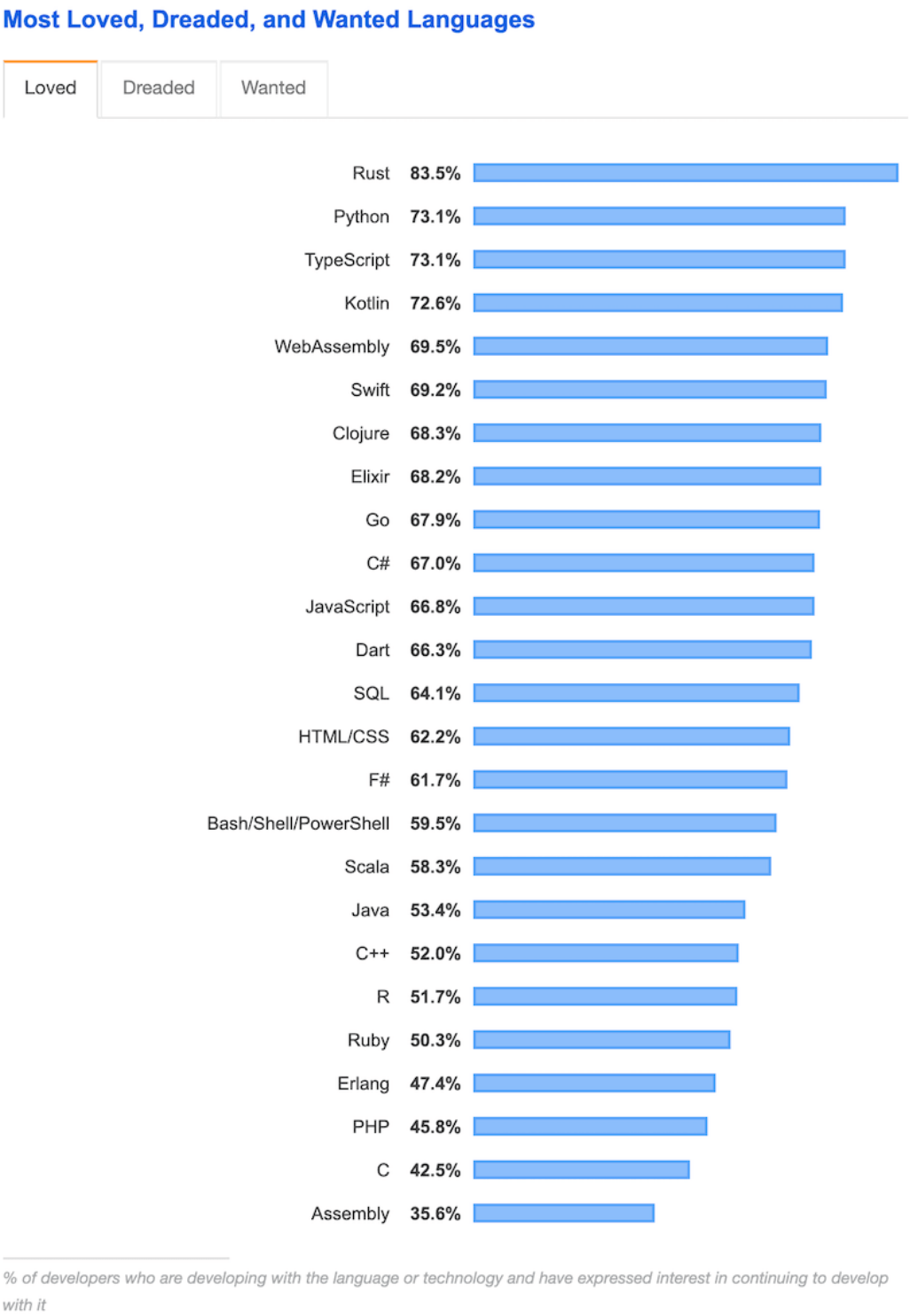
3. Введение в Golang

Golang , также известный как Go, является языком программирования с открытым исходным кодом, созданным Google в 2007 году. С момента своего изобретения Go с годами сильно растет. Golang имеет минимальный синтаксис, очень похожий на другие языки сценариев. Он был создан для повышения производительности труда разработчиков, что позволило им лучше писать программы.



Почему Golang?

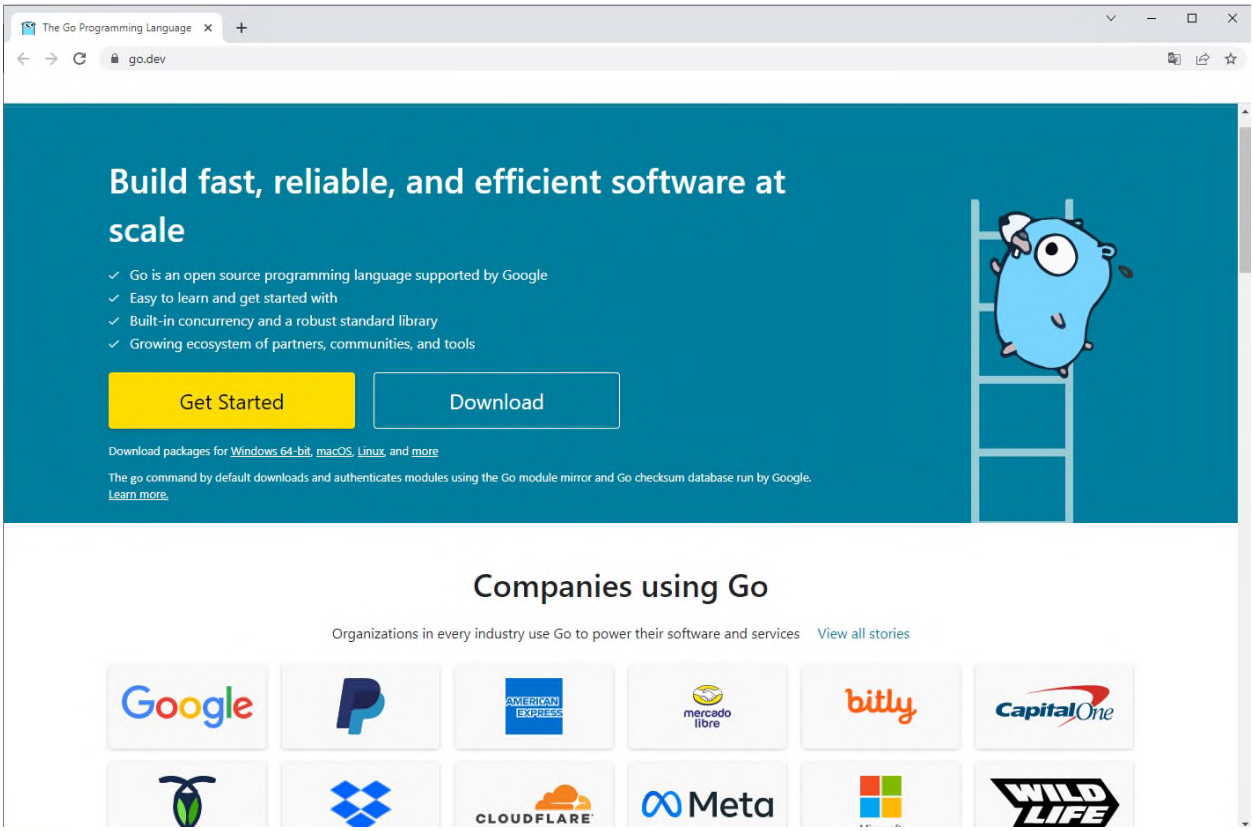
- Чистый, лаконичный и эффективный
- Минимальный и простой в освоении
- Быстрый (используется такими компаниями, как Google, Uber и т. д.)
- Простые двоичные файлы, используемые на сервере
- Поддерживает сборку мусора
- Golang является одним из самых любимых языков многих программистов



Установка

Go — это открытый и бесплатный язык программирования. Разработка Go началась в сентябре 2007 года, его непосредственным проектированием занимались Роберт Гризмер, Роб Пайк и Кен Томпсон, занимавшиеся до этого проектом разработки операционной системы Inferno. Вы можете установить его через скомпилированные двоичные файлы или собрать его из исходного кода. Сегодня мы будем устанавливать его через доступные двоичные файлы.

Скачать дистрибутив можно на официальном сайте Golang



Далее вы можете проверить, успешно ли установлен Go, выполнив следующую команду:

```
go version
```

В случае, если у вас есть какие-то сомнения, вся информация, связанная с загрузкой и установкой, доступна на golang.org.

Первая программа Hello World на Golang

Теперь, когда мы настроили Golang, мы готовы написать самую простую и известную программу **Hello World**.

- Давайте создадим простой файл **helloworld.go**. Да, наверное, вы уже догадались, что golang имеет расширение .go.
- Теперь откройте ваш любимый редактор кода и скопируйте следующий код.

```
package main
 import "fmt"

 func main() {
   fmt.Println("Hello, World! I am learning Golang!")
 }
```

- Сохраните файл и перейдите к терминалу. Прежде чем объяснить, давайте запустим вашу первую программу на Go!

- В терминале (убедитесь, что вы находитесь в том же каталоге, где вы создали файл **helloworld.go**)

```
go run helloworld.go
```

Та-да! Вы увидите следующий вывод:

```
// Output
 Hello, World! I am learning Golang!
```

Поздравляем! Вы только что успешно выполнили свою первую программу на GO.

Давайте разберемся с основным кодом.

Теперь давайте рассмотрим программу более детально. Программы на Go читаются сверху вниз, слева направо (как книга). Первая строка гласит:

```
package main
```

Это называется «определение пакета». Любая Go программа начинается с определения имени пакета. Пакеты — это подход Go к организации повторного использования кода. Есть два типа программ на Go: исполняемые файлы и разделяемые библиотеки. Исполняемые являются видом программ, которые можно запустить прямо из терминала (в Windows их имя заканчивается на .exe). Библиотеки являются коллекциями кода, который можно использовать из других программ. Детальнее мы будем рассматривать библиотеки чуть позже, а пока просто не забудьте включать эту строку в программы, которые вы пишете. Дальше следует пустая строка. Компьютер представляет новые строки специальным символом (или несколькими символами). Символы новой строки, пробелы и символы табуляции называются отступами. Go не обращает на них внимания, но мы используем их, чтобы облегчить себе чтение программы. (Вы можете удалить эту строку и убедится, что программа ведет себя в точности как раньше.)

Дальше следует это:

```
import "fmt"
```

Ключевое слово `import` позволяет подключить сторонние пакеты для использования их функциональности в нашей программе. Пакет `fmt` (сокращение от `format`) реализует форматирование для входных и выходных данных. Учитывая то, что мы только что узнали о пакетах, как вы думаете, что будет содержаться в верхней части файлов пакета `fmt`? Обратите внимание, что `fmt` взят в двойные кавычки. Использование двойных кавычек называется «строковым литералом», который в свою очередь является видом «выражения». Строки в Go представляют собой набор символов (букв, чисел, …) определенной длины. Строки мы рассмотрим детально в следующей главе,

а сейчас главное иметь в виду, что за открывающим символом " в конечном итоге должен последовать и закрывающий. Всё, что находится между ними, будет являться строкой (символ " сам по себе не является частью строки). Строка, начинающаяся с // известна как комментарий. Комментарии игнорируются компилятором Go и служат исключительно пояснениями вам (или тем, кто будет потом читать ваш код). Go поддерживает два вида комментариев: // превращает в комментарий весь текст до конца строки, и /* */, где комментарием является всё, что содержится между символами * (включая переносы строк). Далее можно увидеть объявление функции:

```
func main () {
fmt.Println("Hello World")
}
```

Функции являются кирпичиками программы на Go. Они имеют входы, выходы и ряд действий, называемых операторами, расположенных в определенном порядке. Любая функция начинается с ключевого слова func, за которым следует имя функции (в нашем случае main), список из нуля и более параметров, возвращаемый тип (если есть) и само «тело», заключенное в фигурные скобки. Наша функция не имеет входных параметров, ничего не возвращает и содержит всего один оператор. Имя main является особенным, эта функция будет вызываться сама при запуске программы.

Заключительной частью нашей программы является эта строка:

```
fmt.Println("Hello World")
```

Этот оператор содержит три части. Во-первых: доступ к функции пакета fmt под названием Println (Print line), затем создается новая строка, содержащая Hello World и вызывается функция с этой строкой в качестве первого и единственного аргумента. На данный момент вы уже можете быть немного перегружены от количества новой терминологии. Иногда полезно сознательно читать вашу программу вслух. Программу, которую мы только что написали можно прочитать следующим образом:

Создается новая исполняемая программа, которая использует библиотеку fmt *и содержит функцию* main*. Эта функция не содержит аргументов, ничего не возвращает и делает следующее: использует функцию* Println *из библиотеки* fmt *и вызывает её, используя один аргумент — строку* Hello World*.*

Функция Println выполняет основную работу в этой программе. Вы можете узнать о ней больше, набрав в терминале команду:

```
godoc fmt Println
```

Среди прочей информации вы должны увидеть это:

> Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.

Go — очень хорошо документированный язык, но эту документацию бывает трудно понять, если вы доселе не были знакомы с другими языками программирования. Тем не менее, команда godoc является хорошим местом для начала поиска ответов на возникающие вопросы. Сейчас документация говорит нам, что вызов `Println` пошлет передаваемые ей данные на стандартный вывод — терминал, вы сейчас работаете в нём. Эта функция является причиной, по которой `Hello World` отображается на экране

Типы

В предыдущей главе мы использовали строковый тип данных, чтобы хранить Hello World. Типы данных определяют множество принимаемых значений, описывают, какие операции могут быть применены к ним, и определяют, как данные будут храниться. Поскольку типы данных могут быть сложны для понимания, мы попробуем рассмотреть их подробнее, прежде чем разбираться, как они реализованы в Go.

Предположим, у вас есть собака по имени Шарик. Тут «Шарик» — это «Собака», этот тип описывает какой-то набор свойств, присущий всем собакам. Наши рассуждения должны быть примерно следующие: у собак 4 лапы, Шарик — собака, значит, у Шарика 4 лапы. Типы данных в языках программирования работают похожим образом: у всех строк есть длина; x — строка, а значит у x есть длина.

В математике мы часто говорим о множествах. Например, $\mathbb{R}$ (множество всех вещественных чисел) или $\mathbb{N}$ (множество всех натуральных чисел). Каждый элемент этих множеств имеет такие же свойства, как и все прочие элементы этого множества. Например, все натуральные числа ассоциативны - «для всех натуральных чисел a, b и c выполняется: a + (b + c) = (a + b) + c и a × (b × c) = (a × b) × c»; в этом смысле множества схожи с типами данных в языках программирования тем, что все значения одного типа имеют общие свойства.

Go — это язык программирования со статической типизацией. Это означает, что переменные всегда имеют определенный тип и этот тип нельзя изменить. Статическая типизация, на первый взгляд, может показаться неудобной. Вы потратите кучу времени только на попытки исправить ошибки, не позволяющие программе скомпилироваться. Однако типы дают вам

возможность понять, что именно делает программа, и помогают избежать распространённых ошибок.

В Go есть несколько встроенных типов данных, с которыми мы сейчас ознакомимся.

Числа

В Go есть несколько различных типов для представления чисел. Вообще, мы разделим числа на два различных класса: целые числа и числа с плавающей точкой.

Целые числа

Целые числа точно так же, как их математические коллеги, — это числа без вещественной части. В отличие от десятичного представления чисел, которое используем мы, компьютеры используют двоичное представление.

Наша система строится на 10 различных цифрах. Когда мы исчерпываем доступные нам цифры, мы представляем большое число, используя новую цифру 2 (а затем 3, 4, 5, …) числа следуют одно за другим. Например, число, следующее за 9, это 10, число, следующее за 99, это 100 и так далее. Компьютеры делают то же самое, но они имеют только 2 цифры вместо 10. Поэтому, подсчет выглядит так: 0, 1, 10, 11, 100, 101, 110, 111 и так далее. Другое отличие между той системой счисления, что используем мы, и той, что использует компьютер - все типы чисел имеют строго определенный размер. У них есть ограниченное количество цифр. Поэтому четырехразрядное число может выглядеть так: 0000, 0001, 0010, 0011, 0100. В конце концов мы можем выйти за лимит, и большинство компьютеров просто вернутся к самому началу (что может стать причиной очень странного поведения программы).

В Go существуют следующие типы целых чисел: uint8, uint16, uint32, uint64, int8, int16, int32 и int64. 8, 16, 32 и 64 говорит нам, сколько бит использует каждый тип. uint означает «unsigned integer» (беззнаковое целое), в то время как int означает «signed integer» (знаковое целое). Беззнаковое целое может принимать только положительные значения (или ноль). В дополнение к этому существуют два типа-псевдонима: byte (то же самое, что uint8) и rune (то же самое, что int32). Байты — очень распространенная единица измерения в компьютерах (1 байт = 8 бит, 1024 байта = 1 килобайт, 1024 килобайта = 1 мегабайт, …), и именно поэтому тип byte в Go часто используется для определения других типов. Также существует 3 машинно-зависимых целочисленных типа: uint, int и uintptr. Они машинно-зависимы, потому что их размер зависит от архитектуры используемого компьютера.

В общем, если вы работаете с целыми числами — просто используйте тип int.

Числа с плавающей точкой

Числа с плавающей точкой — это числа, которые содержат вещественную часть (вещественные числа) (1.234, 123.4, 0.00001234, 12340000). Их представление в компьютере довольно сложно и не особо необходимо для их использования. Так что мы просто должны помнить:

- Числа с плавающей точкой неточны. Бывают случаи, когда число вообще нельзя представить. Например, результатом вычисления 1.01–0.99 будет 0.020000000000000018 - число очень близкое к ожидаемому, но не то же самое.
- Как и целые числа, числа с плавающей точкой имеют определенный размер (32 бита или 64 бита). Использование большего размера увеличивает точность (сколько цифр мы можем использовать для вычисления)
- В дополнение к числам существуют несколько других значений, таких как: «not a number» (не число) (NaN, для вещей наподобие 0/0), а также положительная и отрицательная бесконечность (+∞ и −∞).

В Go есть два вещественных типа: float32 и float64 (соответственно, часто называемые вещественными числами с одинарной и двойной точностью). А также два дополнительных типа для представления комплексных чисел (чисел с мнимой частью): complex64 и complex128. Как правило, мы должны придерживаться типа float64, когда работаем с числами с плавающей точкой.

Пример

Давайте напишем программу-пример, использующую числа. Во-первых, создайте папку «chapter3» с файлом main.go внутри со следующим содержимым:

```
package main

import "fmt"

func main() {
   fmt.Println("1 + 1 = ", 1 + 1)
}
```

Если вы запустите программу, то должны увидеть это:

```
$ go run main.go
1 + 1 = 2
```

Заметим, что эта программа очень схожа с программой, которую мы написали в главе 2. Она содержит ту же строку с указанием пакета, ту же

строку с импортом, то же определение функции и использует ту же функцию Println. В этот раз вместо печати строки Hello World мы печатаем строку 1 + 1 = с последующим результатом выражения 1 + 1. Это выражение состоит из трех частей: числового литерала 1 (который является типом int), оператора + (который представляет сложение) и другого числового литерала 1. Давайте попробуем сделать то же самое, используя числа с плавающей точкой:

```
fmt.Println("1 + 1 =", 1.0 + 1.0)
```

Обратите внимание, что мы используем .0, чтобы сказать Go, что это число с плавающей точкой, а не целое. При выполнении этой программы результат будет тот же, что и прежде.

В дополнение к сложению, в Go имеется несколько других операций:

Литерал	**Пояснение**
+	сложение
-	вычитание
*****	умножение
/	деление
%	остаток от деления

Строки

Как мы видели в главе 2, строка — это последовательность символов определенной длины, используемая для представления текста. Строки в Go состоят из независимых байтов, обычно по одному на каждый символ (символы из других языков, таких как китайский, представляются несколькими байтами).

Строковые литералы могут быть созданы с помощью двойных кавычек "Hello World" или с помощью апострофов `Hello World`. Различие между ними в том, что строки в двойных кавычках не могут содержать новые строки и они позволяют использовать особые управляющие последовательности символов. Например, \n будет заменена символом новой строки, а \t - символом табуляции.

Распространенные операции над строками включают в себя нахождение длины строки len("Hello World"), доступ к отдельному символу в строке "Hello World"[1], и конкатенацию двух строк "Hello " + "World". Давайте модифицируем созданную ранее программу, чтобы проверить всё это:

```
package main

import "fmt"
```

```
func main() {
   fmt.Println(len("Hello World"))
   fmt.Println("Hello World"[1])
   fmt.Println("Hello " + "World")
}
```

На заметку:

• Пробел тоже считается символом, поэтому длина строки 11 символов, а не 10 и третья строка содержит "Hello " вместо "Hello".

• Строки “индексируются” начиная с 0, а не с 1. [1] даст вам второй элемент, а не первый. Также заметьте, что вы видите 101 вместо e, когда выполняете программу. Это происходит из-за того, что символ представляется байтом (помните, байт — это целое число).

Можно думать об индексации так: "Hello World" 1. Читайте это так: «строка Hello World позиция 1», «на 1 позиции строки Hello World» или «второй символ строки Hello World».

• Конкатенация использует тот же символ, что и сложение. Компилятор Go выясняет, что должно происходить, полагаясь на типы аргументов. Если по обе стороны от + находятся строки, компилятор предположит, что вы имели в виду конкатенацию, а не сложение (ведь сложение для строк бессмысленно).

Логические типы

Булевский тип (названный так в честь Джорджа Буля) — это специальный однобитный целочисленный тип, используемый для представления истинности и ложности. С этим типом используются три логических оператора:

Литерал	**Пояснение**
&&	И
||	ИЛИ
!	НЕ

Вот пример программы, показывающей их использование:

```
func main() {
   fmt.Println(true && true)
   fmt.Println(true && false)
   fmt.Println(true || true)
   fmt.Println(true || false)
   fmt.Println(!true)
}
```

Запуск этой программы должен вывести:

```
$ go run main.go
true
false
true
true
false
```

Используем таблицы истинности, чтобы определить, как эти операторы работают:

Выражение	**Значение**
true && true	true
true && false	false
false && true	false
false && false	false

Выражение	**Значение**
true || true	true
true || false	true
false || true	true
false || false	false

Выражение	**Значение**
!true	false
!false	true

Всё это — простейшие типы, включенные в Go и являющиеся основой, с помощью которой строятся все остальные типы.

Переменные

Ранее в этой книге мы имели дело с литеральными значениями (числами, строками и т. д.), но программы с одними только литералами фактически бесполезны. Для того, чтобы сделать по-настоящему полезные программы, нам нужно узнать о двух важных вещах: переменных и инструкциях, управляющих ходом выполнения. В этой главе будут рассмотрены переменные.

Переменная — это именованное место хранения какого-то типа данных. Давайте изменим программу, которую мы написали в главе 2 так, чтобы там использовались переменные.

```
package main

import "fmt"
```

```
func main() {
   var x string = "Hello World"
   fmt.Println(x)
}
```

Обратите внимание, что мы по-прежнему используем строковый литерал из оригинальной программы, но вместо того, чтобы напрямую передать его в функцию Println, мы присваиваем его переменной. Переменные в Go создаются с помощью ключевого слова var, за которым следуют имя переменной (x), тип (string) и присваиваемое значение (Hello World). Последний шаг не обязателен, поэтому программа может быть переписана так:

```
package main

import "fmt"

func main() {
   var x string
   x = "Hello World"
   fmt.Println(x)
}
```

Переменные в Go похожи на переменные в алгебре, но есть несколько различий. Во-первых, когда мы видим символ =, то по привычке читаем его как «x равен строке Hello World». Нет ничего неверного в том, чтобы читать программу таким образом, но лучше читать это как «x принимает значение строки Hello World» или «x присваивается строка Hello World». Это различие важно потому, что переменные могут менять свои значения во время выполнения программы (как понятно по их названию). Попробуйте сделать следующее:

```
package main

import "fmt"

func main() {
   var x string
   x = "first"
   fmt.Println(x)
```

```
    x = "second"
    fmt.Println(x)
}
```

На самом деле вы можете сделать даже так:

```
var x string
x = "first "
fmt.Println(x)
x = x + "second"
fmt.Println(x)
```

Эта программа будет бессмысленной, если вы будете читать её как теорему из алгебры. Но она обретет смысл, если вы будете внимательно читать программу как список команд. Когда мы видим x = x + "second", то должны читать это так: «Присвоить конкатенацию значения переменной x и литерала строки переменной x». Операции справа от = выполняются первыми, и результат присваивается левой части.

Запись x = x + y настолько часто встречается в программировании, что в Go есть специальный оператор присваивания +=. Мы можем записать x = x + "second" как x += "second", и результат будет тем же (прочие операторы могут быть использованы подобным же образом).

Другое отличие между Go и алгеброй в том, что для равенства используется другой символ: == (два знака равно, один за другим). == - это оператор. Как и +, он возвращает логический тип. Например:

```
var x string = "hello"
var y string = "world"
fmt.Println(x == y)
```

Эта программа напечатает false, потому что hello отличается от world. С другой стороны:

```
var x string = "hello"
var y string = "hello"
fmt.Println(x == y)
```

напечатает true, потому что обе строки одинаковы.

Если мы хотим присвоить значение переменной при её создании, то можем использовать сокращенную запись:

```
x := "Hello World"
```

Обратите внимание на то, что : стоит перед =, и на отсутствие типа. Тип в данном случае указывать не обязательно, так как компилятор Go способен определить тип по литералу, которым мы инициализируем переменную. Тут

мы присваиваем строку, поэтому x будет иметь тип string. Компилятор может определить тип и при использовании var:

```
var x = "Hello World"
```

И так со всеми типами:

```
x := 5
fmt.Println(x)
```

В общем, желательно всегда использовать краткий вариант написания.

Как назвать переменную

Правильное именование переменных — важная часть разработки ПО. Имена должны начинаться с буквы и могут содержать буквы, цифры и знак _ (знак подчеркивания). Компилятору Go, в принципе, всё равно, как вы назовете переменную, но не забудьте, что вам (и может быть кому-то еще) потом это придется читать. Предположим, у нас есть:

```
x := "Max"
fmt.Println("My dog's name is", x)
```

В этом случае x не самое лучшее имя переменной. Лучше было бы так:

```
name := "Max"
fmt.Println("My dog's name is", name)
```

или даже так:

```
dogsName := "Max"
fmt.Println("My dog's name is", dogsName)
```

В последнем случае мы использовали специальный способ написания имени переменной, состоящей из нескольких слов, известный как lower CamelCase (или camelBack). Первая буква первого слова записывается в нижнем регистре, первая буква последующих слов записывается в верхнем регистре, всё остальное - в нижнем.

Область видимости

Вернемся к программе, которую мы рассматривали в начале главы:

```
package main

import "fmt"

func main() {
   var x string = "Hello World"
   fmt.Println(x)
}
```

Эту программу можно записать следующим образом:

```
package main

import "fmt"

var x string = "Hello World"

func main() {
   fmt.Println(x)
}
```

Мы вынесли переменные за пределы функции main. Это означает, что теперь другие функции имеют доступ к этой переменной:

```
var x string = "Hello World"

func main() {
   fmt.Println(x)
}

func f() {
   fmt.Println(x)
}
```

Функция f имеет доступ к переменной x. Теперь предположим, что вместо этого мы написали:

```
func main() {
   var x string = "Hello World"
   fmt.Println(x)
}

func f() {
   fmt.Println(x)
}
```

Если вы попробуете выполнить эту программу, то получите ошибку:

```
.\main.go:11: undefined: x
```

Компилятор говорит вам, что переменная x внутри функции f не существует. Она существует только внутри функции main. Места, где может использоваться переменная x, называются областью видимости переменной. Согласно спецификации «в Go область видимости ограничена блоками». В

основном это значит, что переменные существуют только внутри текущих фигурных скобок { } (в блоке), включая все вложенные скобки (блоки). Область видимости поначалу может запутать вас, но когда вы увидите больше примеров, то всё станет ясно.

Константы

Go также поддерживает константы. Константы — это переменные, чьи значения не могут быть изменены после инициализации. Они создаются таким же образом, как и переменные, только вместо var используется ключевое слово const:

```
package main

import "fmt"

func main() {
   const x string = "Hello World"
   fmt.Println(x)
}
```

А вот этот код:

```
const x string = "Hello World"
x = "Some other string"
```

вызовет ошибку компиляции:

```
.\main.go:7: cannot assign to x
```

Константы — хороший способ использовать определенные значения в программе без необходимости писать их каждый раз. Например, константа Pi из пакета math.

Определение нескольких переменных

В Go существует еще одно сокращение на случай, если необходимо определить несколько переменных:

```
var (
   a = 5
   b = 10
   c = 15
)
```

Используя ключевые слово var (или const), за которым идут круглые скобки с одной переменной в каждой строке.

Пример программы

```
package main
```

```
import "fmt"

func main() {
   fmt.Print("Enter a number: ")
   var input float64
   fmt.Scanf("%f", &input)

   output := input * 2

   fmt.Println(output)
}
```

Тут мы используем другую функцию из пакета fmt, чтобы считать пользовательский ввод (Scanf). &input будет объяснен в следующих главах, а все, что нам нужно знать сейчас, это то, что Scanf заполняет переменную input числом, введенным нами.

4. Написание собственных провайдеров для Terraform

В Terraform «поставщик» — это логическая абстракция вышестоящего API. В этом руководстве подробно описано, как создать собственный поставщик для Terraform.

Почему?

Существует несколько возможных причин для создания пользовательского поставщика Terraform, например:

- Внутреннее частное облако, функциональность которого либо является собственностью, либо не приносит пользы сообществу открытого исходного кода.
- Поставщик «в процессе работы» проходит локальное тестирование, прежде чем внести свой вклад.
- Расширения существующего провайдера

Локальная установка

Terraform поддерживает модель плагинов, и все провайдеры на самом деле являются плагинами. Плагины распространяются в виде бинарных файлов Go. Хотя технически возможно написать плагин на другом языке, почти все плагины Terraform написаны на Go .

Схема провайдера

Для начала создайте файл с именем provider.go. Это корень провайдера, который должен включать следующий шаблонный код:

```
package main


import (
   "github.com/hashicorp/terraform/helper/schema"
)


func Provider() *schema.Provider {
   return &schema.Provider{
      ResourcesMap: map[string]*schema.Resource{},
   }
}
```

Библиотека helper/schema является частью ядра Terraform. Он абстрагируется от многих сложностей и обеспечивает согласованность между поставщиками. В приведенном выше примере определяется пустой провайдер (нет *ресурсов*).

Тип *schema.Providerописывает свойства поставщика, включая:

- ключи конфигурации, которые он принимает
- ресурсы, которые он поддерживает
- любые обратные вызовы для настройки

Сборка плагина

Для Go требуется main.go файл, который является исполняемым файлом по умолчанию при сборке двоичного файла. Поскольку плагины Terraform распространяются в виде бинарных файлов Go, важно определить эту точку входа с помощью следующего кода:

```
package main


import (
   "github.com/hashicorp/terraform/plugin"
   "github.com/hashicorp/terraform/terraform"
)


func main() {
   plugin.Serve(&plugin.ServeOpts{
      ProviderFunc: func() terraform.ResourceProvider {
         return Provider()
      },
   })
}
```

Это устанавливает основную функцию для создания действительного исполняемого бинарного файла Go. Содержимое функции main потребляет plugin библиотеку Terraform. Эта библиотека отвечает за всю связь между ядром Terraform и плагином.

Затем создайте плагин с помощью набора инструментов Go:

```
$ go build -o terraform-provider-example
```

Выходное имя (-o) **очень важно** . Terraform ищет плагины в формате:

```
terraform-<TYPE>-<NAME>
```

В приведенном выше случае подключаемый модуль имеет тип «поставщик» и имя «пример».

Чтобы убедиться, что все работает правильно, запустите только что созданный двоичный файл:

```
$ ./terraform-provider-example
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically
```

Это базовая структура проекта и каркас для плагина Terraform. Напомним, файловая структура такова:

```
.
├── main.go
└── provider.go
```

Определение ресурсов

Поставщики Terraform управляют ресурсами. Провайдер — это абстракция вышестоящего API, а ресурс — это компонент этого провайдера. Например, провайдер AWS поддерживает aws_instan сеи aws_elastic_ip. DNSimple поддерживает dnsimple_record. Быстро поддерживает fastly_service. Добавим ресурс к нашему вымышленному провайдеру.

Как правило, поставщики Terraform помещают каждый ресурс в свой собственный файл, названный в честь ресурса с префиксом resource_. Чтобы создать example_server, это будет resource_server.go по соглашению:

```
package main


import (
   "github.com/hashicorp/terraform/helper/schema"
)
```

```
func resourceServer() *schema.Resource {
   return &schema.Resource{
      Create: resourceServerCreate,
      Read:   resourceServerRead,
      Update: resourceServerUpdate,
      Delete: resourceServerDelete,

      Schema: map[string]*schema.Schema{
         "address": &schema.Schema{
            Type:    schema.TypeString,
            Required: true,
         },
      },
   }
}
```

Здесь используется тип `schema.Resource` . Эта структура определяет схему данных и операции CRUD для ресурса. Определение этих свойств — единственное, что требуется для создания ресурса.

Схема выше определяет один элемент `"address"`, который является обязательной строкой. Схема Terraform автоматически обеспечивает проверку и приведение типов.

Далее определяются четыре «поля» — `Create`, `Read`, `Update`и `Delete`. Функции `Create`, `Read`и `Delete` необходимы для того, чтобы ресурс работал. Есть и другие функции, но это единственные необходимые. Terraform сам решает, какую функцию вызывать и с какими данными. На основе схемы и текущего состояния ресурса Terraform может определить, нужно ли создать новый ресурс, обновить существующий или уничтожить.

Каждое из четырех полей структуры указывает на функцию. Хотя технически возможно встроить все функции в схему ресурса, наилучшая практика требует включения каждой функции в отдельный метод. Это

оптимизирует как для тестирования, так и для удобства чтения. Заполните эти заглушки сейчас, обращая пристальное внимание на сигнатуры методов.

```
func resourceServerCreate(d *schema.ResourceData, m interface{}) error {
    return nil
}

func resourceServerRead(d *schema.ResourceData, m interface{}) error {
    return nil
}

func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
    return nil
}

func resourceServerDelete(d *schema.ResourceData, m interface{}) error {
    return nil
}
```

Наконец, обновите схему провайдера, provider.go чтобы зарегистрировать этот новый ресурс.

```
func Provider() *schema.Provider {
    return &schema.Provider{
        ResourcesMap: map[string]*schema.Resource{
            "example_server": resourceServer(),
        },
    }
}
```

Соберите и протестируйте плагин. Все должно компилироваться как есть, хотя все операции не выполняются.

```
$ go build -o terraform-provider-example


$ ./terraform-provider-example
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically
```

Теперь макет выглядит так:

```
.
├── main.go
├── provider.go
├── resource_server.go
└── terraform-provider-example
```

Вызов провайдера

В предыдущих разделах был показан запуск провайдера напрямую через оболочку, которая выводит предупреждающее сообщение, например:

```
This binary is a plugin. These are not meant to be executed directly.
Please execute the program that consumes these plugins, which will
load any plugins automatically
```

Плагины Terraform должны выполняться непосредственно Terraform. Чтобы проверить это, создайте файл main.tf в рабочем каталоге (там же, где находится плагин).

```
resource "example_server" "my-server" {}
```

И выполнить `terraform plan`:

```
$ terraform plan


1 error(s) occurred:


* example_server.my-server: "address": required field is not set
```

Это подтверждает, что Terraform правильно делегирует работу нашему плагину и что наша проверка работает должным образом. Исправьте ошибку проверки, добавив addressполе в ресурс:

```
resource "example_server" "my-server" {
  address = "1.2.3.4"
}
```

Выполните terraform plan, чтобы убедиться, что проверка проходит:

```
$ terraform plan


+ example_server.my-server
    address: "1.2.3.4"


Plan: 1 to add, 0 to change, 0 to destroy.
```

Можно запустить terraform apply, но это будет бесполезно, потому что все параметры ресурсов в настоящее время не предпринимают никаких действий.

Реализовать Create

Вернувшись в resource_server.go, реализуйте функциональность создания:

```
func resourceServerCreate(d *schema.ResourceData, m interface{}) error {
    address := d.Get("address").(string)
    d.SetId(address)
    return nil
}
```

Это использует API schema.ResourceData для получения значения, "address" предоставленного пользователем в конфигурации Terraform. Из-за того, как работает Go, мы должны преобразовать его в строку. Однако это безопасная операция, поскольку наша схема гарантирует, что это будет строковый тип.

Затем он использует SetId встроенную функцию для установки идентификатора ресурса в адрес. Существование непустого идентификатора говорит Terraform о том, что ресурс был создан. Этот идентификатор может

быть любым строковым значением, но это должно быть значение, которое можно использовать для повторного чтения ресурса.

Перекомпилируйте бинарник, запуск terraform plan и terraform apply.

```
$ go build -o terraform-provider-example
# ...
$ terraform plan

+ example_server.my-server
    address: "1.2.3.4"

Plan: 1 to add, 0 to change, 0 to destroy.
$ terraform apply

example_server.my-server: Creating...
  address: "" => "1.2.3.4"
example_server.my-server: Creation complete (ID: 1.2.3.4)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Поскольку используется Create операция SetId, Terraform считает, что ресурс создан успешно. Убедитесь в этом, запустив terraform plan.

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

example_server.my-server: Refreshing state... (ID: 1.2.3.4)
No changes. Infrastructure is up-to-date.
```

```
This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, Terraform
doesn't need to do anything.
```

Опять же, из-за вызова SetId Terraform считает, что ресурс был создан. При запуске plan Terraform правильно определяет, что нет никаких изменений для применения.

Чтобы проверить это поведение, измените значение `address` поля и запустите `terraform plan` снова. Вы должны увидеть такой вывод:

```
$ terraform plan
example_server.my-server: Refreshing state... (ID: 1.2.3.4)


~ example_server.my-server
    address: "1.2.3.4" => "5.6.7.8"


Plan: 0 to add, 1 to change, 0 to destroy.
```

Terraform обнаруживает изменение и отображает разницу с `~`префиксом, отмечая, что ресурс будет изменен на месте, а не создан новый.

Запустите `terraform apply`, чтобы применить изменения.

```
$ terraform apply
example_server.my-server: Refreshing state... (ID: 1.2.3.4)
example_server.my-server: Modifying... (ID: 1.2.3.4)
  address: "1.2.3.4" => "5.6.7.8"
example_server.my-server: Modifications complete (ID: 1.2.3.4)


Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Поскольку мы не реализовали эту `Update` функцию, можно было бы ожидать, что `terraform plan` операция сообщит об изменениях, но это не так! Как наши изменения сохранялись без `Update` реализации?

Обработка ошибок и частичное состояние

Ранее наша Update операция завершалась успешно, и новое состояние сохранялось с пустым определением функции. Напомним текущую функцию обновления:

```
func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
  return nil
}
```

Сообщает return nil Terraform, что операция обновления прошла успешно без ошибок. Terraform предполагает, что это означает, что любые запрошенные изменения применяются без ошибок. Из-за этого наше состояние обновилось, и Terraform считает, что дальнейших изменений нет.

Другими словами, если обратный вызов не возвращает ошибок, Terraform автоматически предполагает, что весь diff успешно применен, объединяет diff в финальное состояние и сохраняет его.

Функции *никогда не* должны вызываться намеренно panic или вызываться os.Exit— всегда возвращать ошибку.

На самом деле, это немного сложнее, чем это. Представьте себе сценарий, в котором наша функция обновления должна обновлять два отдельных поля, что требует двух отдельных вызовов API. Что делать, если первый вызов API прошел успешно, а второй — нет? Как правильно указать Terraform сохранять только половину различий? Это известно как сценарий с *частичным состоянием* , и его правильная реализация имеет решающее значение для корректного поведения поставщика.

Вот правила обновления состояния в Terraform. Обратите внимание, что здесь упоминаются обратные вызовы, которые мы не обсуждали для полноты картины.

- Если Create обратный вызов возвращается с ошибкой или без нее без идентификатора, установленного с помощью SetId, предполагается, что ресурс не создан, и состояние не сохраняется.
- Если Create обратный вызов возвращается с ошибкой или без нее и был установлен идентификатор, предполагается, что ресурс создан, и все состояние сохраняется вместе с ним. Повторяю, потому что это важно: если есть ошибка, но установлен ID, то состояние полностью сохраняется.
- Если Update обратный вызов возвращается с ошибкой или без нее, сохраняется полное состояние. Если идентификатор становится пустым, ресурс уничтожается (даже при обновлении, хотя этого не должно происходить, за исключением сценариев ошибок).

- Если Destroy обратный вызов возвращается без ошибки, предполагается, что ресурс уничтожен, а все состояние удалено.
- Если Destroy обратный вызов возвращается с ошибкой, предполагается, что ресурс все еще существует, и все предыдущее состояние сохраняется.
- Если частичный режим (рассматривается далее) включен при возврате создания или обновления, сохраняются только явно включенные ключи конфигурации, что приводит к частичному состоянию.

Частичный режим — это режим, который можно включить с помощью обратного вызова, который сообщает Terraform, что частичное состояние возможно. Когда этот режим включен, провайдер должен явно указать Terraform, что безопасно сохранять, а что нет.

Вот пример частичного режима с функцией обновления:

```
func resourceServerUpdate(d *schema.ResourceData, m interface{}) error {
  // Enable partial state mode
  d.Partial(true)

  if d.HasChange("address") {
    // Try updating the address
    if err := updateAddress(d, meta); err != nil {
      return err
    }

    d.SetPartial("address")
  }

  // If we were to return here, before disabling partial mode below,
  // then only the "address" field would be saved.

  // We succeeded, disable partial mode. This causes Terraform to save
  // save all fields again.
```

```
    d.Partial(false)


    return nil
}
```

Примечание. Этот код не будет компилироваться, так как в нем нет `updateAddress` функции. Вы можете реализовать фиктивную версию этой функции, чтобы поиграть с частичным состоянием. Для этого примера частичное состояние не имеет большого значения в этом примере документации. В случае сбоя `updateAddress` поле адреса не будет обновлено.

Реализация Destroy

Обратный `Destroy` вызов именно так и звучит — он вызывается для уничтожения ресурса. Эта операция никогда не должна обновлять какое-либо состояние ресурса. Нет необходимости вызывать `d.SetId("")`, так как любое возвращаемое значение без ошибки предполагает, что ресурс был успешно удален.

```
func resourceServerDelete(d *schema.ResourceData, m interface{}) error {
  // d.SetId("") is automatically called assuming delete returns no errors, but
  // it is added here for explicitness.
    d.SetId("")
    return nil
}
```

Функция уничтожения всегда должна обрабатывать случай, когда ресурс уже может быть уничтожен (например, вручную). Если ресурс уже уничтожен, это не должно возвращать ошибку. Это позволяет пользователям Terraform вручную удалять ресурсы без нарушения работы Terraform.

```
$ go build -o terraform-provider-example
```

Запустите `terraform destroy`, чтобы уничтожить ресурс.

```
$ terraform destroy
Do you really want to destroy?
  Terraform will delete all your managed infrastructure.
  There is no undo. Only 'yes' will be accepted to confirm.
```

```
 Enter a value: yes


example_server.my-server: Refreshing state... (ID: 5.6.7.8)
example_server.my-server: Destroying... (ID: 5.6.7.8)
example_server.my-server: Destruction complete


Destroy complete! Resources: 1 destroyed.
```

Реализация Read

Обратный `Read` вызов используется для синхронизации локального состояния с фактическим состоянием (вверх по течению). Это вызывается в различных точках Terraform и должна быть операцией только для чтения. Этот обратный вызов никогда не должен изменять реальный ресурс.

Если идентификатор обновляется до пустого, это сообщает Terraform, что ресурс больше не существует (возможно, он был уничтожен вне очереди). Так же, как и обратный вызов уничтожения, `Read` функция должна изящно обрабатывать этот случай.

```
func resourceServerRead(d *schema.ResourceData, m interface{}) error {
 client := meta.(*MyClient)


 // Attempt to read from an upstream API
 obj, ok := client.Get(d.Id())


 // If the resource does not exist, inform Terraform. We want to immediately
 // return here to prevent further processing.
 if !ok {
  d.SetId("")
  return nil
 }
```

```
    d.Set("address", obj.Address)
    return nil
}
```

5. Развертывание инфраструктуры на платформе Базис.DigitalEnergy с применением Terraform

Обзор облачной платформы DECORT

Основные понятия

Ниже перечислены основные понятия с указанием соответствующих им аргументов в Terraform провайдере DECORT.

1. **Контроллер облачной платформы DECORT** – управляющее приложение, которое обеспечивает авторизацию пользователей и оркестрацию облачных ресурсов.

- Адрес контроллера задается в обязательном аргументе controller_url на стадии инициализации Terraform провайдера DECORT. Например, controller_url= "https://ds1.digitalenergy.online"

2. **Авторизационный провайдер** – приложение, работающее по протоколу Oauth2, предназначенное для выпуска и валидации токенов доступа к контроллеру облачной платформы в соответствующих режимах авторизации. Все действия в платформе должны выполняться авторизованными пользователями, и авторизационное приложение позволяет получить токен доступа, действующий некоторое ограниченное время, наличие которого подтверждает успешную авторизацию.

- Адрес авторизационного провайдера задается в аргументеoauth2_url на стадии инициализации Terraform провайдера DECORT. Например, oauth2_url= "https://sso.digitalenergy.online"

3. **Подписчик** (*account*) – сущность, которая используется для группирования облачных ресурсов по принадлежности к определенному клиенту для целей учета потребления и биллинга.

- Имя подписчика задается аргументом account_name при вызове *resource* или *data* функций провайдера. Альтернативной является задание численного идентификатора подписчика в аргументе account_id.

4. **Пользователь** (*user*) – пользователь облачной инфраструктуры, представленный учетной записью. Чтобы получить возможность управлять облачными ресурсами (например, создавать виртуальные серверы или дискт) пользователь должен быть ассоциирован с одним или несколькими подписчиками и иметь соответствующие права, определяемые ролевой моделью, принятой в облачной платформе DECORT. Для доступа к платформе пользователь должен авторизоваться одним из способов, описанных ниже в разделе «Способы авторизации».

5. **Ресурсная группа** (*resource group*) – способ группирования вычислительных ресурсов (например, виртуальных серверов по функциональному признаку или принадлежности к одному и тому же проекту). Ресурсную группу можно рассматривать как небольшой персональный дата-центр, в котором размещаются один или несколько серверов и виртуальных сетевых сегментов. Ресурсная группа идентифицируется по комбинации параметров account и name. Обратите внимание, что имя имя ресурсной группы уникально только в рамках одного и того же account.

6. **Вычислительный ресурс** (*compute*) - универсальная абстракция пользовательского сервера в платформе DECORT. Благодаря использованию такой абстракции можно, например, создать одну виртуальную машину на базе KVM Intel x86, а другую - на базе KVM IBM Power, а потом управлять ими - изменять количество CPU/RAM, подключать/отключать диски и т.п. - одинаковым образом, не задумываясь об их архитектурных различиях. В то же время, так как типизация ресурсов в Terraform не поддерживает наследование, различные типы вычислительных ресурсов, доступных на платформе DECORT и абстрагируемых через понятие унифицированный *compute*, в Terraform представлены разными типами (напр., свой тип для виртуальных серверов на базе KVM и свой тип для перспективных x86-совместимых bare metal серверов).

7. **Ресурс хранения** (*disk*) - универсальная абстракция дискового ресурса в платформе DECORT. Платформа поддерживает различные типы систем хранения данных, но при этом управление созданными на разных системах хранения дисками осуществляется посредством унифицированного набора действий, например, "подключить диск к *compute*", "увеличить размер диска", "сделать мгновенный снимок диска", "настроить параметры быстродействия диска".

8. **Виртуальный сервер** – экземпляр *compute*, в основе технической реализации которого лежит виртуальная машина, работающая в облаке DECORT и доступна по сети. Виртуальный сервер характеризуется количеством выделенных ему CPU (аргументcpu), объемом ОЗУ (ram), размером загрузочного диска (boot_disk size). При создании виртуального сервера на загрузочный диск устанавливается образ операционной системы, заданный в аргументе image_id. Помимо загрузочного диска к виртуальному серверу можно подключить несколько дисков для хранения прикладных данных, список которых задается аргументами extra_disks. Виртуальный сервер идентифицируется по комбинации аргументов name (имя сервера) и rgid

(идентификатор ресурсной группы). Обратите внимание, что имя виртуального сервера name уникально только в рамках одной и той же ресурсной группы.

9. **Виртуальный сетевой сегмент** (*Virtual Network Segment* или *ViNS*) - сетевой сегмент и обеспечивающая его функционирование виртуальная инфраструктура, которые пользователь может создавать для своих нужд на уровне ресурсной группы или подписчика (*account*). ViNS можно создать полностью изолированным от внешних сетей (см. ниже *External Network*) или с подключением во внешнюю сеть. Внутри ViNS работает DHCP-сервис, обеспечивающий управление IP адресами экземпляров *compute*, подключённых в этот ViNS.

10. **Внешняя сеть** (*External Network*) - сетевой сегмент, через который платформа DECORT взаимодействует с внешними по отношению к ней сетевыми ресурсами. Например, в случае с публичным облаком на базе DECORT в качестве внешней сети выступает сеть Интернет. В отличие от ViNS платформа не управляет внешней сетью, а лишь пользуется её ресурсами. В платформе может быть настроено несколько внешних сетей с различными диапазонами IP адресов, и существует механизм управления доступом пользователей к внешним сетям.

11. Сетевой доступ к экземпляру *compute* (виртуальному серверу) реализуется через его подключение к ViNS и/или прямое подключение во внешнюю сеть (External Network). Один и тот же экземпляр *compute* может одновременно иметь несколько подключений в разные ViNS и/или различные внешние сети.

Способы авторизации

Облачная платформа DECORT поддерживает два базовых типа авторизации:

1. С использованием авторизационного провайдера, работающего по протоколу *Oauth2*. Данный способ является предпочтительным, так как обеспечивает бОльшую гибкость и безопасность. Для авторизации в этом режиме при инициализации Terrafrom провайдера DECORT необходимо указать параметры oauth2_url и controller_url, а также предоставить одно из нижеперечисленного:

   - Комбинация Application ID & Application secret, соответствующих пользователю, от имени которого будет осуществляться управление облачными ресурсами в текущей сессии. В процессе проверки предоставленных Application ID & Application secret модуль получает от авторизационного провайдера токен (JSON Web Token, JWT), который затем

используется для доступа к указанному контроллеру DECORT. Для авторизации по данному варианту, при инициализации Terraform провайдера DECORT следует установить аргумент authenticator=oauth2 и задать аргументы app_id и app_secret (или определить соответствующие переменные окружения DECORT_APP_ID и DECORT_APP_SECRET).

- JSON Web Token – заранее полученный от авторизационного провайдера токен доступа, ассоциированный с определенным пользователем, от имени которого будет осуществляться управление облачными ресурсами в текущей сессии. Для авторизации по данному варианту, при инициализации Terraform провайдера DECORT следует установить аргумент authenticator=jwt и задать аргумент jwt (или определить переменную окружения DECORT_JWT).

2. С использованием комбинации *имя пользователя : пароль*. Данный режим не использует внешних авторизационных провайдеров и подразумевает, что пользователь с такой комбинацией зарегистрирован непосредственно на указанном в параметре controller_url контроллере облачной платформы DECORT.

- Чтобы провайдер авторизовался по данному варианту, при его инициализации следует установить аргумент authenticator=legacy и задать аргументы user и password (или определить соответствующие переменные окружения DECORT_USER и DECORT_PASSWORD).

После успешной авторизации пользователь (или приложение-клиент) получает доступ к ресурсам, находящимся под управлением соответствующего DECORT контроллера. Доступ предоставляется в рамках подписчиков (*account*), с которыми ассоциирован данный пользователь (*user*), и в соответствии с присвоенными ему ролями.

Обзор Terraform провайдера DECORT

Назначение

Terraform провайдер DECORT предназначен для управления облачными ресурсами на платформе Digital Energy Cloud Orchestration Technology (DECORT). Основные функции провайдера:

1. Создание виртуальных серверов на базе KVM - экземпляров *compute* на базе виртуальных машин архитектуры Intel x86 или IBM Power (см. *resource* функцию **decort_kvmvm**).
2. Создание ресурсных групп (см. *resource* функцию **decort_resgroup**).
3. Создание ресурсов хранения *disk* (см. *resource* функцию **decort_disk**).
4. Создание образов (см. *resource* функцию **decort_image**)

5. Создание виртуальных образов (см. *resource* функцию **decort_virtual_image**)

6. Создание cdrom - образов (см. *resource* функцию **decort_cdrom_image**)

7. Удаление нескольких образов (см. *resource* функцию **decort_delete_images**)

8. Создание правил трансляции сетевых портов для виртуальных серверов (см. *resource* функцию **decort_pfw**)

9. Создание виртуальных сетевых сегментов (см. *resource* функцию **decort_vins**)

10. Создание снапшотов (см. *resource* функцию **decort_snapshot**)

11. Создание кластера kubernetes (см. *resource* функцию **decort_k8s**)

12. Создание worker group кластера (см. *resource* функцию **decort_k8s_wg**)

13. Создание sep (см. *resource* функцию **decort_sep**)

14. Создание sep config (см. *resource* функцию **decort_sep_config**)

15. Создание pcidevice (см. *resource* функцию **decort_pcidevice**)

16. Получение информации о подписчиках (см. *data source* функцию **decort_account**).

17. Получение информации о существующих виртуальных серверах на базе KVM (см. *data source* функцию **decort_kvmvm**).

18. Получение информации о существующих дисковых ресурсах *disk* (см. *data source* функцию **decort_disk**).

19. Получение информации о существующих виртуальных сетевых сегментах *ViNS* (см. *data source* функцию **decort_vins**).

20. Получение информации о существующих ресурсных группах (см. *data source* функцию **decort_resgroup**).

21. Получение информации об образах ОС, на базе которых можно создавать новые вычислительные ресурсы (см. *data source* функцию **decort_image**).

22. Получение информации обо всех образах ОС, на базе которых можно создавать новые вычислительные ресурсы (см. *data source* функцию **decort_image_lst**)

23. Получение информации о всех доступных снапшотах (см. *data source* функцию **decort_snapshot_list**)

24. Получение информации о всех доступных платформах (см. *data source* функцию **decort_grid_list**)

25. Получение информации о доступной платформе (см. *data source* функцию **decort_grid**)

26. Получение информации о доступных для образа stacks (см. *data source* функцию **decort_image_stacks**)

27. Получение информации о sep (см. *data source* функцию **decort_sep**)

28. Получение информации о всех доступных sep (см. *data source* функцию **decort_sep_list**)

29. Получение информации о конфигурации sep (см. *data source* функцию **decort_sep_config**)

30. Получение информации о дисках, используемых sep (см. *data source* функцию **decort_sep_disk_list**)

31. Получение информации о загрузке sep (см. *data source* функцию **decort_sep_consumption**)

32. Получение информации о sep pool (см. *data source* функцию **decort_sep_pool**)

33. Получение информации о pcidevice (см. *data source* функцию **decort_pcidevice**)

34. Получение информации о всех доступных pcidevice (см. *data source* функцию **decort_pcidevice_list**)

35. Получение информации о виртуальном графическом устройстве vgpu (см. *data source* функцию **decort_vgpu**)

Системные требования

Для запуска провайдера вам потребуется компьютер, на котором установлен Terraform. Провайдер тестировался вместе с Terraform 0.xx.x на системах под управлением ОС семейства Linux и персональных компьютерах под управлением Windows 10.

Кроме того, в связи с тем, что начиная с версии 0.12 Terraform изменил алгоритм поиска и инициализации локальных провайдеров, настройка данного провайдера для работы с Terraform 0.12 или более новыми версиями потребует выполнения ряда дополнительных действий. Подробнее см. Настройка локального провайдера для работы с новыми версиями Terraform.

Установка

Terraform провайдер представляет собой бинарный исполняемый файл. Чтобы быстро начать пользоваться Terraform провайдером DECORT, поместите собранный для вашей целевой платформы исполняемый файл в ту же папку, где находится ваш файл *main.tf*.

Обычно, провайдер устанавливается автоматически из terraform registry. Для этого достаточно в файл main.tf добавить директивы определения провайдера.

```
provider "decort" {
 authenticator = "oauth2"
 #controller_url = <DECORT_CONTROLLER_URL>
 controller_url = "https://mr4.digitalenergy.online"
 #oauth2_url = <DECORT_SSO_URL>
 oauth2_url          = "https://sso.digitalenergy.online"
 allow_unverified_ssl = true
}
```

При выполнении команды terraform init провайдер будет автоматически установлен.

После этого вы можете использовать стандартные команды Terraform-а для управления ресурсами в соответствии с декларациями, содержащимися в main.tf, а именно:

```
terraform init
terraform plan
terraform apply
```

Data source функции Terraform провайдера DECORT.

Data source функции в составе Terraform провайдера DECORT служат для read-only доступа к ресурсам облачной платформы.

Типовой сценарий использования Data source функций - получение информации о характеристиках уже существующих облачных ресурсов (например, идентификатора ресурсной группы или идентификатора образа ОС) и использование этой информации при создании новых ресурсов (так для создания нового виртуального сервера требуется предоставить идентификатор образа ОС).

- decort_resgroup - получение информации о существующей ресурсной группе (*resource group*).

- decort_kvmvm - получение информации о существующем виртуальном сервере (экземпляре *compute*, созданном на базе системы виртуализации KVM).
- decort_disk - получение информации о существующем ресурсе хранения (*disk*).
- decort_vins - получение информации о существующем виртуальном сетевом сегменте (*Virtual Network Segment*, *vins*).
- decort_account - получение информации о подписчике (*account*).
- decort_image_list - получение информации об образах (*image*), *необходимы права администратора*
- decort_image - получение информации об образе (*image*), например, о дисковом ресурсе с образом операционной системы, *необходимы права администратора*.
- decort_grid_list - получение списка с информацией о доступных платформах, *необходимы права администратора*
- decort_grid - получение информации о доступной платформе, *необходимы права администратора*
- decort_image_list_stacks - получение информации о доступном stacks по id образа, *необходимы права администратора*
- decort_snapshot_list - получение списка снапшотов Для *управления* ресурсами облачной платформы предназначены *Resource* функции (см. соответствующий раздел).

Resource функции Terraform провайдера DECORT.

Resource функции в составе Terraform провайдера DECORT служат для управления облачными ресурсами - создания, удаления или изменения их характеристик.

- decort_resgroup - управление ресурсными группами.
- decort_kvmvm - управление виртуальными серверами, создаваемыми на базе системы виртуализации KVM.
- decort_disk - управление ресурсами хранения.
- decort_vins - управление виртуальными сетевыми сегментами (Virtual Network Segment, ViNS).
- decort_k8s - управление кластером
- decort_k8s_wg - управление worker group кластера
- decort_image - управление образом, *требуются права администратора*
- decort_virtual_image - управление виртуальным образом, *требуются права администратора*

- decort_cdrom_image - управление cdrom образом, *требуются права администратора*
- decort_delete_images - удаление образов
- decort_snapshot - управление снапшотами.
- decort_pfw - управление правилами трансляции сетевых портов для серверов, подключенных в виртуальный сетевой сегмент.

Если вам требуется read-only доступ к характеристиками уже существующих облачных ресурсов, то для этих целей предназначены Data source функции.



Модуль 12. Конфигурационное управление. Что такое IaC.

- Введение в Ansible
- Работа с Playbook
- Работа с Roles
- Введение в Python
- Создание собственных Modules
- Создание инфраструктуры в Базис. DigitalEnergy с использованием Ansible

1. Введение в Ansible

Представьте себе, что вам нужно управлять парком серверов, расположенных к тому же в разных географических точках. Каждый из этих серверов требует настройки, регулярного обновления и мониторинга. Конечно, для решения этих задач можно воспользоваться самым простым способом: подключиться к каждому серверу по **ssh** и внести необходимые изменения. При всей своей простоте этот способ сопряжен с некоторыми трудностями: он чрезвычайно трудоемок, а на выполнение однообразных операций уходит очень много времени.

Чтобы упростить процессы настройки и конфигурирования серверов, можно также писать **shell-скрипты**, но и этот способ вряд ли можно назвать совершенным. Скрипты нужно постоянно изменять, подстраивая их под каждую новую задачу. При их написании необходимо учитывать различие операционных систем и версий. Не будем забывать и о том, что отладка скриптов отнимает много усилий и забирает немало времени.

Оптимальным вариантом решения описанных проблем является внедрение системы удаленного управления конфигурацией. В таких системах достаточно лишь описать нужное состояние управляемого узла. Система должна сама определить, что нужно сделать для достижения этого состояния, и осуществит все необходимые действия.

Со всеми сложностями, о которых идет речь выше, вы наверняка хорошо знакомы на собственном опыте: у вас имеется несколько десятков серверов, расположенных в разных точках планеты. На них необходимо регулярно вносить различные изменения: обновлять операционную систему, устанавливать и обновлять различное программное обеспечение, изменять конфигурацию и тому подобное.

Предлагается все эти операции автоматизировать и внедрить систему удаленного управления конфигурациями.

Что такое Ansible?

Ansible — программное решение для удаленного управления конфигурациями, разработанное *Майклом Де Хаанном* в 2012 году. Название продукта взято из научно-фантастической литературы: в романах американской писательницы *Урсулы Ле Гуин* была такая штука, как **Ansible** — это устройство для оперативной космической связи.

Ansible — мощное программное обеспечение автоматизации конфигурирования, с открытым исходным кодом, управления и

развертывания приложений на узлах без каких-либо простоев, для работы которого потребуется только **SSH**. В отличие от подобных продуктов, **Ansible** устанавливается на единственном хосте, который может даже быть вашей локальной машиной и использует **SSH** для связи с каждым удаленным узлом. Это позволяет ему быть невероятно быстрым при конфигурировании новых серверов, поскольку не требуются предварительно установленные дополнительные пакеты на каждом новом сервере.

Машина управления, где установлен **Ansible** и **Узлы**, управляемая этой машиной по **SSH**. Местоположение узлов определяется управляющей машиной через её инструментарий. **Ansible** не требует установки клиентской части или приложения, что означает отсутствие необходимости какой-либо установки агента на удаленных узлах, так что это означает, что нет каких-либо фоновых демонов или программ, выполняемых для **Ansible**, когда он не управляет узлами.

Другими словами, **Ansible** — программное обеспечение для централизованного управления конфигурациями, то есть другими операционными системами и установленными на них программами. Это современный инструмент управления конфигурацией, который облегчает задачу настройки и обслуживания удаленных серверов.

Ansible берет на себя всю работу по приведению удаленных серверов в необходимое состояние. Администратору необходимо лишь описать, как достичь этого состояния с помощью так называемых сценариев — специальных файлов «**playbook**».

В них описывается желаемое состояние управляемой системы, например, необходимо наличие пакета **Midnight Commander**. **Ansible** проверяет, соответствует ли удаленный компьютер описанию в плейбуке, и если это не так, приводит его в должный вид. Формат для **playbook** — **YAML**. Для описания задачи задается ее имя, используемый модуль и список параметров.

Такая технология позволяет очень быстро осуществлять переконфигурирование системы, для этого достаточно всего лишь добавить несколько новых строк в сценарий.

Пример playbook:

```
---
- name: Nginx web server
  hosts: web-servers
```

```
remote_user: root
tasks:
- name: Installs nginx web server
  yum:
    pkg: nginx
    state: installed
    update_cache: true
- name: Push future default virtual host configuration
  copy:
    src: files/site.conf
    dest: /etc/nginx/conf.d/
    mode: 0640
```

В данном примере мы задаем 2 задачи для группы серверов web-servers — сначала необходимо установить пакет **nginx**, затем скопировать файл site.conf с сервера на удаленную систему в каталог /etc/nginx/conf.d.

Почему Ansible лучше других подобных программ?

По сравнению с другими популярными инструментами автоматизации IT-инфраструктуры, **Ansible** не требует установки клиентских приложений на обслуживаемые сервера, что может сократить время настройки перед развертыванием инфраструктуры. Для работы **Ansible** подключается к обслуживаемым серверам по **SSH**.

Преимущества **Ansible** по сравнению с другими аналогичными решениями (здесь в первую очередь следует назвать такие продукты, как **Puppet**, **Chef** и **Salt**) заключаются в следующем:

- на управляемые узлы не нужно устанавливать никакого дополнительного программного оборудования, всё работает через **SSH** (в случае необходимости дополнительные модули можно взять из официального репозитория);
- код программы, написанный на **Python**, очень прост;
- при необходимости написание дополнительных модулей не составляет особого труда;
- язык, на котором пишутся сценарии, также предельно прост;
- низкий порог вхождения: обучиться работе с **Ansible** можно за очень короткое время;
- документация к продукту написана очень подробно и вместе с тем — просто и понятно;

▪ документация регулярно обновляется;

▪ **Ansible** работает не только в режиме **push**, но и **pull**, как это делают большинство систем управления (**Puppet**, **Chef**);

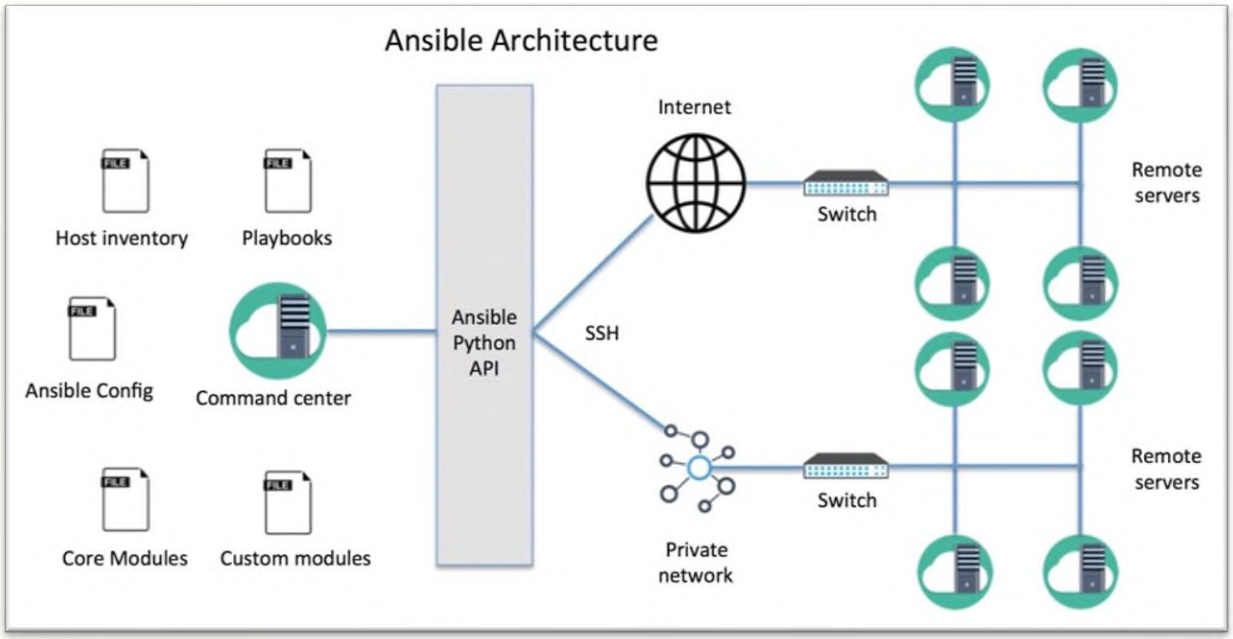
▪ имеется возможность последовательного обновления состояния узлов (**rolling update**).

Какой метод лучше Push или pull?

“Из коробки” все сценарии и команды выполняются методом **push**: когда возникает необходимость, мы запускаем сценарий, и он последовательно выполняется на удалённых серверах. Однако разработчики также предусмотрели метод **pull** и даже написали специальное приложение для установки необходимой для этого части **ansible** на удалённые хосты.

Как работает Ansible.

Основная идея **Ansible** – наличие одного или нескольких управляющих серверов, из которых вы можете отправлять команды или наборы последовательных инструкций (**playbooks**) на удаленные сервера, подключаясь к ним по **SSH**.



Файл **Host inventory** содержит информацию об обслуживаемых серверах, где команды будут исполнены. Файл конфигурации **Ansible** может быть полезен для указания настроек вашего окружения.

Наборы инструкций (**playbooks**) состоят из одной или более задач, которые описываются с помощью функциональность модуля ядра **Ansible** или сторонних модулей, которые могут потребоваться в специфических ситуациях. Сами по себе наборы инструкций — последовательные наборы команд, в

которых могут быть проверки условий: если условие не выполняется, определенные команды могут пропускаться.

Также вы можете использовать **Ansible API** для запуска скриптов. Если скрипту-обертке (**wrapper**) может потребоваться запуск **playbook**, это можно сделать через **API**. Сами **playbooks** описываются декларативно в формате **YAML**. **Ansible** поддерживает сценарии развертывания новых облачных серверов и конфигурирования их на основании ролей. Часть работы может быть проведена в локальном режиме на управляющем сервере, а остальная — на созданном сервере после его первой загрузки.

Краткий словарь терминов Ansible.

В этом этой инструкции широко используются такие термины **Ansible**:

- **Control Machine** или **Node** — ведущая система, в которой установлен **Ansible** и откуда он может подключаться к **нодам** и выполнять на них команды.
- **Хост** — в **Ansible** хост — это удаленный компьютер, которому назначены отдельные переменные, и они далее группируются вместе. У каждого хоста есть выделенное имя или уникальный **IP-адрес**, чтобы сделать его идентификацию легкой и быстрой. Им также может быть присвоен простой номер порта, если вам не нужно обращаться к ним через соединение **ssh**.
- **Нода** или **Узел** — сервер, управляемый **Ansible**.
- **Файл инвентаря** — файл, который содержит информацию о серверах, которыми управляет **Ansible**, обычно находится в /etc/ansible/hosts.
- **Playbooks** — они написаны на языке программирования **YAML** с минимальным синтаксисом и обычно используются для автоматизации задач при необходимости.
- **Role** — коллекция **плейбуков** и других файлов, которые имеют отношение к цели. Например, к установке web-сервера.
- **Play** — полный набор инструкций **Ansible**. В **play** может быть несколько **плейбуков** и **ролей**, включенных в один **плейбук**, который служит точкой входа.
- **Задача** — каждая инструкция, определенная в **книге игр**, называется **задачей**, которая будет выполняться в дальнейшем для выполнения действия.
- **Факты** — они выводятся из удаленных узлов автоматически при выполнении модулей на удаленных узлах.

- **Группа** — это комбинация хостов, которые назначены пулу, и переменные также могут совместно использоваться.
- **Инвентаризация** — инвентаризация является важным компонентом **ANSI** удаленного механизма, который описывает хосты, группы и так далее. С помощью **IP-адреса** или номера порта и так далее. Таким образом, вы можете определить все хосты в одном файле для быстрого доступа.
- **API** — это транспортная среда для различных облачных сервисов, как частных, так и общедоступных.
- **Модули** — с помощью **playbook** модули могут быть выполнены на удаленных узлах напрямую. Кроме того, его можно использовать для управления службами, ресурсами, пакетами, файлами или командами и так далее. Модули являются основными компонентами, которые помогают устанавливать пакеты, позволяют **API-интерфейсам** взаимодействовать друг с другом и планировать действия для системных файлов. В **Ansible** есть множество модулей, которые запрограммированы для автоматизации практически всего внутри инструмента.
- **Плагины** — это специальные части кода, которые помогают быстро писать код. Плагины автоматизируют задачи разработки и помогают максимально ускорить работу по развертыванию. **Ansible** оснащен различными удобными плагинами, которые можно использовать при необходимости, чтобы упростить вам задачу.
- **Оркестровка** — это общий термин, который часто используется в техническом мире. Почему это важно и в **Ansible**? Для разных программных продуктов значение оркестровки может быть различным. **Ansible** использует его в качестве дирижера для управления оркестром.

Установка.

Обновим операционную систему:

```
# yum -y update && yum -y upgrade
```

По умолчанию **Ansible** нет в репозитории **CentOS 7** — устанавливаем репозиторий **EPEL**:

```
# yum -y install epel-release
```

После устанавливаем сам сервер управления:

```
# yum -y install ansible
```

Где система автоматически обновит список пакетов с учетом нового репозитория и начнет установку **Ansible**.

Проверка версии установленной версии **Ansible**:

```
# rpm -qa | grep ansible
```

Ответ:

```
[root@cos7serv ~]# rpm -qa | grep ansible
ansible-2.9.18-1.el7.noarch
[root@cos7serv ~]#
```

или

```
# ansible --version
```

Ответ:

```
[root@cos7serv ~]# ansible --version
ansible 2.9.18
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Nov 16 2020, 22:23:17) [GCC 4.8.5 20150623 (Red Hat 4.8.5-44)]
[root@cos7serv ~]#
```

Настройка Ansible.

Файл конфигурации описывается в *INI–формате*. Вы можете переопределить часть или всю конфигурацию в параметрах **playbook** или переменных окружения.

При исполнении команд **Ansible** проверяет наличие файла конфигурации в следующих расположениях:

- проверяется переменная окружения ANSIBLE_CONFIG, которая может указывать на файл конфигурации;
- ./ansible.cfg – в текущей директории;
- ~/.ansible.cfg — в домашней директории;
- /etc/ansible/ansible.cfg — в каталоге, сгенерированном при установке **Ansible** через менеджера пакетов.

Настройка через переменные окружения.

Большинство параметров конфигурации можно установить через переменные окружения, используя префикс ANSIBLE_ перед названием параметра конфигурации (большими буквами).

Например:

```
export ANSIBLE_SUDO_USER=root
```

После этого переменная ANSIBLE_SUDO_USER может быть использована в **playbook**.

Параметры конфигурации.

Параметров конфигурации **Ansible** множество.

Давайте рассмотрим некоторые из них:

- hostfile: Параметр указывает на путь к inventory file, в котором содержится список адресов хостов, к которым **Ansible** может подключиться.
- Например: hostfile = /etc/ansible/hosts
- library: Путь к директории, где хранятся модули **Ansible**.
- Например: library = /usr/share/ansible
- forks: Количество процессов, которые может породить **Ansible**. По умолчанию установлено 5 процессов.
- Например: forks = 5
- sudo_user: Пользователь по умолчанию, от которого **Ansible** запускает команды на удаленных серверах.
- Например: sudo_user = root
- remote_port: Порт для соединения по **SSH** (по умолчанию 22).
- Например: remote_port = 22
- host_key_checking: Параметр позволяет отключить проверку **SSH-ключа** на хосте. По умолчанию проверка выполняется.
- Например: host_key_checking = False
- timeout: Значение таймаута попытки подключения по **SSH**.
- Например: timeout = 60
- log_path: Путь для хранения файлов логов. По умолчанию **Ansible** не хранит их совсем, но указав этот параметр можно активировать запись логов.
- Например: log_path = /var/log/ansible.log

Файл конфигурации ansible.cfg.

По умолчанию список **хостов**/**групп**, к которым применяются команды содержится в файле /etc/ansible/hosts:

```
# grep -E '^#inventory' /etc/ansible/ansible.cfg
```

Ответ:

```
[root@cos7serv ~]# grep -E '^#inventory' /etc/ansible/ansible.cfg
#inventory      = /etc/ansible/hosts
#inventory_plugins  = /usr/share/ansible/plugins/inventory
#inventory_ignore_extensions = ~, .orig, .bak, .ini, .cfg, .retry, .pyc, .pyo
[root@cos7serv ~]#
```

При необходимости его можно переопределить с помощью опции

```
--inventory-file(-i)
```

Пример создания кастомного файла конфигурации.

Подключитесь по **SSH** к созданному управляющему серверу с установленным **Ansible**.

Создайте директорию для экспериментов ‘ansible‘ в своём домашнем каталоге и перейдите в него:

mkdir ~/ansible# cd ~/ansible

Также создайте каталог для хранения модулей **Ansible** и каталог для хранения логов:

```
# mkdir ~/ansible/modules# mkdir ~/ansible/logs
```

Создайте файл ansible.cfg со следующим содержимым:

```
[defaults]
hostfile = ~/ansible/inventory
sudo_user = root
log_path = ~/ansible/logs/ansible.log
```

Указываем обслуживаемые сервера в **host inventory**.

Для экспериментов можно упомянуть, к примеру пару серверов, которые и можно настраивать.

Нужно сообщить **Ansible** их адреса и сгруппировать их.

Для этого создайте файл inventory в директории ~/ansible/inventory со следующим содержимым:

```
[experiments]
ip_первой_машины
ip_второй_машины
```

Файл hosts.

Работа с **Ansible** начинается с настройки его центрального файла списка хостов — так и называется файл **hosts**.

По умолчанию расположение файла — /etc/ansible/hosts, но оно может также быть задано параметром окружения $ANSIBLE_HOSTS или параметром -i при запуске ansible и ansible-playbook.

Содержимое этого файла может выглядеть, например, так, в квадратных скобках указаны имена групп управляемых узлов, ниже перечисляются входящие в эти группы серверы:

```
[dbservers]
one.example.com
two.example.com
```

```
three.example.com

[dnsservers]
rs1.example.com ansible_ssh_port=5555 ansible_ssh_host=192.168.1.50
rs2.example.com
ns[01:50].example.com
```

Помимо списка управляемых узлов, в файле **hosts** могут быть указаны и другие сведения, необходимые для работы: номера портов для подключения по **SSH**, способ подключения, пароль для подключения по **SSH**, имя пользователя, объединения групп и тому подобно. В некоторых случаях — в частности, при работе с большими и сложными конфигурациями, — различные параметры можно выносить в отдельные файлы и каталоги.

Информация об узлах — Facts.

Перед внесением изменений **Ansible** подключается к управляемым узлам и собирает информацию о них: о сетевых интерфейсах и их состоянии, об установленной операционной системе и тому подобное. Он может делать это как с помощью собственного модуля, так и с помощью инструментов **ohai** и **facter**, если они установлены (такая возможность специально предусмотрена для пользователей, уже имеющих опыт работы с системами удаленного управления конфигурациями: **ohai** и **facter** являются библиотеками фактов для **Chef** и **Puppet).**

Переменные.

Во время работы, как правило, требуется не только установить какое-либо приложение, но и настроить его в соответствии с определенными параметрами на основании принадлежности к группе серверов или индивидуально (например, **IP-адрес BGP-соседа** и номер его **AS** или параметры для базы данных).

Как уже было сказано, загромождать файл hosts будет не очень красиво, поэтому разработчики **Ansible** пошли следующим путём:

- файлы с переменными групп хранятся в каталоге ../group_vars/имя_группы;
- файлы с переменными хостов в каталоге ../hosts_vars/имя_хоста;
- файлы с переменными роли (о них речь пойдет ниже) в директории ../имя_роли/vars/имя_задачи.yml.

Помимо пользовательских переменных можно и даже нужно использовать факты, собранные **Ansible** перед выполнением сценариев и отдельных задач.

Модули Ansible.

В состав **Ansible** входит огромное количество модулей для развёртывания, контроля и управления различными компонентами, которые можно условно разделить на следующие группы (в скобках приведены названия некоторых продуктов и сервисов):

- облачные ресурсы и виртуализация (**Openstack**, **libvirt**);
- базы данных (**MySQL**, **Postgresql**, **Redis**, **Riak**);
- файлы (шаблонизация, регулярные выражения, права доступа);
- мониторинг (**Nagios**, **monit**);
- оповещения о ходе выполнения сценария (**Jabber**, **Irc**, **почта**, **MQTT**, **Hipchat**);
- сеть и сетевая инфраструктура (**Openstack**, **Arista**);
- управление пакетами (**apt**, **yum**, **rhn-channel**, **npm**, **pacman**, **pip**, **gem**);
- система (**LVM**, **Selinux**, **ZFS**, **cron**, файловые системы, сервисы, модули ядра);
- работа с различными утилитами (**git**, **hg**).

О том, с чем умеет работать **Ansible**, можно прочитать в официальной документации. Список действительно впечатляет.

Начальная настройка и тестовый запуск Ansible.

Установим файловый менеджер и текстовый редактор в одном лице — **Midnight Commander**:

```
# yum -y install mc
```

Создадим резервную копию оригинального файла конфигурации:

```
# cp /etc/ansible/hosts /etc/ansible/hosts.original
```

Откроем на редактирование файл с серверами, которыми хотим управлять:

```
# mcedit /etc/ansible/hosts
```

и приведем его к следующему виду:

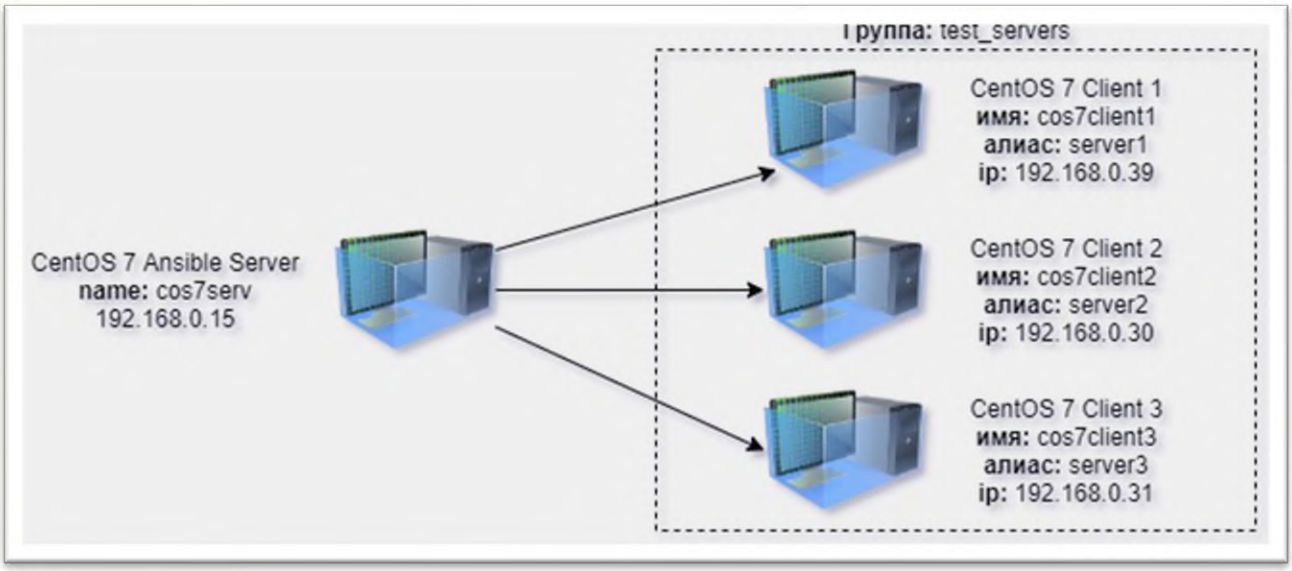
```
[test_servers]
server1 ansible_ssh_host=192.168.0.39 ansible_ssh_user=root
server2 ansible_ssh_host=192.168.0.30 ansible_ssh_user=root
```

```
server3 ansible_ssh_host=192.168.0.31 ansible_ssh_user=root
```

где

- test_servers — это группа серверов, в которую добавлены три сервера с **IP-адресами** 192.168.0.30, 192.168.0.31 и 192.168.0.39;
- server 1–3 — это индивидуальные алиасы каждого из серверов группыtest_servers в списке инвентаря;
- ansible_ssh_host — это специальная переменная, которая содержит **IP-адрес** узла, к которому будет создаваться соединение;
- ansible_ssh_user — это еще одна специальная переменная, которая говорит **Ansible**‘у подключаться под указанным аккаунтом, то есть пользователем. По умолчанию **Ansible** использует ваш текущий аккаунт пользователя, или другое значение по умолчанию, указанное в ~/.ansible.cfg (**remote_user**).

Все сервера, в данном примере, имеют одинаковую операционную систему *CentOS Linux release 7.9.2009 (Core)*, одинаковых пользователей **root** и одинаковые пароли на учетных записях **root**.



Открываем конфигурационный файл **Ansible** (выше по тексту был способ изменить место хранения файла конфигурации):

```
# mcedit /etc/ansible/ansible.cfg
```

Снимаем комментарий с опции host_key_checking, приведя ее к виду:

host_key_checking = False

Данная настройка позволит нашему серверу управления автоматически принимать **ssh fingerprint**, избавляя нас от необходимости постоянно вводить yes, когда мы впервые конфигурируем новый сервер.

Теперь выполним проверку доступности добавленных серверов:

```
# ansible -m ping test_servers -u root -kK
```

Данная команда проверит доступность по сети двух серверов из группы test_servers от учетной записи **root**.

Будет запрошен пароль от учетной записи (в нашем случае, **root**).

После будет запрошен пароль суперпользователя на серверах.

```
[root@cos7serv ~]# ansible -m ping test_servers -u root -kK
SSH password:
BECOME password[defaults to SSH password]:
```

2. Работа с Playbook

Одним из основных понятий в Ansible является плейбук. Это просто yaml-файл, в котором мы указываем, какие задачи и на каких серверах будут выполняться. Пример такого файла *playbook.yml*:

```
# На какой группе серверов
- hosts: webservers

  tasks:
    - name: install redis server
      # apt-get update && apt-get install redis-server
      ansible.builtin.apt: # имя модуля Ansible
        name: redis-server
        state: present
        update_cache: yes

    - name: remove redis server
      # apt-get remove redis-server
      ansible.builtin.apt:
        name: redis-server
        state: absent
```

В этом плейбуке на группе хостов *webservers* выполняются две задачи (таски). Первая — это установка *redis-server*, и вторая — его удаление. Структура любой задачи такая:

- имя задачи — необязательный параметр, в котором описывается задача. Нужна только для вывода во время выполнения плейбука.
- модуль и его параметры – определяет команду, которая будет выполнена на указанной группе серверов. В примере выше это модуль *apt*, который запускает пакетный менеджер *apt* входящий в стандартную поставку Ubuntu. С его помощью устанавливаются и удаляются программы.

Имя модуля в задачах задается с префиксом *ansible.builtin*. Ansible позволяет создавать свои модули, у которых имена могут совпадать со встроенными модулями. Поэтому разработчики ввели пространства имен, позволяющие избежать неоднозначностей

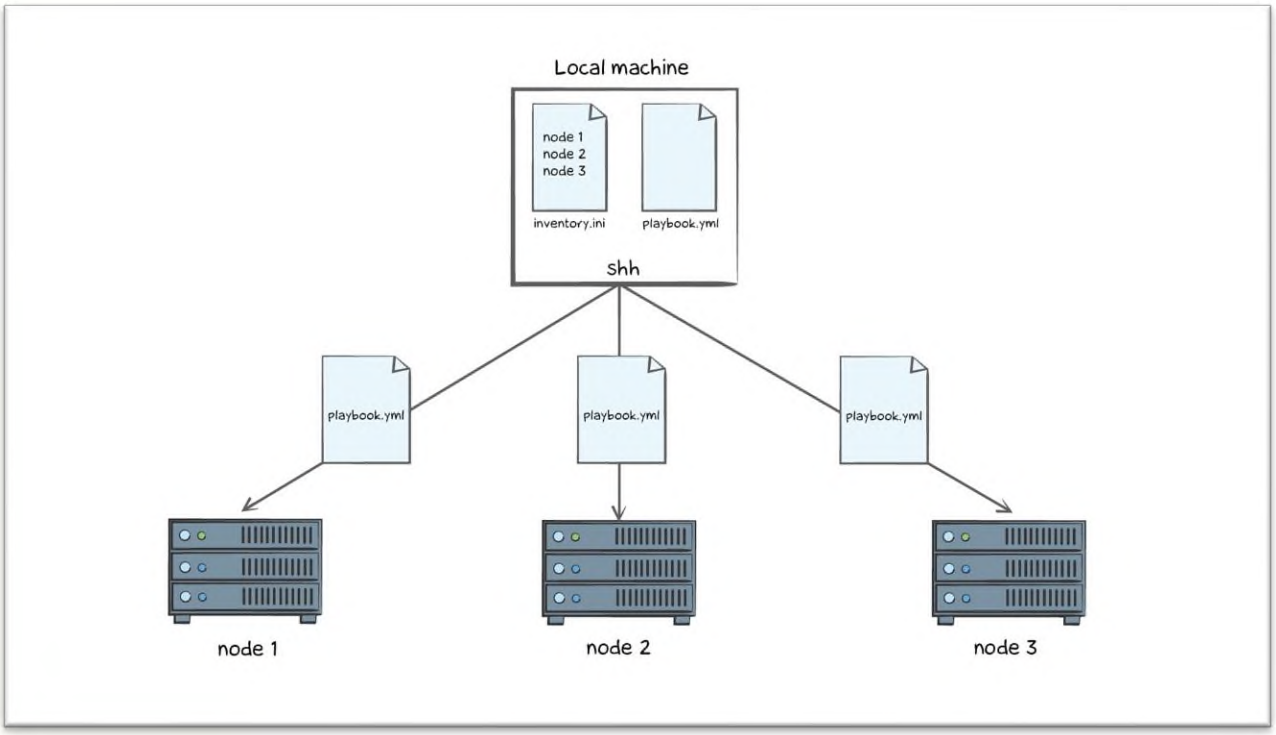
Каждый модуль принимает параметры. В первой задаче три параметра:

- name: redis-server — имя пакета.

• state: present — состояние, в которое требуется привести модуль. Ansible убедится, что этот пакет есть, либо доустановит его.

• update_cache: yes — выполняет команду apt-get update для того, чтобы обновить информацию о пакетах в индексе. Во второй задаче обновлять индекс не требуется. Достаточно указать состояние state: absent, чтобы Ansible просто удалил redis-сервер.

В плейбуке может быть сколько угодно задач, но каждая задача работает только с одним модулем. Будьте внимательны.



Для запуска плейбука используется команда ansible-playbook, которой мы передаем путь к файлу *playbook.yml* и указываем inventory-файл:

 # В отличие от ad-hoc режима, группа хостов указывается внутри плейбука

ansible-playbook playbook.yml -i inventory.ini

При запуске этой команды первая же задача упадёт с ошибкой *Failed to lock apt for exclusive operation*. Из этой ошибки следует, что команда выполняется без прав администратора и не может выполнить установку программ. Чтобы это изменить, можно заходить на сервер под root, что очень часто неприемлемо, и так лучше не делать. Поэтому в Ansible существует механизм переключения пользователя уже внутри. Он называется *become*. В простейшем случае достаточно прописать become: yes в нужные задачи:

- hosts: webservers
  tasks:

```
- name: install redis server
  ansible.builtin.apt:
    name: redis-server
    state: present
    update_cache: yes
  become: yes # <---

- name: remove redis server
  ansible.builtin.apt:
    name: redis-server
    state: absent
  become: yes # <---
```

По умолчанию become использует sudo и переключает в root, поэтому у вашего пользователя должны быть необходимые права. Если понадобится другой пользователь, то его можно указать в параметре become_user. Естественно, необходимо чтобы у вашего пользователя было sudo. Теперь задача отрабатывает. Обновление индекса и установка занимает некоторое время:

```
TASK [install redis server] **********************************
changed: [157.230.82.133]

TASK [remove redis server] ***********************************
changed: [157.230.82.133]
```

3. Работа с Roles

Roles позволяют автоматически загружать связанные переменные, файлы, задачи, обработчики и другие артефакты Ansible на основе известной файловой структуры. После того, как вы сгруппируете свой контент по ролям, вы можете легко использовать его повторно и делиться им с другими пользователями.

Role ansible — это концепция в Ansible, которая имеет дело с идеями, а не с событиями. По сути, роль — это уровень абстракции, используемый для упрощения написания файлов playbook. Role предоставляет основу для повторно используемых компонентов, таких как переменные, модули, задачи и факты, которые можно загрузить в файл Playbook.

Структура каталога ролей

Role Ansible имеет определенную структуру каталогов с восемью основными стандартными каталогами. Вы должны включить хотя бы один из этих каталогов в каждую роль. Вы можете опустить любые каталоги, которые роль не использует. Например:

```
# playbooks
site.yml
webservers.yml
fooservers.yml
roles/
   common/
      tasks/
      handlers/
      library/
      files/
      templates/
      vars/
      defaults/
      meta/
   webservers/
      tasks/
      defaults/
      meta/
```

По умолчанию Ansible будет искать в каждом каталоге `roles` файл `main.yml` для соответствующего содержимого (также `main.yaml` и `main`):

- tasks/main.yml- основной список задач, которые выполняет роль.
- handlers/main.yml- обработчики, которые могут использоваться внутри или вне этой роли.
- library/my_module.py- модули, которые можно использовать в рамках этой роли.
- defaults/main.yml- переменные по умолчанию для роли. Эти переменные имеют самый низкий приоритет из всех доступных переменных и могут быть легко переопределены любой другой переменной, включая переменные инвентаризации.
- vars/main.yml- другие переменные для роли
- files/main.yml- файлы, которые развертывает роль.
- templates/main.yml- шаблоны, которые развертывает роль.
- meta/main.yml- метаданные для роли, включая ролевые зависимости.

Вы можете добавить другие файлы YAML в некоторые каталоги. Например, вы можете поместить специфичные для платформы задачи в отдельные файлы и ссылаться на них в `tasks/main.yml` файле:

```
# roles/example/tasks/main.yml
- name: Install the correct web server for RHEL
  import_tasks: redhat.yml
  when: ansible_facts['os_family']|lower == 'redhat'

- name: Install the correct web server for Debian
  import_tasks: debian.yml
  when: ansible_facts['os_family']|lower == 'debian'

# roles/example/tasks/redhat.yml
- name: Install web server
  ansible.builtin.yum:
    name: "httpd"
    state: present
```

```
# roles/example/tasks/debian.yml
- name: Install web server
  ansible.builtin.apt:
    name: "apache2"
    state: present
```

Roles также могут включать модули и другие типы подключаемых модулей в каталоге с именем library.

Хранение и поиск ролей

По умолчанию Ansible ищет роли в следующих местах:

- в коллекциях, если вы их используете
- в каталоге с именем roles/ относительно файла playbook
- в настроенном role_path . Путь поиска по умолчанию — ~/.ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles.
- в каталоге, где находится файл playbook

Если вы храните свои роли в другом месте, установите параметр конфигурации roles_path, чтобы Ansible мог найти ваши роли. Сохранение общих ролей в одном месте упрощает их использование в нескольких сборниках сценариев.

Кроме того, вы можете вызвать роль с полным путем:

```
---
- hosts: webservers
  roles:
    - role: '/path/to/my/roles/common'
```

Использование ролей

Вы можете использовать роли тремя способами:

- на уровне play с опцией roles: Это классический способ использования ролей в play.
- на уровне task с помощью include_role: вы можете динамически повторно использовать роли в любом месте секции tasks play, используя include_role.

- на уровне задач с помощью import_role: вы можете повторно использовать roles статически в любом месте секции tasks play, используя import_role.

Использование ролей на уровне play

Классический способ использования ролей — это опция `roles` для данной play:

```
---
- hosts: webservers
  roles:
    - common
    - webservers
```

Когда вы используете `roles` опцию на уровне play, для каждой roles «x»:

- Если файл roles/x/tasks/main.yml существует, Ansible добавляет задачи из этого файла в play.
- Если roles/x/handlers/main.yml существует, Ansible добавляет обработчики из этого файла в play.
- Если файл roles/x/vars/main.yml существует, Ansible добавляет в play переменные из этого файла.
- Если файл roles/x/defaults/main.yml существует, Ansible добавляет переменные из этого файла в play.
- Если файл roles/x/meta/main.yml существует, Ansible добавит любые зависимости ролей в этом файле в список ролей.
- Любая копия, сценарий, шаблон или включаемые задачи (в roles) могут ссылаться на файлы в ролях/x/{файлы, шаблоны, задачи}/ (каталог зависит от задачи) без необходимости относительного или абсолютного пути к ним.

Когда вы используете этот параметр `roles` на уровне play, Ansible рассматривает roles как статический импорт и обрабатывает их во время синтаксического анализа playbook. Ansible выполняет ваш плейбук в следующем порядке:

- Любой определенный `pre_tasks` в play.
- Любые обработчики, запущенные `pre_tasks`.

- Каждая роль, указанная в списке roles:, в указанном порядке. Любые зависимости ролей, определенные в роли, meta/main.ymlзапускаются первыми, с учетом фильтрации тегов и условий.
- Любой определенный tasks в play.
- Любые обработчики, запускаемые ролями или задачами.
- Любой определенный post_tasks в play.
- Любые обработчики, запускаемые post_tasks.

Вы можете передать другие ключевые слова в опцию roles:

```
---
- hosts: webservers
  roles:
    - common
    - role: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
      tags: typeA
    - role: foo_app_instance
      vars:
        dir: '/opt/b'
        app_port: 5001
      tags: typeB
```

Когда вы добавляете тег к опции role, Ansible применяет тег ко ВСЕМ задачам в роли.

При использовании vars: в разделе roles: плейбука переменные добавляются к play переменным, что делает их доступными для всех задач в play до и после роли. Это поведение можно изменить с помощью DEFAULT_PRIVATE_ROLE_VARS .

Включение ролей: динамическое повторное использование

Вы можете динамически повторно использовать роли в любом месте tasks plays, используя файлы include_role. В то время как роли, добавленные в раздел roles, выполняются перед любыми другими задачами в playbook,

включенные роли выполняются в том порядке, в котором они определены. Если перед задачей include_role есть другие задачи, другие задачи будут выполняться первыми.

Чтобы включить роль:

```
---
- hosts: webservers
  tasks:
    - name: Print a message
      ansible.builtin.debug:
        msg: "this task runs before the example role"

    - name: Include the example role
      include_role:
        name: example

    - name: Print a message
      ansible.builtin.debug:
        msg: "this task runs after the example role"
```

Вы можете передавать другие ключевые слова, включая переменные и теги, при включении ролей:

```
---
- hosts: webservers
  tasks:
    - name: Include the foo_app_instance role
      include_role:
        name: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
      tags: typeA
  ...
```

Когда вы добавляете тег к задаче include_role, Ansible применяет тег *только* к самому включению. Это означает, что вы можете перейти к запуску –tags

только выбранных задач из роли, если сами эти задачи имеют тот же тег, что и оператор включения.

Можно условно включить роль:

```
---
- hosts: webservers
  tasks:
    - name: Include the some_role role
      include_role:
        name: some_role
      when: "ansible_facts['os_family'] == 'RedHat'"
```

Импорт ролей: статическое повторное использование

Вы можете повторно использовать роли статически в любом месте tasks plays, используя файлы `import_role`. Поведение такое же, как при использовании ключевого слова `roles`. Например:

```
---
- hosts: webservers
  tasks:
    - name: Print a message
      ansible.builtin.debug:
        msg: "before we run our role"

    - name: Import the example role
      import_role:
        name: example

    - name: Print a message
      ansible.builtin.debug:
        msg: "after we ran our role"
```

Вы можете передавать другие ключевые слова, включая переменные и теги, при импорте ролей:

```
---
- hosts: webservers
  tasks:
```

```
  - name: Import the foo_app_instance role
    import_role:
      name: foo_app_instance
    vars:
      dir: '/opt/a'
      app_port: 5000
...
```

Когда вы добавляете тег к оператору import_role, Ansible применяет тег ко *всем* задачам в роли.

Проверка аргумента роли

Начиная с версии 2.11, вы можете включить проверку аргумента роли на основе спецификации аргумента. Эта спецификация определяется в meta/argument_specs.yml файле (или с .yaml расширением файла). Когда эта спецификация аргумента определена, в начале выполнения роли вставляется новая задача, которая проверяет параметры, предоставленные для роли, на соответствие спецификации. Если параметры не проходят проверку, роль не будет выполняться.

Формат спецификации

Спецификация аргумента роли должна быть определена в блоке argument_specs верхнего уровня в файле meta/argument_specs.yml роли. Все поля в нижнем регистре.

entry-point-name:	• Имя точки входа роли. • Это должно быть main в случае неуказанной точки входа. • Это будет базовое имя исполняемого файла задач без расширения файла .yml или ..yaml	
	short_description:	• Краткое однострочное описание точки входа. • short_description Отображается как ansible-doc -t role -l
	description:	• Более длинное описание, которое может состоять из нескольких строк.
	author:	• Имя авторов точки входа. • Используйте многострочный список, если авторов несколько.

	options:	• Опции часто называют «параметрами» или «аргументами». В этом разделе определяются эти параметры. • Для каждого параметра роли (аргумента) вы можете включить:	
		option-name:	• Имя опции/аргумента.
		description:	• Подробное объяснение того, что делает эта опция. Следует писать полными предложениями.
		type:	• Тип данных опции. Значение по умолчанию str. • Если параметр имеет тип list, elements его следует указать.
		required:	• Нужен только если true. • Если отсутствует, опция не требуется.
		default:	• Если required ложно/отсутствует, default может быть указано (предполагается «нулевым», если отсутствует). • Убедитесь, что значение по умолчанию в документах соответствует значению по умолчанию в коде. Фактическое значение по умолчанию для

			переменной роли всегда будет исходить из defaults/main.yml. • Поле по умолчанию не должно быть указано как часть описания, если только оно не требует дополнительной информации или условий. • Если опция является логическим значением, вы можете использовать любое из логических значений, распознаваемых Ansible: (например, true/false или yes/no). Выберите тот, который лучше читается в контексте варианта.
		choices:	• Список значений опций. • Должен отсутствовать, если пуст.
		elements:	• Указывает тип данных для элементов списка, если тип равен list.
		options:	• Если эта опция принимает словарь или список словарей, вы можете определить структуру здесь.

Пример спецификации

```
# roles/myapp/meta/argument_specs.yml
```

```
---
argument_specs:
  # roles/myapp/tasks/main.yml entry point
  main:
    short_description: The main entry point for the myapp role.
    options:
      myapp_int:
        type: "int"
        required: false
        default: 42
        description: "The integer value, defaulting to 42."

      myapp_str:
        type: "str"
        required: true
        description: "The string value"

  # roles/maypp/tasks/alternate.yml entry point
  alternate:
    short_description: The alternate entry point for the myapp role.
    options:
      myapp_int:
        type: "int"
        required: false
        default: 1024
        description: "The integer value, defaulting to 1024."
```

Запуск роли несколько раз в одном плейбуке

Ansible выполняет каждую роль только один раз, даже если вы определяете ее несколько раз, если только параметры, определенные для роли, не различаются для каждого определения. Например, Ansible запускает роль foo только один раз в такой play:

```
---
- hosts: webservers
  roles:
    - foo
```

```
  - bar
  - foo
```

У вас есть два варианта заставить Ansible запускать роль более одного раза.

Передача разных параметров

Если вы передаете разные параметры в каждом определении роли, Ansible запускает роль более одного раза. Предоставление разных значений переменных — это не то же самое, что передача разных параметров роли. Вы должны использовать ключевое слово `roles` для этого поведения, так как `import_role` и `include_role` не принимают параметры роли.

В этой play эта роль выполняется foo дважды:

```
---
- hosts: webservers
  roles:
    - { role: foo, message: "first" }
    - { role: foo, message: "second" }
Этот синтаксис также запускает foороль дважды;
---
- hosts: webservers
  roles:
    - role: foo
      message: "first"
    - role: foo
      message: "second"
```

В этих примерах Ansible запускается дважды foo, потому что каждое определение роли имеет разные параметры.

С использованием allow_duplicates: true

Добавьте в файл для `role:allow_duplicates: truemeta/main.yml`

```
# playbook.yml
---
- hosts: webservers
  roles:
    - foo
    - foo
```

```
# roles/foo/meta/main.yml
---
allow_duplicates: true
```

В этом примере Ansible запускается foo дважды, потому что мы явно разрешили ему это делать.

Использование role dependencies

Зависимости ролей позволяют автоматически подключать другие роли при использовании роли. Ansible не выполняет зависимости ролей, когда вы включаете или импортируете роль. Вы должны использовать ключевое слово `roles`, если хотите, чтобы Ansible выполнял зависимости ролей.

Role dependencies — это предпосылки, а не настоящие зависимости. Roles не имеют отношений родитель/потомок. Ansible загружает все перечисленные роли, начала `dependenciesc` запускает роли, перечисленные ниже, а затем запускает роль, в которой они перечислены. Объект воспроизведения является родителем всех ролей, включая роли, вызываемые списком `dependencies`.

Зависимости ролей хранятся в meta/main.yml файле в каталоге ролей. Этот файл должен содержать список ролей и параметров для вставки перед указанной ролью. Например:

```
# roles/myapp/meta/main.yml
---
dependencies:
  - role: common
    vars:
      some_parameter: 3
  - role: apache
    vars:
      apache_port: 80
  - role: postgres
    vars:
      dbname: blarg
      other_parameter: 12
```

Ansible всегда выполняет роли, перечисленные в dependencies до роли, которая их перечисляет. Ansible выполняет этот шаблон рекурсивно, когда вы используете ключевое слово roles. Например, если вы указываете роль foo в разделе roles:, роль foo перечисляет роль bar в dependencies своем файле meta/main.yml, а роль bar перечисляет роль baz в dependencies своем файле meta/main.yml, Ansible выполняет baz, затем bar, затем foo.

Запуск ролевых зависимостей несколько раз в одной play

Ansible обрабатывает зависимости повторяющихся ролей, как дублирующие роли, перечисленные в разделе roles:: Ansible выполняет зависимости ролей только один раз, даже если они определены несколько раз, за исключением случаев, когда параметры, теги или оператор when, определенные для роли, различаются для каждого определения. Если две роли в play указывают третью роль как зависимость, Ansible запускает зависимость этой роли только один раз, если только вы не передаете разные параметры, теги, предложение when или не используете роль, которую хотите запускать несколько раз.

Встраивание модулей и плагинов в роли

Например, если вы пишете модуль, который помогает настроить внутреннее программное обеспечение вашей компании, и вы хотите, чтобы другие люди в вашей организации использовали этот модуль, но вы не хотите рассказывать всем, как настроить путь к своей библиотеке Ansible, вы можете включить модуль в вашей роли internal_config.

Чтобы добавить модуль или плагин к роли: Наряду со структурой «задачи» и «обработчики» роли добавьте каталог с именем «библиотека», а затем включите модуль непосредственно в каталог «библиотека».

Предполагая, что у вас было это:

```
roles/
  my_custom_modules/
    library/
      module1
      module2
```

Модуль можно будет использовать в самой роли, а также в любых ролях, вызываемых *после* этой роли, следующим образом:

```
---
- hosts: webservers
  roles:
    - my_custom_modules
    - some_other_role_using_my_custom_modules
    - yet_another_role_using_my_custom_modules
```

При необходимости вы также можете встроить модуль в роль, чтобы изменить модуль в базовом дистрибутиве Ansible. Например, вы можете использовать разрабатываемую версию определенного модуля до его выпуска в рабочих выпусках, скопировав модуль и встроив копию в роль. Используйте этот подход с осторожностью, так как сигнатуры API могут измениться в основных компонентах, а работа этого обходного пути не гарантируется.

Тот же механизм можно использовать для встраивания и распространения подключаемых модулей в роли с использованием той же схемы. Например, для плагина фильтра:

```
roles/
   my_custom_filter/
      filter_plugins
         filter1
         filter2
```

Затем эти фильтры можно использовать в шаблоне Jinja в любой роли, называемой после «my_custom_filter».

Разделение ролей: Ansible Galaxy

Ansible Galaxy — это бесплатный сайт для поиска, загрузки, оценки и обзора всех видов ролей Ansible, разработанных сообществом, который может стать отличным способом дать толчок вашим проектам автоматизации.

Клиент ansible-galaxy включен в Ansible. Клиент Galaxy позволяет загружать роли из Ansible Galaxy, а также предоставляет отличную стандартную среду для создания собственных ролей.

Прочитайте страницу документации Ansible Galaxy для получения дополнительной информации .

Введение в Python

Краткая историческая справка

Язык программирования Python был создан к 1991 году голландцем Гвидо ван Россумом.

Свое имя – Пайтон (или Питон) – получил от названия телесериала, а не пресмыкающегося.

После того, как Россум разработал язык, он выложил его в Интернет, где сообщество программистов присоединилось к его улучшению.

Python активно развивается и сейчас. Часто выходят новые версии. Еще недавно поддерживались две отдельные ветки языка: Python 2.x и Python 3.x. Здесь английской буквой "x" обозначается конкретный релиз. Между вторым и третьим Питоном есть небольшая разница. В настоящее время поддержка Python 2 прекращена.

Официальный сайт языка – https://www.python.org.

Основные особенности языка

Python – интерпретируемый язык программирования. Это значит, что исходный код частями преобразуется в машинный в процессе его чтения специальной программой – интерпретатором.

Python характеризуется ясным синтаксисом. Читать код на нем легче, чем на других языках программирования, так как в Питоне мало используются такие вспомогательные синтаксические элементы как скобки, точки с запятыми. С другой стороны, **правила языка заставляют программистов делать отступы** для обозначения вложенных конструкций. Понятно, что хорошо оформленный текст с малым количеством отвлекающих элементов читать и понимать легче.

Python – это полноценный во многом универсальный язык программирования, используемый в различных сферах. Основная, но не единственная, поддерживаемая им парадигма, – объектно-ориентированное программирование. Однако в данном курсе мы только упомянем об объектах, а будем изучать структурное программирование, так как оно является базой. Без знания основных типов данных, ветвлений, циклов,

функций нет смысла изучать более сложные парадигмы, так как в них все это используется.

Интерпретаторы Python распространяется свободно на основании лицензии подобной GNU General Public License.

Дзен Питона

Если интерпретатору Питона дать команду import this, то выведется так называемый "Дзен Питона", иллюстрирующий идеологию и особенности данного языка. Понимание смысла этих постулатов в приложении к программированию придет тогда, когда вы освоите язык в полной мере и приобретете опыт практического программирования.

- Beautiful is better than ugly. Красивое лучше уродливого.
- Explicit is better than implicit. Явное лучше неявного.
- Simple is better than complex. Простое лучше сложного.
- Complex is better than complicated. Сложное лучше усложнённого.
- Flat is better than nested. Плоское лучше вложенного.
- Sparse is better than dense. Разрежённое лучше плотного.
- Readability counts. Удобочитаемость важна.
- Special cases aren't special enough to break the rules. Частные случаи не настолько существенны, чтобы нарушать правила.
- Although practicality beats purity. Однако практичность важнее чистоты.
- Errors should never pass silently. Ошибки никогда не должны замалчиваться.
- Unless explicitly silenced. За исключением замалчивания, которое задано явно.
- In the face of ambiguity, refuse the temptation to guess. Перед лицом неоднозначности сопротивляйтесь искушению угадать.
- There should be one — and preferably only one — obvious way to do it. Должен существовать один — и, желательно, только один — очевидный способ сделать это.

- Although that way may not be obvious at first unless you're Dutch. Хотя он может быть с первого взгляда не очевиден, если ты не голландец.
- Now is better than never. Сейчас лучше, чем никогда.
- Although never is often better than *right* now. Однако, никогда чаще лучше, чем прямо сейчас.
- If the implementation is hard to explain, it's a bad idea. Если реализацию сложно объяснить — это плохая идея.
- If the implementation is easy to explain, it may be a good idea. Если реализацию легко объяснить — это может быть хорошая идея.
- Namespaces are one honking great idea — let's do more of those! Пространства имён — прекрасная идея, давайте делать их больше!

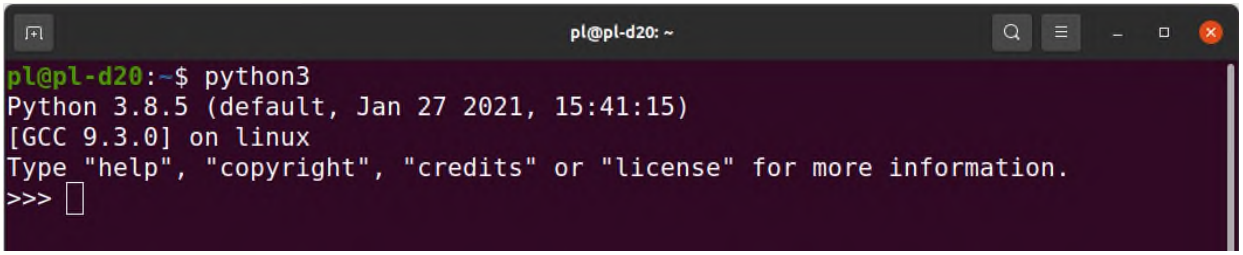
Как писать программы на Python

Интерактивный режим

Грубо говоря, интерпретатор выполняет команды построчно. Пишешь строку, нажимаешь Enter, интерпретатор выполняет ее, наблюдаешь результат.

Это удобно, когда изучаешь особенности языка или тестируешь какую-нибудь небольшую часть кода. Ведь если работать на компилируемом языке, пришлось бы сначала создать файл с кодом на исходном языке программирования, затем передать его компилятору, получить от него исполняемый файл и только потом выполнить программу и оценить результат. К счастью, даже в случае с компилируемыми языками все эти действия выполняет среда разработки, что упрощает жизнь программиста.

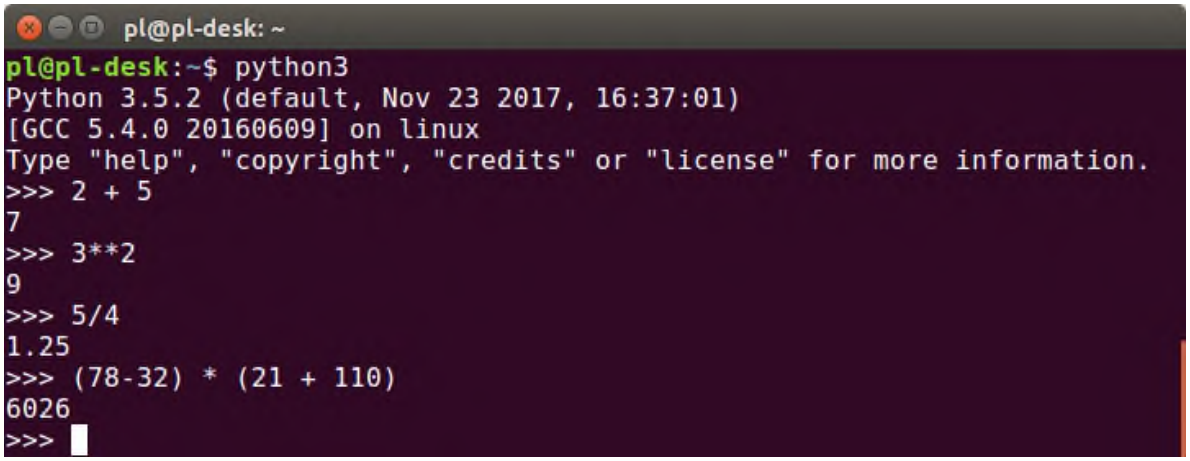
В операционных системах на базе ядра Linux можно программировать на Python в интерактивном режиме с помощью приложения «Терминал», в котором работает командная оболочка Bash. Здесь, чтобы запустить интерпретатор, надо выполнить команду python3.



В данном случае запустилась версия 3.8.5. Первое число 3 указывает на то, что это интерпретатор для языка программирования Python 3. Последняя строка с тремя угловыми скобками (>>>) – это приглашение для ввода команд.

Для операционных систем семейства Windows надо скачать интерпретатор с официального сайта языка (https://www.python.org/downloads/windows/). После установки он будет запускаться по ярлыку. Использовать командную оболочку здесь не требуется.

Возможности Python позволяют использовать его как калькулятор. Поскольку команды языка мы не изучали, это хороший способ протестировать интерактивный ввод команд.



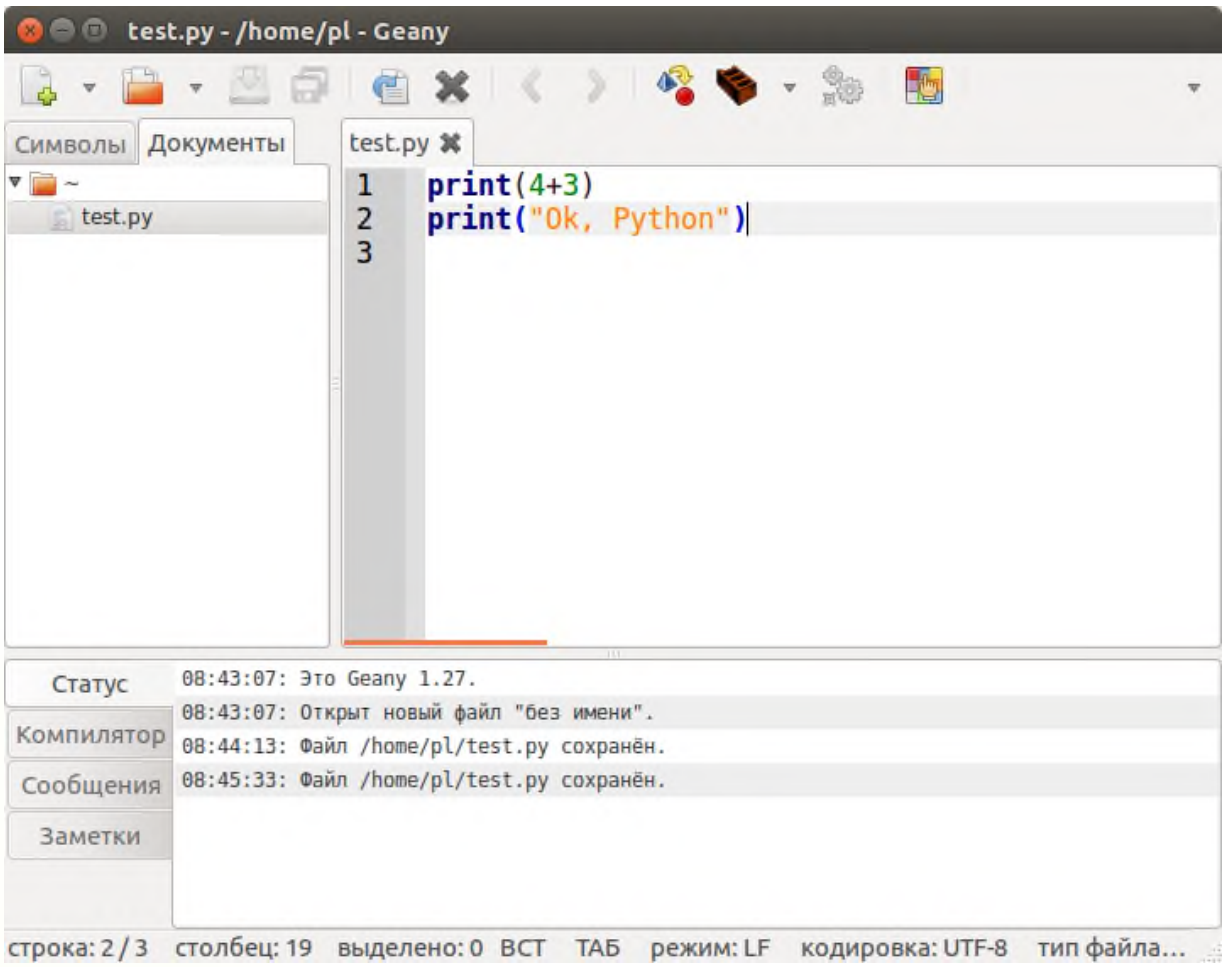
Бывает, что в процессе ввода была допущена ошибка или требуется повторить ранее используемую команду. Чтобы заново не вводить строку, в консоли можно прокручивать историю команд, используя для этого стрелки вверх и вниз на клавиатуре. В среде IDLE (в Windows) для этого используются сочетания клавиш (скорее всего Alt+N и Alt+P).

Создание скриптов

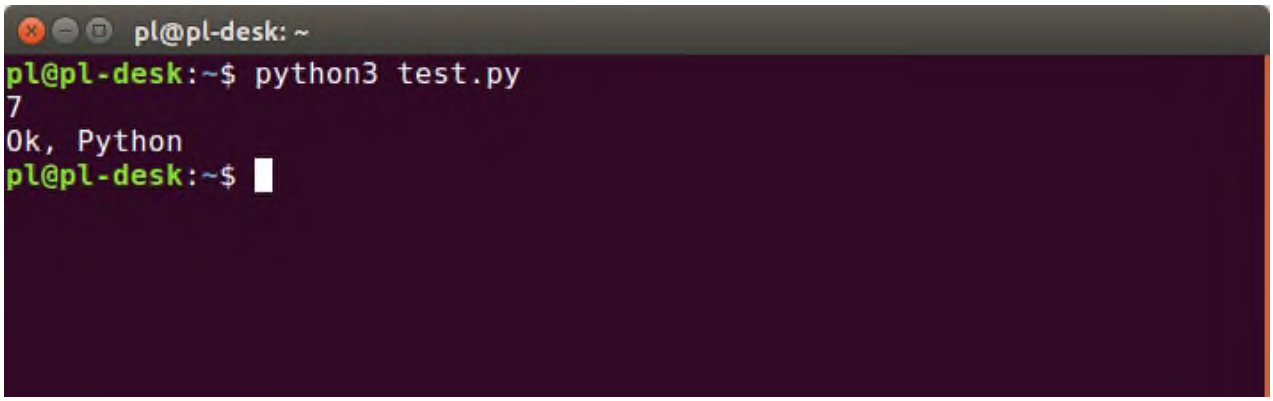
Несмотря на удобства интерактивного режима, чаще всего необходимо сохранить исходный программный код для последующего выполнения и использования. В таком случае подготавливаются файлы, которые передаются затем интерпретатору на исполнение. Файлы с кодом на Python обычно имеют расширение *.py*.

Существует множество сред разработки (IDE), в том числе созданные для программирования почти исключительно на Python. Примером такой среды

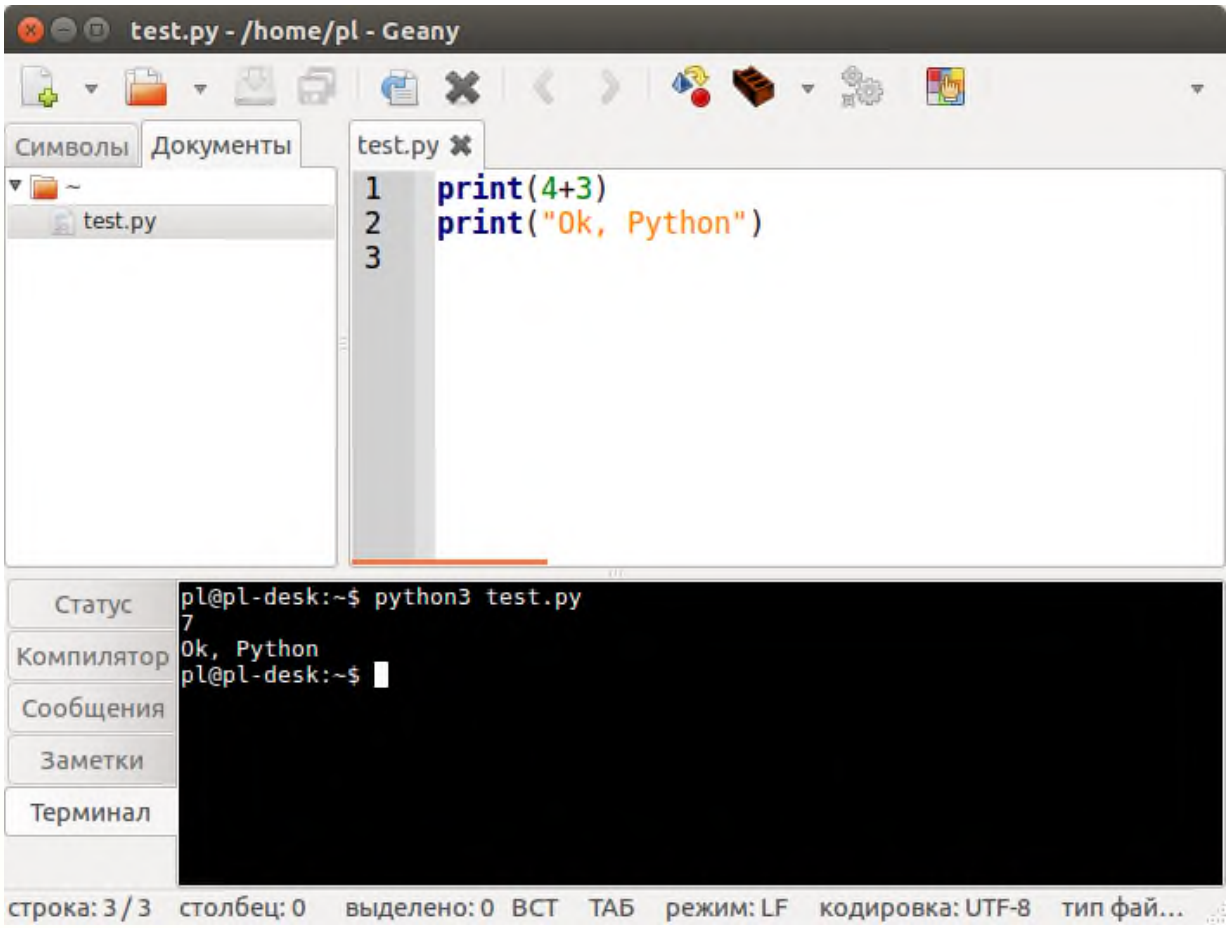
является PyCharm. Однако на первое время подойдет текстовый редактор с подсветкой синтаксиса, например, Geany.



Здесь создается и сохраняется файл с кодом. Далее его можно запустить на выполнение через терминал. При этом сначала указывается интерпретатор (в данном случае python3), потом имя файла (если файл находится в другом каталоге, то указывается с адресом, или надо перейти в этот каталог с помощью команды **cd** оболочки Bash).



При этом в Geany может быть установлен свой встроенный «терминал», что упрощает работу.



Также в Geany можно просто нажать F5, что отправит файл на исполнение (терминал откроется сам, после выполнения программы и нажатия Enter закроется). Однако при этом должен быть правильно настроен вызываемый интерпретатор (пункт меню **Сборка → Установить команды сборки**).

В Windows подготовить файлы можно в той же среде IDLE. Для этого в меню следует выбрать команду **File → New Window** (Crtl + N), откроется чистое (без приглашения >>>) новое окно. Желательно сразу сохранить файл с расширением *.py*, чтобы появилась подсветка синтаксиса. После того как код будет подготовлен, снова сохраните файл. Запуск скрипта выполняется командой **Run → Run Module** (F5). После этого в окне интерактивного режима появится результат выполнения кода.

Типы данных

Итак, мы видим, что Python умеет работать как минимум с двумя видами данных — числами и строками. Числа записываются последовательностью цифр, также перед числом может стоять знак минус, а строки записываются в одинарных кавычках. 2 и '2' — это разные объекты, первый объект — число,

а второй — строка. Операция + для целых чисел и для строк работает по-разному: для чисел это сложение, а для строк — конкатенация.

Кроме целых чисел есть и другой класс чисел: действительные (вещественные числа), представляемые в виде десятичных дробей. Они записываются с использованием десятичной точки, например, 2.0.

Определить тип объекта можно при помощи функции type:

```
>>> type(2)
<class 'int'>
>>> type('2')
<class 'str'>
>>> type(2.0)
<class 'float'>
```

Обратите внимание: type является функцией, аргументы функции указываются в скобках после ее имени.

Операции с числами

Вот список основных операций для чисел:

- A+B — сумма;
- A-B — разность;
- A*B — произведение;
- A/B — частное;
- A**B — возведение в степень.

Полезно помнить, что квадратный корень из числа x — это x**0.5, а корень степени n — это x**(1/n).

Есть также унарный вариант операции -, то есть это операция с одним аргументом. Она возвращает число, противоположное данному. Например: -A.

В выражении может встречаться много операций подряд. Как в этом случае определяется порядок действий? Например, чему будет равно 1+2*3**1+1? В данном случае ответ будет 8, так как сначала выполняется возведение в степень, затем — умножение, затем — сложение.

Более общие правила определения приоритетов операций такие:

1. Выполняются возведения в степень **справа налево**, то есть 3**3**3 это 3^{27}.
2. Выполняются унарные минусы (отрицания).
3. Выполняются умножения и деления слева направо. Операции умножения и деления имеют одинаковый приоритет.
4. Выполняются сложения и вычитания слева направо. Операции сложения и вычитания имеют одинаковый приоритет.

Операции над строками

- A+B — конкатенация;
- A*n — повторение n раз, значение n должно быть целого типа.

Ветвление

Ветвление (или условная инструкция) в Python имеет следующий синтаксис:

```
if Условие:
   Блок_инструкций_1
else:
   Блок_инструкций_2
```

Блок_инструкций_1 будет выполнен, если Условие истинно. Если Условие ложно, будет выполнен Блок_инструкций_2.

В условной инструкции может отсутствовать слово else и последующий блок. Такая инструкция называется неполным ветвлением. Например, если дано число x и мы хотим заменить его на абсолютную величину x, то это можно сделать следующим образом:

```
if x < 0:
   x = -x
print(x)
```

В этом примере переменной x будет присвоено значение -x, но только в том случае, когда x<0. А вот инструкция print(x) будет выполнена всегда, независимо от проверяемого условия.

Для выделения блока инструкций, относящихся к инструкции if или else, в языке Python используются отступы. Все инструкции, которые относятся к одному блоку, должны иметь равную величину отступа, то есть одинаковое число пробелов в начале строки. Рекомендуется использовать *отступ в 4 пробела*.

Вложенные условные инструкции

Внутри условных инструкций можно использовать любые инструкции языка Python, в том числе и условную инструкцию. Вложенное ветвление — после одной развилки в ходе исполнения программы появляется другая развилка. При этом вложенные блоки имеют больший размер отступа (например, 8 пробелов).

Примере программы, которая по данным ненулевым числам x и y определяет, в какой из четвертей координатной плоскости находится точка (x,y):

```
x = int(input())
y = int(input())
if x > 0:
    if y > 0:           # x>0, y>0
        print("Первая четверть")
    else:               # x>0, y<0
        print("Четвертая четверть")
else:
    if y > 0:           # x<0, y>0
        print("Вторая четверть")
    else:               # x<0, y<0
        print("Третья четверть")
```

В этом примере мы использовали *комментарии* – текст, который интерпретатор игнорирует. Комментариями в Pythone является символ # и весь текст после этого символа до конца строки. Желательно писать код так, чтобы комментарии были излишними, однако допускается писать их там, где возникают "призраки" (утверждения или теоремы, которые использованы при написании кода, но не следуют из самого кода). Однако код выше является **плохим примером документации**: комментарии врут, поскольку автором не учтены точки на осях.

Операторы сравнения

Как правило, в качестве проверяемого условия используется результат вычисления одного из следующих операторов сравнения:

Оператор	Значение
<	Меньше — условие верно, если первый операнд меньше второго.
>	Больше — условие верно, если первый операнд больше второго.
<=	Меньше или равно — условие верно, если первый операнд меньше или второму.
>=	Больше или равно — условие верно, если первый операнд больше или р второму.
==	Равенство. Условие верно, если два операнда равны.

Например, условие (x * x < 1000) означает «значение x * x меньше 1000», а условие (2 * x != y) означает «удвоенное значение переменной x не равно значению переменной y».

Операторы сравнения в можно объединять в цепочки, например, a == b == c или 1 <= x <= 10.

Тип данных bool

Операторы сравнения возвращают значения специального логического типа bool. Значения логического типа могут принимать одно из двух значений: True (истина) или False (ложь). Если преобразовать логическое True к типу int, то получится 1, а преобразование False даст 0. При обратном преобразовании число 0 преобразуется в False, а любое ненулевое число в True. При преобразовании str в bool пустая строка преобразовывается в False, а любая непустая строка в True.

Каскадные условные инструкции

Пример программы, определяющий четверть координатной плоскости, можно переписать, используя «каскадную» последовательность инструкцией if... elif... else:

```
x = int(input())
y = int(input())
if x > 0 and y > 0:
   print("Первая четверть")
elif x < 0 and y > 0:
   print("Вторая четверть")
elif x < 0 and y < 0:
   print("Третья четверть")
elif x > 0 and y < 0:
   print("Четвертая четверть")
else:
   print("Точка находится на осях или в центре координат.")
```

В такой конструкции условия if, ..., elif проверяются по очереди, выполняется блок, соответствующий первому из истинных условий. Если все проверяемые условия ложны, то выполняется блок else, если он присутствует. Обратите внимание, что таким образом мы чётче видим условия наступления случаев (нет "призраков"), а также отлавливаем ситуацию, когда точка не находится ни в одной из четвертей.

Цикл while

Цикл while («пока») позволяет выполнить одну и ту же последовательность действий, пока проверяемое условие истинно. Условие записывается до тела цикла и проверяется до выполнения тела цикла. Как правило, цикл while используется, когда невозможно определить точное значение количества проходов исполнения цикла.

Синтаксис цикла while в простейшем случае выглядит так:

```
while Условие:
   Блок_инструкций
```

При выполнении цикла while сначала проверяется условие. Если оно ложно, то выполнение цикла прекращается и управление передается на следующую инструкцию после тела цикла while. Если условие истинно, то выполняется инструкция, после чего условие проверяется снова и снова выполняется

инструкция. Так продолжается до тех пор, пока условие будет истинно. Как только условие станет ложно, работа цикла завершится и управление передастся следующей инструкции после цикла.

Например, следующий фрагмент программы напечатает на экран всех целые числа, не превосходящие n:

```
a = 1
while a <= n:
   print(a)
   a += 1
```

Общая схема цикла while в данном случае для перебора всех подходящих значений такая:

```
a = начальное_значение
while a_является_подходящим_числом:
   обработать_a
   перейти_к_следующему_a
```

Выведем все степени двойки, не превосходящие числа n:

```
a = 1
while a <= n:
   print(a)
   a *= 2
```

Цикл for

Цикл for может быть использован как более краткая альтернатива циклу while.

Для последовательного перебора целых чисел из диапазона [0; n) можно использовать цикл for:

```
for i in range(10):
  print(i)
```

Этот код по выполняемым действиям полностью соответствуют циклу while:

```
i = 0
while i < 10:
 print(i)
 i += 1
```

Можно задавать начальные и конечные значения для переменной цикла, а также шаг:

```
for i in range(20, 10, -2):
  print(i)
```

Аналогичный цикл while

```
i = 20
while i > 10:
  print(i)
  i -= 2
```

Определение функции. Оператор def

В языке программирования Python функции определяются с помощью оператора **def**. Рассмотрим код:

```
def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")
```

Это пример определения функции. Как и другие сложные инструкции вроде условного оператора и циклов функция состоит из заголовка и тела. Заголовок оканчивается двоеточием и переходом на новую строку. Тело имеет отступ.

Ключевое слово **def** сообщает интерпретатору, что перед ним определение функции. За **def** следует имя функции. Оно может быть любым, также как и всякий идентификатор, например, переменная. В программировании весьма желательно давать всему осмысленные имена. Так в данном случае функция названа "посчитать_еду" в переводе на русский.

После имени функции ставятся скобки. В приведенном примере они пустые. Это значит, что функция не принимает никакие данные из вызывающей ее программы. Однако она могла бы их принимать, и тогда в скобках были бы указаны так называемые параметры.

После двоеточия следует тело, содержащее инструкции, которые выполняются при вызове функции. Следует различать определение функции и ее вызов. В программном коде они не рядом и не вместе. Можно

определить функцию, но ни разу ее не вызвать. Нельзя вызвать функцию, которая не была определена. Определив функцию, но ни разу не вызвав ее, вы никогда не выполните ее тела.

Вызов функции

Рассмотрим полную версию программы с функцией:

```
def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")

print("Сколько бананов и ананасов для обезьян?")
count_food()

print("Сколько жуков и червей для ежей?")
count_food()

print("Сколько рыб и моллюсков для выдр?")
count_food()
```

После вывода на экран каждого информационного сообщения осуществляется вызов функции, который выглядит просто как упоминание ее имени со скобками. Поскольку в функцию мы ничего не передаем скобки опять же пустые. В приведенном коде функция вызывается три раза.

Когда функция вызывается, поток выполнения программы переходит к ее определению и начинает исполнять ее тело. После того, как тело функции исполнено, поток выполнения возвращается в основной код в то место, где функция вызывалась. Далее исполняется следующее за вызовом выражение.

В языке Python определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строка за строкой и о том, что находится ниже по течению, ему еще неизвестно. Поэтому если вызов функции предшествует ее определению, то возникает ошибка (выбрасывается исключение **NameError**):

```
print("Сколько бананов и ананасов для обезьян?")
count_food()
```

```
print("Сколько жуков и червей для ежей?")
count_food()

print("Сколько рыб и моллюсков для выдр?")
count_food()

def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")
```

Результат:

```
Сколько бананов и ананасов для обезьян?
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    count_food()
NameError: name 'count_food' is not defined
```

Для многих компилируемых языков это не обязательное условие. Там можно определять и вызывать функцию в произвольных местах программы. Однако для удобочитаемости кода программисты даже в этом случае предпочитают соблюдать определенные правила.

Модули

Встроенные в язык программирования функции доступны сразу. Чтобы их вызвать, не надо выполнять никаких дополнительных действий. Однако за время существования любого популярного языка на нем было написано столько функций и классов, которые оказались востребованными множеством программистов и в разных областях, что включить весь этот объем кода в сам язык если возможно, то нецелесообразно.

Чтобы разрешить проблему доступа к дополнительным возможностям языка, в программировании стало общепринятой практикой использовать так называемые модули, пакеты и библиотеки. Каждый модуль содержит коллекцию функций и классов, предназначенных для решения задач из определенной области. Так в модуле math языка Python содержатся математические функции, модуль random позволяет генерировать

псевдослучайные числа, в модуле datetime содержатся классы для работы с датами и временем, модуль sys предоставляет доступ к системным переменным и т. д.

Количество модулей для языка Python огромно, что связано с популярностью языка. Часть модулей собрана в так называемую стандартную библиотеку. Стандартная она потому, что поставляется вместе с установочным пакетом. Однако существуют сторонние библиотеки. Они скачиваются и устанавливаются отдельно.

Для доступа к функционалу модуля, его надо импортировать в программу. После импорта интерпретатор "знает" о существовании дополнительных классов и функций и позволяет ими пользоваться.

В Питоне импорт осуществляется командой **import**. При этом существует несколько способов импорта. Рассмотрим работу с модулем на примере **math**. Итак,

```
>>> import math
```

Ничего не произошло. Однако в глобальной области видимости появилось имя **math**. Если до импорта вы упомянули бы имя **math**, то возникла бы ошибка **NameError**. Теперь же

```
>>> math
<module 'math' (built-in)>
```

В программе завелся объект **math**, относящийся к классу **module**.

Чтобы увидеть перечень функций, входящих в этот модуль, воспользуемся встроенной в Python функцией dir(), передав ей в качестве аргумента имя модуля:

```
>>> dir(math)
['__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
 'hypot', 'inf', 'isclose', 'isfinite',
```

```
'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
```

Проигнорируем имена с двойными подчеркиваниями. Все остальное – имена функций и констант (переменных, которые не меняют своих значений), включенных в модуль math. Чтобы вызвать функцию из модуля, надо впереди написать имя модуля, поставить точку, далее указать имя функции, после чего в скобках передать аргументы, если они требуются. Например, чтобы вызвать функцию pow из math, надо написать так:

```
>>> math.pow(2, 2)
4.0
```

Обратите внимание, эта другая функция pow(), не та, что встроена в сам язык. "Обычная" функция pow() возвращает целое, если аргументы целые числа:

```
>>> pow(2, 2)
4
```

Для обращения к константе скобки не нужны:

```
>>> math.pi
3.141592653589793
```

Если мы не знаем, что делает та или иная функция, то можем получить справочную информацию о ней с помощью встроенной в язык Python функции help():

```
>>> help(math.gcd)
Help on built-in function gcd in module math:

gcd(...)
    gcd(x, y) -> int
    greatest common divisor of x and y
```

Для выхода из интерактивной справки надо нажать клавишу q. В данном случае сообщается, что функция возвращает целое число, и это наибольший общий делитель для чисел *x* и *y*. Описание модулей и их содержания также можно посмотреть в официальной документации на сайте python.org.

Второй способ импорта – это когда импортируется не сам модуль, а только необходимые функции из него.

```
>>> from math import gcd, sqrt, hypot
```

Перевести можно как "из модуля **math** импортировать функции **gcd**, **sqrt** и **hypot**".

В таком случае при их вызове не надо перед именем функции указывать имя модуля:

```
>>> gcd(100, 150)
50
>>> sqrt(16)
4.0
>>> hypot(3, 4)
5.0
```

Чтобы импортировать сразу все функции из модуля:

```
>>> from math import *
```

Импорт через **from** не лишен недостатка. В программе уже может быть идентификатор с таким же именем, как имя одной из импортируемых функций или констант. Ошибки не будет, но одно из них окажется "затерто":

```
>>> pi = 3.14
>>> from math import pi
>>> pi
3.141592653589793
```

Здесь исчезает значение 3.14, присвоенное переменной *pi*. Это имя теперь указывает на число из модуля **math**. Если импорт сделать раньше, чем присвоение значения *pi*, то будет все наоборот:

```
>>> from math import pi
>>> pi = 3.14
>>> pi
3.14
```

В этой связи более опасен именно импорт всех функций. Так как в этом случае очень легко не заметить подмены значений идентификаторов.

Однако можно изменить имя идентификатора из модуля на какое угодно:

```
>>> from math import pi as P
>>> P
3.141592653589793
>>> pi
3.14
```

В данном случае константа **pi** из модуля импортируется под именем *P*. Смысл подобных импортов в сокращении имен, так как есть модули с длинными именами, а имена функций и классов в них еще длиннее. Если в программу импортируется всего пара сущностей, и они используются в ней часто, то имеет смысл переименовать их на более короткий вариант. Сравните:

```
>>> import calendar
>>> calendar.weekheader(2)
'Mo Tu We Th Fr Sa Su'
```

и

```
>>> from calendar import weekheader as week
>>> week(3)
'Mon Tue Wed Thu Fri Sat Sun'
```

Во всех остальных случаях лучше оставлять идентификаторы содержимого модуля в пространстве имен самого модуля и получать доступ к ним через имя модуля, то есть выполнять импорт командой import имя_модуля, а вызывать, например, функции через имя_модуля.имя_функции().

Списки

Список в Python – это встроенный тип (класс) данных, представляющий собой одну из разновидностей структур данных. Структуру данных можно представить, как сложную единицу, объединяющую в себе группу более простых. Каждая разновидность структур данных имеет свои особенности. Список – это изменяемая последовательность произвольных элементов.

В большинстве других языков программирования есть такой широко используемый тип данных как массив. В Питоне такого встроенного типа нет. Однако списки условно можно считать аналогом массивов за одним исключением. Составляющие массив элементы должны принадлежать одному типу данных, для списков такого ограничения нет.

Например, массив может содержать только целые числа или только вещественные числа или только строки. Список также может содержать элементы только одного типа, что делает его внешне неотличимым от массива. Но вполне допустимо, чтобы в одном списке содержались как числа, так и строки, а также что-нибудь еще.

Создавать списки можно разными способами. Создадим его простым перечисление элементов:

```
>>> a = [12, 3.85, "black", -4]
>>> a
[12, 3.85, 'black', -4]
```

Итак, у нас имеется список, присвоенный переменной *a*. В Python список определяется квадратными скобками. Он содержит четыре элемента. Если где-то в программе нам понадобится весь этот список, мы получим доступ к нему, указав всего лишь одну переменную – *a*.

Элементы в списке упорядочены, имеет значение в каком порядке они расположены. Каждый элемент имеет свой индекс, или номер. Индексация начинается с нуля. В данном случае число 12 имеет индекс 0, строка "black" – индекс 2. Чтобы извлечь конкретный элемент, надо после имени переменной указать в квадратных скобках его индекс:

```
>>> a[0]
12
>>> a[3]
-4
```

В Python существует также индексация с конца. Она начинается с -1:

```
>>> a[-1]
-4
>>> a[-2]
'black'
>>> a[-3], a[-4]
(3.85, 12)
```

Часто требуется извлечь не один элемент, а так называемый срез – часть списка. В этом случае указывается индекс первого элемента среза и индекс следующего за последним элементом среза:

```
>>> a[0:2]
[12, 3.85]
```

В данном случае извлекаются первые два элемента с индексами 0 и 1. Элемент с индексом 2 в срез уже не входит. В таком случае возникает вопрос, как извлечь срез, включающий в себя последний элемент? Если какой-либо индекс не указан, то считается, что имеется в виду начало или конец:

```
>>> a[:3]
[12, 3.85, 'black']
>>> a[2:]
['black', -4]
>>> a[:]
[12, 3.85, 'black', -4]
```

Списки – изменяемые объекты. Это значит, что в них можно добавлять элементы, удалять их, изменять существующие. Проще всего изменить значение элемента. Для этого надо обратиться к нему по индексу и перезаписать значение в заданной позиции:

```
>>> a[1] = 4
>>> a
[12, 4, 'black', -4]
```

Добавлять и удалять лучше с помощью специальных встроенных методов списка:

```
>>> a.append('wood')
>>> a
[12, 4, 'black', -4, 'wood']
>>> a.insert(1, 'circle')
>>> a
[12, 'circle', 4, 'black', -4, 'wood']
>>> a.remove(4)
>>> a
[12, 'circle', 'black', -4, 'wood']
>>> a.pop()
'wood'
>>> a
```

```
[12, 'circle', 'black', -4]
>>> a.pop(2)
'black'
>>> a
[12, 'circle', -4]
```

Перечень всех методов списка можно узнать с помощью встроенной в Python функции dir(), передав в качестве аргумента переменную, связанную со списком, или название класса (в данном случае – **list**). В полученном из dir() списке надо смотреть имена без двойных подчеркиваний.

Для получения информации о конкретном методе следует воспользоваться встроенной функцией help(), передав ей в качестве аргумента имя метода, связанное с объектом или классом. Например, help(a.pop) или help(list.index). Выход из справки – q.

Можно изменять списки не используя методы, а с помощью взятия и объединения срезов:

```
>>> b = [1, 2, 3, 4, 5, 6]
>>> b = b[:2] + b[3:]
>>> b
[1, 2, 4, 5, 6]
```

Здесь берется срез из первых двух элементов и срез, начиная с четвертого элемента (индекс 3) и до конца. После чего срезы объединяются с помощью оператора "сложения".

Можно изменить не один элемент, а целый срез:

```
>>> mylist = ['ab','ra','ka','da','bra']
>>> mylist[0:2] = [10,20]
>>> mylist
[10, 20, 'ka', 'da', 'bra']
```

Пример создания пустого списка с последующим заполнением его в цикле случайными числами:

```
>>> import random
>>> c = []
>>> i = 0
>>> while i < 10:
```

```
...     c.append(random.randint(0,100))
...     i += 1
...
>>> c
[30, 44, 35, 77, 53, 44, 49, 17, 61, 82]
```

Кортежи

Кортежи (tuple) в Python – это те же списки за одним исключением. Кортежи неизменяемые структуры данных. Так же как списки они могут состоять из элементов разных типов, перечисленных через запятую. Кортежи заключаются в круглые, а не квадратные скобки.

```
>>> a = (10, 2.13, "square", 89, 'C')
>>> a
(10, 2.13, 'square', 89, 'C')
```

Из кортежа можно извлекать элементы и брать срезы:

```
>>> a[3]
89
>>> a[1:3]
(2.13, 'square')
```

Однако изменять его элементы нельзя:

```
>>> a[0] = 11
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
support item assignment
```

Также у типа **tuple** нет методов для добавления и удаления элементов.

Возникает резонный вопрос. Зачем в язык программирования был введен этот тип данных, по-сути представляющий собой неизменяемый список? Дело в том, что иногда надо защитить список от изменений. Преобразовать же кортеж в список, если это потребуется, как и выполнить обратную операцию легко с помощью встроенных в Python функций list() и tuple():

```
>>> a = (10, 2.13, "square", 89, 'C')
>>> b = [1, 2, 3]
>>> c = list(a)
```

```
>>> d = tuple(b)
>>> c
[10, 2.13, 'square', 89, 'C']
>>> d
(1, 2, 3)
```

Рассмотрим случай, когда уместно использовать кортежи. В Python изменяемые объекты передаются в функцию по ссылке. Это значит, что не создается копия объекта, а переменной-параметру присваивается ссылка на уже существующий объект. В итоге, если в теле функции объект изменяется, то эти изменения касаются глобального объекта.

```
def add_num(seq, num):
    for i in range(len(seq)):
        seq[i] += num
    return seq

origin = [3, 6, 2, 6]
changed = add_num(origin, 3)

print(origin)
print(changed)
```

Данная программа неправильная. Хотя никаких выбросов исключений не произойдет, она содержит логическую ошибку. На выводе получаем:

```
[6, 9, 5, 9]
[6, 9, 5, 9]
```

То есть исходный список был также изменен. Параметр *seq* содержал ссылку не на свой локальный список, а на список-оригинал. Таким образом, в операторе **return** здесь нет смыла. Если функция замысливалась как изменяющая глобальный список, то программа должна выглядеть так:

```
def add_num(seq, num):
    for i in range(len(seq)):
        seq[i] += num

origin = [3, 6, 2, 6]
add_num(origin, 3)
```

```
print(origin)
```

Что делать, если все же требуется не изменять исходный список, а сформировать по нему новый. Задачу можно решить несколькими способами. Во-первых, в функции можно создать локальный список, после чего возвращать его:

```
def add_num(seq, num):
    new_seq = []
    for i in seq:
        new_seq.append(i + num)
    return new_seq

origin = [3, 6, 2, 6]
changed = add_num(origin, 3)

print(origin)
print(changed)
Результат:
[3, 6, 2, 6]
[6, 9, 5, 9]
```

Исходный список в функции не меняется. Его элементы лишь перебираются.

Второй способ защитить список-оригинал – использовать кортеж. Этот способ более надежный, так как в больших программах трудно отследить, что ни одна функция не содержит команд изменения глобальных данных.

Хотя преобразовывать к кортежу можно как при передаче в функцию, так и в самой функции, лучше сразу делать глобальный список кортежем. Поскольку неизменяемые объекты передаются по значению, а не по ссылке, то в функцию будет поступать копия структуры, а не оригинал. Даже если туда передается оригинал, изменить его невозможно. Можно лишь, как вариант, скопировать его и/или изменить тип, создав тем самым локальную структуру, и делать с ней все, что заблагорассудится.

```
def add_num(seq, num):
    seq = list(seq)
    for i in range(len(seq)):
        seq[i] += num
```

```
    return seq

origin = (3, 6, 2, 6)
changed = add_num(origin, 3)

print(origin)
print(changed)
```

Списки в кортежах

Кортежи могут содержать списки, также как списки быть вложенными в другие списки.

```
>>> nested = (1, "do", ["param", 10, 20])
```

Как вы думаете, можем ли мы изменить список ["param", 10, 20] вложенный в кортеж *nested*? Список изменяем, кортеж – нет. Если вам кажется, что нельзя, то вам кажется неправильно. На самом деле можно:

```
>>> nested[2][1] = 15
>>> nested
(1, 'do', ['param', 15, 20])
```

Примечание. Выражения типа nested[2][1] используются для обращения к вложенным объектам. Первый индекс указывает на позицию вложенного объекта, второй – индекс элемента внутри вложенного объекта. Так в данном случае сам список внутри кортежа имеет индекс 2, а элемент списка 10 – индекс 1 в списке.

Странная ситуация. Кортеж неизменяем, но мы все-таки можем изменить его. На самом деле кортеж остается неизменяемым. Просто в нем содержится не сам список, а ссылка на него. Ее изменить нельзя. Но менять сам список можно.

Чтобы было проще понять, перепишем кортеж так:

```
>>> l = ["param", 10, 20]
>>> t = (1, "do", l)
>>> t
(1, 'do', ['param', 10, 20])
```

Кортеж содержит переменную-ссылку. Поменять ее на другую ссылку нельзя. Но кортеж не содержит самого списка. Поэтому его можно менять как угодно:

```
>>> l.pop(0)
'param'
>>> t
(1, 'do', [10, 20])
```

Однако такой номер не проходит с неизменяемыми типами:

```
>>> a = "Kat"
>>> t = (a, l)
>>> t
('Kat', [10, 20])
>>> a = "Bat"
>>> t
('Kat', [10, 20])
```

Они передаются в кортеж, как и в функцию – по значению. То есть их значение копируется в момент передачи.

Словари

В языке программирования Python словари (тип **dict**) представляют собой еще одну разновидность структур данных наряду со списками и кортежами. *Словарь - это **изменяемый** (как список) **неупорядоченный** (в отличие от строк, списков и кортежей) набор элементов "ключ:значение".*

"Неупорядоченный" – значит, что последовательность расположения пар не важна, в следствие чего обращение к элементам по индексам невозможно.

В других языках структуры, схожие со словарями, называются по-другому. Например, в Java подобный тип данных называется отображением.

Чтобы представление о словаре стало более понятным, проведем аналогию с обычным словарем, например, англо-русским. На каждое английское слово в таком словаре есть русское слово-перевод: cat – кошка, dog – собака, table – стол и т. д. Если англо-русский словарь описать с помощью Python, то английские слова можно сделать ключами, а русские – их значениями:

{'cat': 'кошка', 'dog': 'собака', 'bird': 'птица'}

Обратите внимание, что для определения словаря используются фигурные скобки. Синтаксис словаря на Питоне описывается такой схемой:



В словаре доступ к значениям осуществляется не по индексам, а **по ключам**, которые заключаются в квадратные скобки (по аналогии с индексами списков):

```
>>> a['cat']
'кошка'
>>> a['bird']
'птица'
```

Словари, как и списки, являются изменяемым типом данных: позволительно изменять, добавлять и удалять элементы (пары "ключ:значение"). Изначально словарь можно создать пустым (например, d = {}) и потом заполнить его элементами. Добавление и изменение имеет одинаковый синтаксис: словарь[ключ] = значение. Ключ может быть как уже существующим (тогда происходит изменение значения), так и новым (происходит добавление элемента словаря). Удаление элемента осуществляется с помощью встроенной оператора **del** языка Python.

```
>>> a = {}
>>> a[1] = 2.34
>>> a[2] = 4.88
>>> a
{1: 2.34, 2: 4.88}
>>> a[1] = 3.01
>>> a
{1: 3.01, 2: 4.88}
>>> del a[2]
>>> a
{1: 3.01}
```

В словаре не может быть двух элементов с одинаковыми ключами. Однако могут быть одинаковые значения у разных ключей.

Ключом может быть любой неизменяемый тип данных. Значением – любой тип данных. Значения словарей вполне могут быть структурами, например, другими словарями или списками.

```
>>> nums = {'one': (1, 'I'), 'two': (2, 'II')}
>>> person1 = {'name': 'Tom', 'data': [4, 2.5]}
>>> person1['data'][0]
4
```

Перебор элементов словаря в цикле for

Элементы словаря перебираются в цикле **for** также, как элементы других сложных объектов. Однако "по-умолчанию" извлекаются только ключи:

```
>>> nums
{1: 'one', 2: 'two', 3: 'three'}
>>> for i in nums:
...     print(i)
...
1
2
3
```

Но по ключам всегда можно получить значения:

```
>>> for i in nums:
...     print(nums[i])
...
one
two
three
```

С другой стороны у словаря как класса есть метод items(), который создает особую структуру, состоящую из кортежей. Каждый кортеж включает ключ и значение:

```
>>> n = nums.items()
>>> n
dict_items([(1, 'one'), (2, 'two'), (3, 'three')])
```

В цикле **for** можно распаковывать кортежи, таким образом сразу извлекая как ключ, так и его значение:

```
>>> for key, value in nums.items():
...     print(key, 'is', value)
...
1 is one
2 is two
3 is three
```

Методы словаря keys() и values() позволяют получить отдельно перечни ключей и значений. Так что если, например, надо перебрать только значения или только ключи, лучше воспользоваться одним из этих методов:

```
>>> v_nums = []
>>> for v in nums.values():
...     v_nums.append(v)
...
>>> v_nums
['one', 'two', 'three']
```

Методы словаря

Кроме рассмотренных выше трех методов items(), keys() и values() словари обладают еще восемью. Это методы clear(), copy(), fromkeys(), get(), pop(), popitem(), setdefault(), update().

Метод clear() удаляет все элементы словаря, но не удаляет сам словарь. В итоге остается пустой словарь:

```
>>> nums
{'one': (1, 'I'), 'two': (2, 'II')}
>>> nums.clear()
>>> nums
{}
```

Словарь – это изменяемый тип данных. Следовательно, как и список он передается в функцию по ссылке. Поэтому иногда, чтобы избежать нежелательного изменения глобального словаря его копируют. Это делают и с другими целями.

```
>>> n1 = {'a': 1, 'b': 2}
>>> n2 = n1
>>> n3 = n1.copy()
```

```
>>> n2['c'] = 3
>>> n1
{'a': 1, 'b': 2, 'c': 3}
>>> n2
{'a': 1, 'b': 2, 'c': 3}
>>> n3
{'a': 1, 'b': 2}
```

Метод fromkeys() позволяет создать словарь из списка, элементы которого становятся ключами. Применять метод можно как классу **dict**, так и к его объектам:

```
>>> a = [1, 2, 3]
>>> c = dict.fromkeys(a)
>>> c
{1: None, 2: None, 3: None}
>>> d = dict.fromkeys(a, 10)
>>> d
{1: 10, 2: 10, 3: 10}
```

Метод get() позволяет получить элемент по его ключу:

```
>>> nums.get(1)
'one'
```

Равносильно nums[1].

Метод pop() удаляет из словаря элемент по указанному ключу и возвращает значение удаленной пары. Метод popitem() не принимает аргумента, удаляет элемент по принципу "последним вошел, первым вышел" и возвращает его в программу в виде кортежа (ключ, значение).

```
>>> nums = {1: 'one', 2: 'two', 3: 'three'}
>>> nums.pop(1)
'one'
>>> nums
{2: 'two', 3: 'three'}
>>> nums.popitem()
(3, 'three')
>>> nums
{2: 'two'}
```

С помощью setdefault() можно добавить элемент в словарь:

```
>>> nums.setdefault(4, 'four')
'four'
>>> nums
{3: 'three', 4: 'four'}
```

Равносильно nums[4] = 'four', если элемент с ключом 4 отсутствует в словаре. Если он уже есть, то nums[4] = 'four' перезапишет старое значение, setdefault() – нет.

С помощью update() можно добавить в словарь другой словарь:

```
>>> nums.update({6: 'six', 7: 'seven'})
>>> nums
{3: 'three', 4: 'four', 6: 'six', 7: 'seven'}
```

Также метод обновляет значения существующих ключей. Включает еще ряд особенностей.

Создание собственных Modules

Модуль — это многоразовый автономный сценарий, который Ansible запускает от вашего имени локально или удаленно. Модули взаимодействуют с вашим локальным компьютером, API или удаленной системой для выполнения определенных задач, таких как изменение пароля базы данных или запуск облачного экземпляра. Каждый модуль может использоваться Ansible API или программами **ansible** или **ansible-playbook** . Модуль предоставляет определенный интерфейс, принимает аргументы и возвращает информацию в Ansible, выводя строку JSON на стандартный вывод перед выходом.

Если вам нужна функциональность, которой нет ни в одном из тысяч модулей Ansible, найденных в коллекциях, вы можете легко написать свой собственный модуль. Когда вы пишете модуль для локального использования, вы можете выбрать любой язык программирования и следовать своим собственным правилам. Используйте этот раздел, чтобы узнать, как создать модуль Ansible в Python. После того, как вы создадите модуль, вы должны добавить его локально в соответствующий каталог, чтобы Ansible мог его найти и выполнить.

Подготовка среды для разработки модулей Ansible

Установка необходимых компонентов через apt (Ubuntu)

Из-за зависимостей (например, ansible -> paramiko -> pynacl -> libffi):

```
sudo apt update
sudo apt install build-essential libssl-dev libffi-dev python-dev
```

Установка необходимых компонентов через yum (CentOS)

Из-за зависимостей (например, ansible -> paramiko -> pynacl -> libffi):

```
sudo yum check-update
sudo yum update
sudo yum group install "Development Tools"
sudo yum install python3-devel openssl-devel libffi libffi-devel
```

Создание среды разработки (этапы, не зависящие от платформы)

1. Клонируйте репозиторий Ansible: `$ git clone https://github.com/ansible/ansible.git`

2. Измените каталог на корневой каталог репозитория:

```
$ cd ansible
```

3. Создайте виртуальную среду:

```
$ python3 -m venv venv$ virtualenv venv$ pip install virtualenv
```

4. Активируйте виртуальную среду:

```
$ . venv/bin/activate
```

5. Требования к разработке установки:

```
$ pip install -r requirements.txt
```

6. Запустите сценарий установки среды для каждого нового процесса оболочки разработки:

```
$ . hacking/env-setup
```

Примечание

После первоначальной настройки, описанной выше, каждый раз, когда вы будете готовы начать разработку Ansible, вы сможете просто запустить следующее из корня репозитория

```
Ansible: $ . venv/bin/activate && . hacking/env-setup
```

Создание модуля информации или фактов

В Ansible 2.8 и более поздних версиях у нас есть два типа информационных модулей: это `*_info` и `*_facts`.

Если модуль назван `<something>_facts`, это должно быть потому, что его основная цель - возврат `ansible_facts`. Не называйте модули, которые этого не делают, с помощью `_facts`. Используйте только `ansible_facts` информацию, относящуюся к хост-компьютеру, например, сетевые интерфейсы и их конфигурацию, какая операционная система и какие программы установлены.

Модули, которые запрашивают/возвращают общую информацию (а не `ansible_facts`), должны называться `_info`. Общая информация — это информация, не относящаяся к хосту, например, информация об онлайн-/облачных службах (вы можете получить доступ к разным учетным записям для одной и той же онлайн-службы с одного и того же хоста), или

информация о виртуальных машинах и контейнерах, доступных с компьютера, или информация об отдельных файлах. или программы.

Модули информации и фактов такие же, как и любой другой модуль Ansible, с несколькими незначительными требованиями:

1. Они ДОЛЖНЫ быть названы `<something>_info` или `<something>_facts`, где `<something>` в единственном числе.
2. Информационные `*_info` модули ДОЛЖНЫ возвращаться в виде словаря результатов, чтобы другие модули могли получить к ним доступ.
3. Модули фактов `*_facts` ДОЛЖНЫ возвращаться в ansible_facts поле словаря результатов, чтобы другие модули могли получить к ним доступ.
4. Они ДОЛЖНЫ поддерживать check_mode .
5. Они НЕ ДОЛЖНЫ вносить какие-либо изменения в систему.
6. Они ДОЛЖНЫ документировать возвращаемые поля и примеры .

Чтобы создать информационный модуль:

1. Перейдите в правильный каталог для вашего нового модуля: `$ cd lib/ansible/modules/`. Если вы разрабатываете модуль, `$ cd plugins/modules/` используя коллекцию, внутри дерева разработки вашей коллекции.
2. Создайте новый файл модуля: `$ touch my_test_info.py`.
3. Вставьте содержимое ниже в ваш новый файл информационного модуля. Он включает в себя необходимый формат и документацию Ansible , простую спецификацию аргументов для объявления параметров модуля и некоторый пример кода.
4. Измените и расширьте код, чтобы он делал то, что вы хотите.

```
#!/usr/bin/python

# Copyright: (c) 2020, Your Name <YourName@example.org>
# GNU General Public License v3.0+ (see COPYING or
https://www.gnu.org/licenses/gpl-3.0.txt)
```

```
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

DOCUMENTATION = r'''
---
module: my_test_info

short_description: This is my test info module

version_added: "1.0.0"

description: This is my longer description explaining my test info module.

options:
    name:
        description: This is the message to send to the test module.
        required: true
        type: str

author:
    - Your Name (@yourGitHubHandle)
'''

EXAMPLES = r'''
# Pass in a message
- name: Test with a message
  my_namespace.my_collection.my_test_info:
    name: hello world
'''

RETURN = r'''
# These are examples of possible return values, and in general should use other
names for return values.
original_message:
    description: The original name param that was passed in.
    type: str
```

```
    returned: always
    sample: 'hello world'
message:
    description: The output message that the test module generates.
    type: str
    returned: always
    sample: 'goodbye'
my_useful_info:
    description: The dictionary containing information about your system.
    type: dict
    returned: always
    sample: {
        'foo': 'bar',
        'answer': 42,
    }
'''

from ansible.module_utils.basic import AnsibleModule


def run_module():
    # define available arguments/parameters a user can pass to the module
    module_args = dict(
        name=dict(type='str', required=True),
    )

    # seed the result dict in the object
    # we primarily care about changed and state
    # changed is if this module effectively modified the target
    # state will include any data that you want your module to pass back
    # for consumption, for example, in a subsequent task
    result = dict(
        changed=False,
        original_message='',
        message='',
        my_useful_info={},
```

```
    )

    # the AnsibleModule object will be our abstraction working with Ansible
    # this includes instantiation, a couple of common attr would be the
    # args/params passed to the execution, as well as if the module
    # supports check mode
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    # if the user is working with this module in only check mode we do not
    # want to make any changes to the environment, just return the current
    # state with no modifications
    if module.check_mode:
        module.exit_json(**result)

    # manipulate or modify the state as needed (this is going to be the
    # part where your module will do what it needs to do)
    result['original_message'] = module.params['name']
    result['message'] = 'goodbye'
    result['my_useful_info'] = {
        'foo': 'bar',
        'answer': 42,
    }
    # in the event of a successful module execution, you will want to
    # simple AnsibleModule.exit_json(), passing the key/value results
    module.exit_json(**result)


def main():
    run_module()


if __name__ == '__main__':
    main()
```

Используйте тот же процесс для создания модуля фактов.

```
#!/usr/bin/python

# Copyright: (c) 2020, Your Name <YourName@example.org>
# GNU General Public License v3.0+ (see COPYING or
https://www.gnu.org/licenses/gpl-3.0.txt)
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

DOCUMENTATION = r'''
---
module: my_test_facts

short_description: This is my test facts module

version_added: "1.0.0"

description: This is my longer description explaining my test facts module.

author:
   - Your Name (@yourGitHubHandle)
'''

EXAMPLES = r'''
- name: Return ansible_facts
  my_namespace.my_collection.my_test_facts:
'''

RETURN = r'''
# These are examples of possible return values, and in general should use other
names for return values.
ansible_facts:
  description: Facts to add to ansible_facts.
  returned: always
  type: dict
  contains:
```

```
    foo:
      description: Foo facts about operating system.
      type: str
      returned: when operating system foo fact is present
      sample: 'bar'
    answer:
      description:
      - Answer facts about operating system.
      - This description can be a list as well.
      type: str
      returned: when operating system answer fact is present
      sample: '42'
'''

from ansible.module_utils.basic import AnsibleModule


def run_module():
    # define available arguments/parameters a user can pass to the module
    module_args = dict()

    # seed the result dict in the object
    # we primarily care about changed and state
    # changed is if this module effectively modified the target
    # state will include any data that you want your module to pass back
    # for consumption, for example, in a subsequent task
    result = dict(
        changed=False,
        ansible_facts=dict(),
    )

    # the AnsibleModule object will be our abstraction working with Ansible
    # this includes instantiation, a couple of common attr would be the
    # args/params passed to the execution, as well as if the module
    # supports check mode
    module = AnsibleModule(
```

```
    argument_spec=module_args,
    supports_check_mode=True
  )

  # if the user is working with this module in only check mode we do not
  # want to make any changes to the environment, just return the current
  # state with no modifications
  if module.check_mode:
    module.exit_json(**result)

  # manipulate or modify the state as needed (this is going to be the
  # part where your module will do what it needs to do)
  result['ansible_facts'] = {
    'foo': 'bar',
    'answer': '42',
  }
  # in the event of a successful module execution, you will want to
  # simple AnsibleModule.exit_json(), passing the key/value results
  module.exit_json(**result)


def main():
  run_module()


if __name__ == '__main__':
  main()
```

Создание модуля

Чтобы создать модуль:

1. Перейдите в правильный каталог для вашего нового модуля: `$ cd lib/ansible/modules/`. Если вы разрабатываете модуль в коллекции , `$ cd plugins/modules/` внутри дерева разработки вашей коллекции.
2. Создайте новый файл модуля: `$ touch my_test.py`.

3. Вставьте содержимое ниже в ваш новый файл модуля. Он включает в себя необходимый формат и документацию Ansible , простую спецификацию аргументов для объявления параметров модуля и некоторый пример кода.
4. Измените и расширьте код, чтобы он делал то, что вы хотите.

```
#!/usr/bin/python

# Copyright: (c) 2018, Terry Jones <terry.jones@example.org>
# GNU General Public License v3.0+ (see COPYING or
https://www.gnu.org/licenses/gpl-3.0.txt)
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

DOCUMENTATION = r'''
---
module: my_test

short_description: This is my test module

# If this is part of a collection, you need to use semantic versioning,
# i.e. the version is of the form "2.5.0" and not "2.4".
version_added: "1.0.0"

description: This is my longer description explaining my test module.

options:
    name:
        description: This is the message to send to the test module.
        required: true
        type: str
    new:
        description:
            - Control to demo if the result of this module is changed or not.
            - Parameter description can be a list as well.
        required: false
```

```
        type: bool
# Specify this value according to your collection
# in format of namespace.collection.doc_fragment_name
extends_documentation_fragment:
    - my_namespace.my_collection.my_doc_fragment_name

author:
    - Your Name (@yourGitHubHandle)
'''

EXAMPLES = r'''
# Pass in a message
- name: Test with a message
  my_namespace.my_collection.my_test:
    name: hello world

# pass in a message and have changed true
- name: Test with a message and changed output
  my_namespace.my_collection.my_test:
    name: hello world
    new: true

# fail the module
- name: Test failure of the module
  my_namespace.my_collection.my_test:
    name: fail me
'''

RETURN = r'''
# These are examples of possible return values, and in general should use other
names for return values.
original_message:
    description: The original name param that was passed in.
    type: str
    returned: always
    sample: 'hello world'
```

```
message:
    description: The output message that the test module generates.
    type: str
    returned: always
    sample: 'goodbye'
'''

from ansible.module_utils.basic import AnsibleModule


def run_module():
    # define available arguments/parameters a user can pass to the module
    module_args = dict(
        name=dict(type='str', required=True),
        new=dict(type='bool', required=False, default=False)
    )

    # seed the result dict in the object
    # we primarily care about changed and state
    # changed is if this module effectively modified the target
    # state will include any data that you want your module to pass back
    # for consumption, for example, in a subsequent task
    result = dict(
        changed=False,
        original_message='',
        message=''
    )

    # the AnsibleModule object will be our abstraction working with Ansible
    # this includes instantiation, a couple of common attr would be the
    # args/params passed to the execution, as well as if the module
    # supports check mode
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )
```

```
    # if the user is working with this module in only check mode we do not
    # want to make any changes to the environment, just return the current
    # state with no modifications
    if module.check_mode:
        module.exit_json(**result)

    # manipulate or modify the state as needed (this is going to be the
    # part where your module will do what it needs to do)
    result['original_message'] = module.params['name']
    result['message'] = 'goodbye'

    # use whatever logic you need to determine whether or not this module
    # made any modifications to your target
    if module.params['new']:
        result['changed'] = True

    # during the execution of the module, if there is an exception or a
    # conditional state that effectively causes a failure, run
    # AnsibleModule.fail_json() to pass in the message and the result
    if module.params['name'] == 'fail me':
        module.fail_json(msg='You requested this to fail', **result)

    # in the event of a successful module execution, you will want to
    # simple AnsibleModule.exit_json(), passing the key/value results
    module.exit_json(**result)


def main():
    run_module()


if __name__ == '__main__':
    main()
```

Проверка кода вашего модуля

Локальная проверка кода модуля

Если вашему модулю не нужно нацеливаться на удаленный хост, вы можете быстро и легко проверить свой код локально следующим образом:

- Создайте файл аргументов, базовый файл конфигурации JSON, который передает параметры вашему модулю, чтобы вы могли его запустить. Назовите файл аргументов `/tmp/args.json` и добавьте следующее содержимое:

```
{
   "ANSIBLE_MODULE_ARGS": {
      "name": "hello",
      "new": true
   }
}
```

- Если вы используете виртуальную среду (которая настоятельно рекомендуется для разработки), активируйте ее: `$ . venv/bin/activate`
- Настраиваем среду для разработки: `$ . hacking/env-setup`
- Запустите тестовый модуль локально и напрямую: `$ python -m ansible.modules.my_test /tmp/args.json`

Это должно возвращать вывод следующим образом:

```
{"changed": true, "state": {"original_message": "hello", "new_message": "goodbye"}, "invocation": {"module_args": {"name": "hello", "new": true}}}
```

Проверка кода вашего модуля в playbook

Следующим шагом в проверке вашего нового модуля является его использование с плейбуком Ansible.

- Создайте playbook в любом каталоге: `$ touch testmod.yml`
- Добавьте следующее в новый файл playbook:

```
- name: test my new module
  hosts: localhost
  tasks:
  - name: run the new module
    my_test:
      name: 'hello'
```

```
    new: true
  register: testout
- name: dump test output
  debug:
    msg: '{{ testout }}'
```

- Запустите playbook и проанализируйте вывод: $ ansible-playbook ./testmod.yml

Модуль decs_vm

Назначение

Модуль *decs_vm* предназначен для выполнения следующих действий над виртуальными серверами в облачной платформе DECS:

1. Создание нового виртуального сервера.
2. Изменение конфигурации существующего виртуального сервера:
    - Изменение количества выделенных CPU и виртуальной ОЗУ. Следует иметь ввиду, что изменение этих параметров в меньшую сторону может потребовать перезагрузки гостевой ОС.
    - Увеличение размера загрузочного диска.
    - Добавление, редактирование и удаление правил трансляции сетевых портов (управление *port forwarding rules*).
    - Назначение и удаление прямого внешнего IP адреса.
3. Изменение состояния существующего виртуального сервера:
    - Выключение / включение.
    - Перезагрузка, приостановка / возобновление работы гостевой ОС.
4. Удаление существующего виртуального сервера.
5. Получение информации о существующем виртуальном сервере без изменения его конфигурации.

Помимо модуля *decs_vm* также имеются модули, предназначенные для решения специфических задач:

- *decs_jwt* - для получения авторизационного токена (JWT - JSON Web Token).
- *decs_osimage* - для получения идентификатора образа ОС.
- *decs_vdc* - для управления виртуальными сетевыми сегментами.

- *decs_gpu* - для управления виртуальными GPU ресурсами (Graphical Processing Unit).

Системные требования

Для корректного функционирования модулей управления облачной инфраструктурой DECS на Ansible-сервере требуется наличие следующего программного обеспечения:

1. Python модуль *decs_utility.py* (содержит библиотечные функции, поддерживающие функциональность Ansible-модулей для управления облачными ресурсами DECS).
2. Python модуль *PyJWT* для работы с JSON Web Tokens (JWT).
    - Для установки можно использовать команду: pip install PyJWT <enter>
3. Python модуль *requests* для работы с API.
    - Для установки можно использовать команду: pip install requests <enter>
4. Ansible версии 2.2 или выше.
5. Интерпретатор языка Python версии 2.6 или выше.

Установка

Актуальный исходный код модуля *decs_vm* и необходимых для его работы библиотечных модулей на языке Python доступен по адресу https://github.com/rudecs/decsamo .

Установка модулей локально

Вы можете установить модуль локально, чтобы использовать его с выбранным *playbook*-ом. Это удобно, если вы не хотите делать модуль доступным для всех пользователей или не имеете прав администратора на сервере. Чтобы установить модуль локально, например, для использования с *playbook*-ом, расположенным в папке ./myplaybook, выполните следующие действия:

1. Скачайте из папки Modules по адресу https://github.com/rudecs/decsamo файлы decs_vm.py и decs_utility.py.

2. Создайте папку ./myplaybook/module_utils
3. Поместите в папку ./myplaybook/module_utils файл decs_utility.py
4. Создайте папку ./myplaybook/library
5. Скопируйте в папку ./myplaybook/library файл decs_vm.py. Если вы планируете использовать дополнительные модули, то также скопируйте файлы decs_vdc.py, decs_jwt.py, decs_gpu.py.

Данная структура папок является рекомендованной Ansible. Подробнее см. https://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html#directory-layout

Установка модулей для всех пользователей

Данный вариант требует наличия прав администратора. Чтобы установить модуль *decs_vm* на Ansible сервер для всех пользователей и/или *playbooks*, выполните следующие шаги:

1. Скачайте из папки Modules по адресу https://github.com/rudecs/decsamo файлы *decs_vm.py* и *decs_utility.py*.
2. Поместите файл *decs_vm.py* в папку /usr/share/ansible/plugins/modules/ на Ansible сервере. Проверьте, что права доступа к файлу допускают его выполнение. Если вы планируете использовать дополнительные модули, то также скопируйте файлы decs_vdc.py, decs_jwt.py, decs_gpu.py.
3. Поместите файл *decs_utility.py* в папку /usr/lib/python2.7/dist-packages/ansible/module_utils/ на Ansible сервере. Проверьте, что права доступа к файлу допускают его выполнение.

Параметры модуля decs_vm

Ниже в алфавитном порядке приведен полный список параметров для модуля *decs_vm*. Актуальную информацию по параметрам, которые поддерживает версия модуля, установленного на вашем Ansible-сервере, можно получить командой:

```
ansible-doc -t module decs_vm
```

Параметр	Тип, допустимые значения	Описание
annotation	(string)	Опциональное описание виртуального сервера. Этот параметр используется только в момент создания нового виртуального сервера и игнорируется при любых действиях над существующими серверами.
app_id	(string)	Идентификатор приложения, использующийся для подключения к контроллеру облачной платформы DECS в режиме authenticator: oauth2. Данный параметр является обязательным для указанного режима. Если параметр не задан в *playbook*, модуль *decs_vm* будет использовать значение переменной окружения *DECS_APP_ID*.
app_secret	(string)	Секретный ключ приложения, который используется для подключения к контроллеру облачной платформы DECS в режиме authenticator: oauth2. Данный параметр является обязательным для указанного режима. Так как он содержит секретную информацию, то его не рекомендуется задавать непосредственно в *playbook*. Если параметр не задан в *playbook*,

Параметр	Тип, допустимые значения	Описание
		то модуль *decs_vm* будет использовать значение переменной окружения *DECS_APP_SECRET*.
authenticator	Значения: legacy oauth2 jwt <- default	Режим аутентификации при подключении к контроллеру облачной платформы DECS.
boot_disk	(dict)	Конфигурация загрузочного диска виртуального сервера. Данный параметр является обязательным при создании нового виртуального сервера. Для существующих виртуальных серверов этот параметр является опциональным – с его помощью можно увеличить размер загрузочного диска. Параметр представляет собой словарь со следующими ключами: * (int) size – размер диска в GB. * (string) model – модель системы хранения данных, на базе которой будет создан данный загрузочный диск. Допустимые модели: ‘ovs’, ‘iscsi’. * (string) pool – пул дисковых ресурсов, из которого будет предоставлен данный загрузочный диск. Имена пулов могут меняться от одной модели СХД к другой. Если указанный пул не найден, модуль

Параметр	Тип, допустимые значения	Описание
		ожидает, что платформа создаст дисковый ресурс в “default” пуле, который всегда должен присутствовать. Обратите внимание, что в текущей реализации облачной инфраструктуры DECS используется только одна модель системы хранения данных типа ‘ovs’. Поэтому модуль *decs_vm* пока игнорирует параметры model и pool.
controller_url	(string)	URL контроллера, соответствующего экземпляру облачной платформы DECS, в рамках которого должен быть создан (или уже существует) данный виртуальный сервер. Данный параметр является обязательным.
cpu	(int)	Количество виртуальных CPU, выделенных виртуальному серверу. Параметр является обязательным при создании нового сервера, во всех других случаях он опциональный. Если указать его для уже существующего сервера, то будет выполнена попытка изменить количество CPU. Следует иметь ввиду, что *уменьшение* количества CPU у

Параметр	Тип, допустимые значения	Описание
		работающего сервера, как правило, потребует его перезагрузки.
datacenter	(string)	Целевой *datacenter* под управлением заданного DECS контроллера, где размещается данный виртуальный сервер. Этот параметр является обязательным при создании нового виртуального сервера, когда вместе с сервером также требуется создать виртуальный сетевой сегмент, в котором он будет расположен. При всех прочих операциях данный параметр игнорируется.
data_disks	(list of dict)	Список data-дисков (дополнительных дисков помимо загрузочного), которые надо создать и презентовать виртуальному серверу. Данный параметр является опциональным, используется только при создании нового виртуального сервера и игнорируется при любых других операциях. Параметр представляет собой словарь со следующими ключами: * (int) size – размер диска в GB. * (string) model – модель системы хранения данных, на базе которой будет создан данный data-диск.

Параметр	Тип, допустимые значения	Описание
		Допустимые модели: ‘ovs’, ‘iscsi’. * (string) pool – пул дисковых ресурсов, из которого будет предоставлен данный data-диск. Имена пулов могут меняться от одной модели СХД к другой. Если указанный пул не найден, модуль ожидает, что платформа создаст дисковый ресурс в “*default*” пуле, который всегда должен присутствовать. Обратите внимание, что в текущей реализации облачной инфраструктуры DECS используется только одна модель системы хранения данных типа ‘ovs’. Поэтому модуль *decs_vm* пока игнорирует параметры model и pool.
ext_network	Значения: present absent <- default	Данный опциональный параметр задает наличие/отсутствие прямого сетевого подключения данного виртуального сервера к внешнему сетевому сегменту. Технически данная операция заключается в выделении серверу публичного IP адреса и создании еще одного виртуального сетевого интерфейса. Если адрес выделен из сегмента, в котором нет DHCP сервера, то владелец сервера должен самостоятельно настроить

Параметр	Тип, допустимые значения	Описание
		IP адрес, маску подсети и шлюз по умолчанию на новой сетевом интерфейсе, выполнив необходимые действия на уровне гостевой ОС сервера.
ext_network_id	(int)	Данный опциональный параметр используется в паре с ext_network, чтобы задать идентификатор внешней сети, из которой следует брать внешний IP адрес. Если этот параметр опущен при ext_network: present, то внешний IP адрес будет выбран из внешнего сетевого сегмента по умолчанию.
id	(int)	Уникальный цифровой идентификатор виртуального сервера внутри платформы. Этот параметр используется как один из методов идентификации существующего сервера (альтернатива – по комбинации name, vdc_name и tenant) и игнорируется при создании нового сервера, так как для нового сервера облачная платформа назначает этот идентификатор автоматически. Если при вызове модуля *decs_vm* существующий виртуальный сервер идентифицируется по своему id, то

Параметр	Тип, допустимые значения	Описание
		параметры tenant, vdc_name и vdc_id игнорируются.
image_id	(int)	Уникальный цифровой идентификатор образа ОС, на базе которого следует создать виртуальный сервер. При создании нового виртуального сервера требуется задать этот параметр или параметр image_name. При любых других операциях данные параметры игнорируются. Если заданы оба этих параметра (image_id и image_name), то image_name игнорируется. Для получения image_id по image_name можно использовать модуль *decs_osimage*.
image_name	(string)	Название образа ОС, на базе которого следует создать виртуальный сервер. При создании нового виртуального сервера требуется задать этот параметр или параметр image_id. При любых других операциях данные параметры игнорируются. Параметр image_name должен задаваться в точном соответствии с тем, как назван нужный образ ОС в облачной инфраструктуре (с

Параметр	Тип, допустимые значения	Описание
		соблюдением заглавных и строчных символов, а также пробелов).
jwt	(string)	JSON Web Token (JWT), который будет использоваться для подключения к контроллеру облачной платформы DECS в режиме authenticator: jwt Данный параметр является обязательным для указанного режима. Так как он содержит потенциально секретную информацию, а сам JWT, как правило, имеет ограниченное время жизни, то его не рекомендуется задавать непосредственно в *playbook*. Если этот параметр не определен в *playbook*, то модуль *decs_vm* будет использовать значение переменной окружения *DECS_JWT*.
name	(string)	Название виртуального сервера. Чтобы модуль *decs_vm* мог управлять сервером по его названию, также необходимо задать комбинацию tenant и vdc_name или идентификатор vdc_id. Параметр должен задаваться в точном соответствии с тем, как назван сервер в облачной инфраструктуре (с соблюдением

Параметр	Тип, допустимые значения	Описание
		заглавных и строчных символов, а также пробелов). Если для существующего виртуального сервера указаны и name, и id, то параметр name игнорируется и идентификация сервера выполняется по его id.
oauth2_url	(string)	URL авторизационного сервера, работающего по протоколу *Oauth2*, который должен использоваться в режиме authenticator: oauth2. Данный параметр является обязательным для указанного режима. Если параметр не задан в *playbook*, модуль *decs_vm* будет использовать значение переменной окружения *DECS_OAUTH2_URL*.
password	(string)	Пароль для подключения к контроллеру облачной инфраструктуры DECS в режиме authenticator: legacy. Данный параметр является обязательным для указанного режима. Так как он содержит секретную информацию, то его не рекомендуется задавать в *playbook*. Если параметр не задан в *playbook*, то модуль *decs_vm* будет

Параметр	Тип, допустимые значения	Описание
		использовать значение переменной окружения *DECS_PASSWORD*.
port_forwards	(list of dict)	Список правил настройки port forwarding для виртуального сервера. Каждое правило в списке это словарь со следующими ключами: * (int) ext_port – номер внешнего сетевого порта; * (int) int_port – номер внутреннего сетевого порта; * (str) proto – наименование протокола. Доступные протоколы: ‘tcp’, ‘udp’. Модуль применяет следующую логику: * Если какие-то из уже настроенных в настоящий момент правил в целевом состоянии отсутствуют, они будут удалены. * Если какие-то из уже настроенных правил присутствуют в целевом состоянии, они будут оставлены без изменений. * Если какие-то из правил, перечисленных в целевом состоянии, отсутствуют в текущем состоянии, то они будут созданы.
ram	(int)	Объем оперативной памяти (RAM) в MB, выделенной данному

Параметр	Тип, допустимые значения	Описание
		виртуальному серверу. Параметр является обязательным при создании нового сервера. Если указать его для уже существующего сервера, то будет выполнена попытка изменить объем выделенной серверу памяти. Следует иметь ввиду, что уменьшение объема памяти работающего сервера в большинстве случаев потребует его перезагрузки.
ssh_key	(string)	Открытая часть SSH ключа, который необходимо авторизовать на создаваемом виртуальном сервере для пользователя, заданного параметром ssh_key_user. Данный параметр применим только для Linux серверов, является опциональным, используется только при создании нового сервера и игнорируется при других операциях.
ssh_key_user	(string)	Имя пользователя на уровне гостевой ОС (только для Linux серверов) для которого авторизуется SSH ключ, заданный параметром ssh_key. Данный параметр является обязательным, если задан ssh_key, используется только при создании

Параметр	Тип, допустимые значения	Описание
		нового сервера и игнорируется при других операциях.
state	Значения: present <- default absent poweredon poweredoff halted paused check	Целевое состояние виртуального сервера на выходе из модуля *decs_vm*. Значение 'halted' - синоним к 'poweredoff'. Значение 'check' вызывает модуль в *read-only* режиме и считывает характеристики существующего виртуального сервера.
tags	(string)	Строка, содержащая набор текстовых меток, которые надлежит присвоить данному виртуальному серверу. Данные текстовые метки представляют собой произвольный текст, который можно использовать для группировки и индексирования виртуальных серверов во внешних приложениях. Параметр является опциональным.
tenant	(string)	Имя подписчика, которому будет принадлежать новый виртуальный сервер (или уже принадлежит существующий). Параметр должен задаваться в точном соответствии с тем, как назван нужный подписчик в

Параметр	Тип, допустимые значения	Описание
		облачной инфраструктуре (с соблюдением заглавных и строчных символов, а также пробелов). Этот параметр является опциональным и используется в сценариях, когда вместе с новым виртуальным сервером создается новый виртуальный сетевой сегмент (*VDC*), в котором будет размещаться данный сервер, или когда уже существующий целевой сегмент задается комбинацией tenant и vdc_name.
user	(string)	Имя пользователя, непосредственно зарегистрированного на контроллере облачной инфраструктуры DECS, которое используется для подключения к контроллеру в режиме authenticator: legacy. Данный параметр является обязательным для указанного режима. Если параметр не задан в *playbook*, модуль *decs_vm* будет использовать значение переменной окружения *DECS_USER*.
vdc_id	(int)	Уникальный цифровой идентификатор уже существующего виртуального сетевого сегмента

Параметр	Тип, допустимые значения	Описание
		(VDC), где будет создан новый или находится уже существующий виртуальный сервер. Данный параметр является одним из методов идентификации существующего VDC (альтернативой является задание комбинации tenant и vdc_name).
vdc_name	(string)	Имя виртуального сетевого сегмента (VDC), где будет создан новый или находится уже существующий виртуальный сервер. Данный параметр является одним из методов идентификации существующего VDC, когда задается пара tenant и vdc_name (альтернативой является задание vdc_id, однако такой метод применим только для уже существующих VDC). Параметр должен задаваться в точном соответствии с тем, как назван нужный сегмент в облачной инфраструктуре (с соблюдением заглавных и строчных символов, а также пробелов). Если заданы и vdc_id, и vdc_name, то параметр vdc_name игнорируется.

Параметр	Тип, допустимые значения	Описание
verify_ssl	(bool) True <- default False	Позволяет отключить проверку SSL сертификатов при выполнении API вызовов в адрес контроллера облачной инфраструктуры, например, при работе с изолированной облачной инфраструктурой, использующей самоподписанные сертификаты. Применяйте данный параметр с осторожностью, предпочтительно в защищенных средах.
workflow_callback	(string)	URL, по которому вышестоящее приложение (например, пользовательский портал или оркестратор верхнего уровня, инициирующий запуск Ansible playbook) ожидает API вызова, в параметрах которого модуль *desc_vm* будет оперативно передавать информацию о своем статусе и текущей фазе исполнения. Данный параметр является опциональным. Функциональность callbacks в текущей версии модуля *decs_vm* не реализована.
workflow_context	(string)	Контекстная информация, которая будет содержаться в параметрах API вызова, адресованного к workflow_callback URL.

Параметр	Тип, допустимые значения	Описание
		Данная информация призвана однозначно идентифицировать задачу, выполняемую модулем в настоящий момент, чтобы оркестратор верхнего уровня мог сопоставить получаемые через вызов workflow_callback данные со своим внутренним состоянием и отслеживать инициированные им задачи. Параметр является опциональным и имеет значение только при условии, что также задан workflow_callback. Функциональность callbacks в текущей версии модуля *decs_vm* не реализована.

Возвращаемые значения

Модуль *decs_vm* возвращает информацию о виртуальном сервере в виде словаря vm_facts со следующими ключами:

Ключ	Тип данных	Описание
ext_gateway	string	IP адрес шлюза по умолчанию (default gateway) для внешнего прямого IP адреса, назначенного данному серверу. Если серверу не назначен прямой внешний IP адрес, по данному ключу возвращается пустая строка.

Ключ	Тип данных	Описание
ext_ip	string	Прямой внешний IP адрес, назначенный данному виртуальному серверу. Если серверу не назначен прямой внешний IP адрес, по данному ключу возвращается пустая строка.
ext_mac	string	MAC адрес виртуального сетевого интерфейса, подключенного к прямому внешнему (публичному) IP адресу. Если серверу не назначен прямой внешний IP адрес, по данному ключу возвращается пустая строка.
ext_netmask	int	Маска подсети для прямого внешнего IP адреса. Если серверу не назначен прямой внешний IP адрес, по данному ключу возвращается пустая строка.
id	int	Уникальный цифровой идентификатор виртуального сервера в платформе DECS.
int_ip	string	IP адрес виртуального сервера внутри виртуального сетевого сегмента (vdc_id), в котором размещен данный сервер.
name	string	Имя виртуального сервера. Имя уникально только в рамках одного и того же виртуального сетевого сегмента.
password	string	Пароль системного пользователя по умолчанию.

Ключ	Тип данных	Описание
state	string	Состояние виртуального сервера.
usrename	string	Имя системного пользователя по умолчанию.
vdc_ext_ip	string	Внешний (публичный) IP адрес виртуального сетевого сегмента, к которому принадлежит данный сервер.
vdc_id	int	Уникальный цифровой идентификатор виртуального сетевого сегмента (*VDC*), к которому принадлежит данный сервер.
vdc_name	string	Имя виртуального сетевого сегмента, к которому принадлежит данный сервер. Имя уникально только в рамках одного и того же подписчика (*tenant*).

Типовые сценарии и примеры использования модуля decs_vm

Как создать новый сервер в новом виртуальном сетевом сегменте

Описанный ниже сценарий удобен в тех случаях, когда нужно создать новый виртуальный сервер, но не хочется утруждать себя проверками, существует ли целевой сетевой сегмент. Если целевой сетевой сегмент не существует, то он будет создан до того, как начнется создание виртуального сервера. Если же целевой сегмент уже существует, то модуль извлечет из контроллера необходимую информацию о сегменте и поместит в этот сегмент новую виртуальную машину.

Дано:

1. DECS контроллер находится по адресу *cloud.digitalenergy.online*
2. Пользователь *abstract_user* имеет доступ к подписчику *MyMainTenant* в указанном DECS контроллере.

3. На данном контроллере определены:
    - Дата-центр с именем «site1».
    - Образ ОС с именем «Ubuntu 16.04».
4. Oauth2 провайдер находится по адресу *sso.decs.online*
5. Пользователь _abstract_use-r идентифицируется в Oauth2 провайдере по известным *Application ID* & *Application Secret*, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача:

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Создать новый виртуальный сетевой сегмент, принадлежащий подписчику *MyMainTenant*, со следующими характеристиками:
    - Имя сегмента – *MyFirstVDC* (предполагается, что у данного подписчика еще нет сегмента с таким именем).
    - Расположение – в дата-центре *site1*.
3. Поместить в только что созданный виртуальный сетевой сегмент новый виртуальный сервер с именем *NewVM01*. Сервер должен иметь следующие характеристики:
    - CPU – 2.
    - RAM – 4096 МБ.
    - Загрузочный диск – 10 ГБ.
    - Образ ОС – “Ubuntu 16.04”.
    - Описание – “My new VM created and managed with DECS VM module”
4. Сохранить информацию о созданном виртуальном сервере в переменной vm01_specs.

```
---
- hosts: ansible_master
  tasks:
  - name: create new VM in a new private network segment
```

```
  decs_vm:
    authenticator: oauth2
    app_id: "{{ my_app_id }}"
    app_secret: "{{ my_app_secret }}"
    oauth2_url: "https://sso.decs.online"
    controller_url: "https://cloud.digitalenergy.online"
    name: NewVM01
    state: present
    cpu: 2
    ram: 4096
    boot_disk:
      size: 10
    image_name: "Ubuntu 16.04"
    tenant: "MyMainTenant"
    datacenter: site1
    vdc_name: "MyFirstVDC"
    annotation: "My new VM created and managed with DECS VM module"
  delegate_to: localhost
  register: vm01_specs
...
```

Следует иметь ввиду, что для создания нового сетевого сегмента обязательны следующие параметры: tenant, datacenter, vdc_name. Если хотя бы один из этих параметров не указан, а сегмент не существует, модуль вернет ошибку.

Если достоверно известно, что целевой сегмент существует, то для его идентификации будет достаточно tenant и vdc_name (либо просто указать его уникальный идентификатор vdc_id – см. пример в разделе «Как создать новый сервер в существующем виртуальном сетевом сегменте»).

Как создать новый сервер в существующем виртуальном сетевом сегменте

Данный сценарий удобен для быстрого создания нового виртуального сервера или получения информации об уже существующем виртуальном сервере (подробнее об этом см. в разделе «Как получить и использовать информацию о созданном сервере внутри playbook-а»), когда известен

цифровой идентификатор сетевого сегмента, где должен располагаться данный сервер.

Дано:

1. DECS контроллер находится по адресу *cloud.digitalenergy.online*
2. На данном контроллере:
    - Имеется образ ОС с именем «Ubuntu 16.04».
    - Имеется сетевой сегмент (VDC) с идентификатором 685, принадлежащий подписчику *MyMainTenant*.
3. Пользователь abstract_user имеет доступ к подписчику *MyMainTenant* в указанном DECS контроллере.
4. Oauth2 провайдер находится по адресу *sso.decs.online* и отвечает по порту 4443
5. Пользователь abstract_user идентифицируется в Oauth2 провайдере по известным *Application ID* & *Application Secret*, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача:

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Создать новый виртуальный сервер с именем *NewVM02*. Сервер должен иметь следующие характеристики:
    - Располагаться в сетевом сегменте с идентификатором 685 (предполагается, что данный сетевой сегмент уже существует и принадлежит подписчику *MyMainTenant*)
    - CPU – 1.
    - RAM – 2048 МБ.
    - Загрузочный диск – 20 ГБ.
    - Образ ОС – “Ubuntu 16.04”.
    - Описание – “Another VM created and managed with DECS VM module”.

3. Сохранить информацию о созданном виртуальном сервере в переменной vm02_specs.

```
---
- hosts: ansible_master
  tasks:
  - name: create new VM in the pre-existing private network segment
    decs_vm:
      authenticator: oauth2
      app_id: "{{ my_app_id }}"
      app_secret: "{{ my_app_secret }}"
      oauth2_url: "https://sso.decs.online:4443"
      controller_url: "https://cloud.digitalenergy.online"
      name: NewVM02
      state: present
      cpu: 1
      ram: 2048
      boot_disk:
        size: 20
      image_name: "Ubuntu 16.04"
      vdc_id: 685
      annotation: "My new VM created and managed with DECS VM module"
    delegate_to: localhost
    register: vm02_specs
...
```

Как создать новый сервер и авторизовать SSH ключ заданному пользователю

Данный сценарий иллюстрирует пример создания нового виртуального сервера в существующем виртуальном сетевом сегменте и авторизацию SSH ключа заданному пользователю этого сервера. Как следствие, после создания сервера данный пользователь сможет авторизоваться в гостевую ОС по SSH ключу.

Типовое применение данного сценария - авторизация SSH-ключа, под которым работает Ansible, пользователю *root* на создаваемом виртуальном сервере, чтобы впоследствии Ansible мог управлять настройками установленных на этом сервере гостевой ОС и прикладного ПО.

Дано:

1. DECS контроллер находится по адресу *cloud.digitalenergy.online*
2. На данном контроллере:
    - Имеется образ ОС с именем «Ubuntu 16.04».
    - Имеется сетевой сегмент (VDC) с идентификатором 314, принадлежащий подписчику *MyMainTenant*.
3. Пользователь *abstract_user* имеет доступ к подписчику *MyMainTenant* в указанном DECS контроллере.
4. Oauth2 провайдер находится по адресу *sso.decs.online* и отвечает по порту 4443
5. Пользователь *abstract_user* идентифицируется в Oauth2 провайдере по известным *Application ID* & *Application Secret*, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача:

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Создать новый виртуальный сервер с именем *NewVM03*. Сервер должен иметь следующие характеристики:
    - Располагаться в сетевом сегменте с идентификатором 314 (предполагается, что данный сетевой сегмент уже существует и принадлежит подписчику MyMainTenant)
    - CPU – 4.
    - RAM – 8192 МБ.
    - Загрузочный диск – 10 ГБ.
    - Образ ОС – “Ubuntu 16.04”.
    - Описание – “New VM with authorized SSH key”.
3. Авторизовать публичную часть SSH ключа, хранящуюся в переменной my_ssh_key пользователю *root* на созданном виртуальном сервере.
4. Сохранить информацию о созданном виртуальном сервере в переменной vm03_specs.

```
---
- hosts: ansible_master
  tasks:
  - name: create new VM and authorize supplied SSH key to root
    decs_vm:
      authenticator: oauth2
      app_id: "{{ my_app_id }}"
      app_secret: "{{ my_app_secret }}"
      oauth2_url: "https://sso.decs.online:4443"
      controller_url: "https://cloud.digitalenergy.online"
      name: NewVM03
      state: present
      cpu: 4
      ram: 8192
      boot_disk:
        size: 10
      image_name: "Ubuntu 16.04"
      vdc_id: 314
      ssh_key: "{{ my_ssh_key }}"
      ssh_key_user: root
      annotation: " New VM with authorized SSH key "
    delegate_to: localhost
    register: vm03_specs
...
```

Как получить и использовать информацию о созданном сервере внутри playbook-а

После успешного выполнения *task*, вызывающего модуль *decs_vm*, базовая информация о полученном виртуальном сервере может быть записана в переменную с помощью директивы register (см. последнюю строку в примерах из разделов, посвященных созданию нового виртуального сервера).

Обратите внимание, что даже если сервер с заданными в *task* характеристиками существовал на момент запуска модуля *decs_vm*, и никаких фактических изменений в его конфигурации не требовалось, модуль все равно возвращает информацию о текущем состоянии виртуального

сервера. Содержание и формат этой информации подробно описаны в разделе «Возвращаемые значения модуля **decs_vm**».

Дано:

1. Первый task в playbook успешно создал новый виртуальный сервер и с помощью директивы register информация о нем сохранена в переменной vm_specs.
2. На сервере настроена трансляция сетевых портов следующим образом:
    - Внешний порт – 21022.
    - Внутренний порт – 22 (протокол SSH).
    - Сетевой протокол – tcp.

Задача:

1. Внести запись о только что созданном сервере в *in-memory inventory* и запустить на нем *task*, который установит пакет nginx.

```
---
- hosts: ansible_master
 tasks:
 - name: obtain VM
   decs_vm:
   <<<детали опущены>>>
   register: vm_specs

 - name: add the VM to the in-memory inventory into group just_created
   add_host:
     groups: just_created
     hostname: "{{ vm_specs.vm_facts.name }}"
     ansible_host: "{{ vm_specs.vm_facts.vdc_ext_ip }}"
     ansible_port: 21022
     ansible_user: "{{ vm_specs.vm_facts.username }}"
     ansible_password: "{{ vm_specs.vm_facts.password }}"

- hosts: just_created
```

```
 tasks:
 - name: install the latest nginx and update package cache on the way
   apt:
      name: nginx
      state: latest
      update_cache: yes
...
```

Как управлять правилами трансляции сетевых портов сервера

Дано:

- DECS контроллер находится по адресу *cloud.digitalenergy.online*
- На данном контроллере имеется образ ОС с именем «Ubuntu 16.04».
- Пользователь *abstract_user* имеет доступ к подписчику *MyMainTenant* в указанном DECS контроллере.
- Oauth2 провайдер находится по адресу *sso.decs.online*
- Пользователь *abstract_user* идентифицируется в Oauth2 провайдере по известным *Application ID* & _Application Secret-, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача №1 – создание/добавление правил

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Создать новый виртуальный сервер с именем AppSrv01. Сервер должен иметь следующие характеристики:
    - Располагаться в сетевом сегменте с именем *MyApps* (предполагается, что данный сетевой сегмент уже существует и принадлежит подписчику *MyMainTenant*)
    - CPU – 4.
    - RAM – 8192 МБ.
    - Загрузочный диск – 20 ГБ.
    - Образ ОС – “Ubuntu 16.04”.
    - Настройки трансляции сетевых портов:

- Внешний порт 21022 -> внутренний порт 22, протокол tcp.
- Внешний порт 443 -> внутренний порт 443, протокол tcp.

3. Сохранить информацию о созданном виртуальном сервере в переменной appsrv01.

```
---
- hosts: ansible_master
  tasks:
  - name: create new VM with port forwards
    decs_vm:
      authenticator: oauth2
      app_id: "{{ my_app_id }}"
      app_secret: "{{ my_app_secret }}"
      oauth2_url: "https://sso.decs.online"
      controller_url: "https://cloud.digitalenergy.online"
      name: AppSrv01
      state: present
      cpu: 4
      ram: 8192
      boot_disk:
        size: 20
      image_name: "Ubuntu 16.04"
      port_forwards:
        - ext_port: 21022
          int_port: 22
          proto: tcp
        - ext_port: 443
          int_port: 443
          proto: tcp
      tenant: "MyMainTenant"
      vdc_name: "MyApps"
    delegate_to: localhost
    register: appsrv01
...
```

Задача №2 – изменение/удаление правил

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Изменить настройки трансляции сетевых портов у ранее созданного виртуального сервера с именем *AppSrv01* следующим образом:
    - Добавить новое правило «Внешний порт 80 -> внутренний порт 80, протокол tcp».
    - Правило «Внешний порт 21022 -> внутренний порт 22, протокол tcp» оставить без изменений.
    - Правило «Внешний порт 443 -> внутренний порт 443, протокол tcp» удалить.
3. Прочие характеристики сервера оставить без изменений.
4. Сохранить информацию о созданном виртуальном сервере в переменной appsrv01.

```
---
- hosts: ansible_master
 tasks:
 - name: create new VM with port forwards
   decs_vm:
    authenticator: oauth2
    app_id: "{{ my_app_id }}"
    app_secret: "{{ my_app_secret }}"
    oauth2_url: "https://sso.decs.online"
    controller_url: "https://cloud.digitalenergy.online"
    name: AppSrv01
    state: present
    port_forwards:
     - ext_port: 80
       int_port: 80
       proto: tcp
     - ext_port: 21022
       int_port: 22
       proto: tcp
    tenant: "MyMainTenant"
```

```
    vdc_name: "MyApps"
  delegate_to: localhost
  register: appsrv01
...
```

Как следует из логики вышеприведенного *playbook*, при вызове модуля *decs_vm* для уже существующего виртуального сервера с настроенными ранее правилами трансляции сетевых адресов мы всякий раз задаем целевое состояние правил. Таким образом:

- Если какие-то из уже настроенных в настоящий момент правил в целевом состоянии отсутствуют, они будут удалены.
- Если какие-то из уже настроенных правил присутствуют в целевом состоянии, они будут оставлены без изменений.
- Если какие-то из правил, перечисленных в целевом состоянии, отсутствуют в текущем состоянии, то они будут созданы.

Как управлять подключением / отключением внешнего IP адреса

Дано:

1. DECS контроллер находится по адресу cloud.digitalenergy.online
2. На данном контроллере имеется образ ОС с именем «Ubuntu 16.04».
3. Пользователь abstract_user имеет доступ к подписчику MyMainTenant в указанном DECS контроллере.
4. Oauth2 провайдер находится по адресу sso.decs.online
5. Пользователь abstract_user идентифицируется в Oauth2 провайдере по известным Application ID & Application Secret, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача №1 – подключить внешний IP адрес к серверу:

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Создать новый виртуальный сервер с именем *AppSrv02*. Сервер должен иметь следующие характеристики:

- Располагаться в сетевом сегменте с именем *MyApps* (предполагается, что данный сетевой сегмент уже существует и принадлежит подписчику *MyMainTenant*).
- CPU – 1.
- RAM – 4096 МБ.
- Загрузочный диск – 50 ГБ.
- Образ ОС – “Ubuntu 16.04”.
- Настройки трансляции сетевых портов:
  - Внешний порт 21022 -> внутренний порт 22, протокол tcp
- Выделить серверу внешний IP адрес из сетевого сегмента с идентификатором 42.

3. Сохранить информацию о созданном виртуальном сервере в переменной appsrv02.

```
---
- hosts: ansible_master
 tasks:
 - name: create new VM with port forwards and direct external IP
   decs_vm:
     authenticator: oauth2
     app_id: "{{ my_app_id }}"
     app_secret: "{{ my_app_secret }}"
     oauth2_url: "https://sso.decs.online"
     controller_url: "https://cloud.digitalenergy.online"
     name: AppSrv02
     state: present
     cpu: 1
     ram: 4096
     boot_disk:
       size: 50
     image_name: "Ubuntu 16.04"
     ext_network: present
     ext_network_id: 42
```

```
    port_forwards:
     - ext_port: 21022
       int_port: 22
       proto: tcp
    tenant: "MyMainTenant"
    vdc_name: "MyApps"
  delegate_to: localhost
  register: appsrv02
...
```

После успешного выполнения данного *task* получить сетевые характеристики сервера из переменной appsrv02 можно следующим образом:

- Внешний прямой IP адрес – appsrv02.vm_facts.ext_ip
- Шлюз по умолчанию – appsrv02.vm_facts.ext_gateway
- Маска подсети – appsrv02.vm_facts.ext_netmask
- MAC-адрес внешнего сетевого интерфейса – appsrv02.vm_facts.ext_mac

Как включать / выключать существующий сервер

Дано:

1. DECS контроллер находится по адресу *cloud.digitalenergy.online*
2. Пользователь *abstract_user* имеет доступ к подписчику *MyMainTenant* в указанном DECS контроллере.
3. Подписчик *MyMainTenant* владеет виртуальным сетевым сегментом с именем *MyApps*, в котором имеется виртуальный сервер с именем *AppSrv01*.
4. Oauth2 провайдер находится по адресу *sso.decs.online*
5. Пользователь abstract_user идентифицируется в Oauth2 провайдере по известным *Application ID* & *Application Secret*, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача №1 – выключить виртуальный сервер:

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Выключить виртуальный сервер *AppSrv01*.

3. Прочие характеристики виртуального сервера *AppSrv01* должны остаться без изменений.

```
---
- hosts: ansible_master
  tasks:
  - name: poweroff existing VM
    decs_vm:
      authenticator: oauth2
      app_id: "{{ my_app_id }}"
      app_secret: "{{ my_app_secret }}"
      oauth2_url: "https://sso.decs.online"
      controller_url: "https://cloud.digitalenergy.online"
      name: AppSrv01
      state: poweredoff
      tenant: "MyMainTenant"
      vdc_name: "MyApps"
    delegate_to: localhost
...
```

Аналогично, чтобы запустить остановленный виртуальный сервер, параметру state следует присвоить значение *poweredon*. Работающий сервер можно перевести в состояние «приостановлен», выставив параметр state в *paused*.

Задача №2 – включить виртуальный сервер:

1. Авторизоваться в DECS контроллер в режиме oauth2.
2. Включить ранее остановленный виртуальный сервер AppSrv01.
3. В процессе включения:
    - Гарантировать, что сервер не будет иметь прямого выделенного IP адреса.

```
---
- hosts: ansible_master
  tasks:
  - name: poweron existing VM, resize it and remove external IP (if any)
    decs_vm:
      authenticator: oauth2
```

```
    app_id: "{{ my_app_id }}"
    app_secret: "{{ my_app_secret }}"
    oauth2_url: "https://sso.decs.online"
    controller_url: "https://cloud.digitalenergy.online"
    name: AppSrv01
    ext_network: absent
    state: poweredon
    tenant: "MyMainTenant"
    vdc_name: "MyApps"
  delegate_to: localhost
...
```

Следует иметь ввиду, что определенные комбинации целевого и текущего состояния сервера не подразумевают изменения «параметров питания» сервера (например, целевое состояние *present*, а текущее – *poweredoff*). В таких ситуациях изменение параметров питания не производится, но может быть выполнена настройка сетевых параметров (правила трансляции сетевых портов, подключение/отключение прямого IP адреса) и изменение объема выделенных серверу ресурсов (CPU, RAM, размер загрузочного диска).

7.8 Как изменить CPU, RAM и размер загрузочного диска существующего сервера

Дано:

1. DECS контроллер находится по адресу *cloud.digitalenergy.online*
2. Пользователь *abstract_user* имеет доступ к подписчику *MyMainTenant* в указанном DECS контроллере.
3. Подписчик *MyMainTenant* владеет виртуальным сетевым сегментом с именем *MyApps*, в котором имеется виртуальный сервер с именем *AppSrv02*. Текущие характеристики сервера:
    - CPU – 2.
    - RAM – 4096 МБ.
    - Загрузочный диск – 50 ГБ.
4. Oauth2 провайдер находится по адресу *sso.decs.online*

5. Пользователь *abstract_user* идентифицируется в Oauth2 провайдере по известным *Application ID* & *Application Secret*, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача:

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Задать новые характеристики для виртуального сервера *AppSrv02*:
    - CPU – 4.
    - RAM – 8192 МБ.
    - Загрузочный диск – 80 ГБ.

```
---
- hosts: ansible_master
  tasks:
  - name: poweron existing VM, resize it and remove external IP (if any)
    decs_vm:
      authenticator: oauth2
      app_id: "{{ my_app_id }}"
      app_secret: "{{ my_app_secret }}"
      oauth2_url: "https://sso.decs.online"
      controller_url: "https://cloud.digitalenergy.online"
      name: AppSrv02
      cpu: 4
      ram: 8192
      bootdisk:
        size: 100
      state: present
      tenant: "MyMainTenant"
      vdc_name: "MyApps"
    delegate_to: localhost
...
```

Необходимо отметить, что одновременно с изменением выделенных CPU, RAM и размера загрузочного диска сервера можно перенастроить правила трансляции сетевых портов (см. «Как управлять правилами трансляции

сетевых портов сервера») и подключить/отключить прямой внешний IP адрес (см. «Как управлять подключением / отключением внешнего IP адреса»).

7.9 Как удалить существующий сервер

Дано:

1. DECS контроллер находится по адресу *cloud.digitalenergy.online*
2. Пользователь *abstract_user* имеет доступ к подписчику *MyMainTenant* в указанном DECS контроллере.
3. Подписчик *MyMainTenant* владеет виртуальным сетевым сегментом с именем *MyApps*, в котором имеется виртуальный сервер с именем *AppSrv01*.
4. Oauth2 провайдер находится по адресу *sso.decs.online*
5. Пользователь *abstract_user* идентифицируется в Oauth2 провайдере по известным *Application ID* & *Application Secret*, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача:

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Удалить виртуальный сервер *AppSrv01*.

```
---
- hosts: ansible_master
  tasks:
  - name: delete existing VM
    decs_vm:
      authenticator: oauth2
      app_id: "{{ my_app_id }}"
      app_secret: "{{ my_app_secret }}"
      oauth2_url: "https://sso.decs.online"
      controller_url: "https://cloud.digitalenergy.online"
      name: AppSrv01
      state: absent
      tenant: "MyMainTenant"
      vdc_name: "MyApps"
    delegate_to: localhost
```

```
...
```

Следует иметь в виду, что данная операция необратима и удаляет все дисковые ресурсы, которые были ассоциированы с данным сервером (загрузочный диск и data-диски).

7.10 Как получить информацию об уже существующем виртуальном сервере

Описанный ниже сценарий полезен для тех случаев, когда нужно получить информацию о ранее созданных в облачной инфраструктуре виртуальных серверах. Например, если серверы были созданы другим пользователем или с использованием отличного от Ansible средства оркестрации.

Дано:

1. DECS контроллер находится по адресу *cloud.digitalenergy.online*
2. Пользователь *abstract_user* имеет доступ к подписчику *MyMainTenant* в указанном DECS контроллере. В рамках этого подписчика есть виртуальный сегмент *Project1*, в которм имеется виртуальный сервер *Srv01*
3. Oauth2 провайдер находится по адресу *sso.decs.online*
4. Пользователь _abstract_user идентифицируется в Oauth2 провайдере по известным *Application ID* & *Application Secret*, которые занесены в переменные my_app_id и my_app_secret соответственно.

Задача:

1. Авторизоваться в DECS контроллер в режиме *oauth2*.
2. Получить информацию о виртуальном сервере *Srv01*, находящемся в сегменте *Project1*, владельцем которого является подписчик *MyMainTenant*.
3. Сохранить полученную информацию об этом виртуальном сервере в переменной srv01_specs.

```
---
- hosts: ansible_master
  tasks:
  - name: check if server exists and read in its specifications
```

```
  decs_vm:
    authenticator: oauth2
    app_id: "{{ my_app_id }}"
    app_secret: "{{ my_app_secret }}"
    oauth2_url: "https://sso.decs.online"
    controller_url: "https://cloud.digitalenergy.online"
    name: Srv01
    state: check
    tenant: "MyMainTenant"
    vdc_name: "Project1"
  delegate_to: localhost
  register: srv01_specs
...
```

Удобство такого метода заключается в том, что если просто требуется получить информацию о заведомо существующем виртуальном сервере, то нет необходимости задавать все его конфигурационные параметры. Достаточно знать его имя и принадлежность к подписчику и виртуальному сетевому сегменту.

Примеры различных режимов авторизации

Как выглядит task для разных режимов авторизации

В разделе «Способы авторизации» описаны доступные в рамках облачной платформы DECS режимы авторизации. В данном разделе приведены примеры *task* для каждого из этих режимов.

Также обратите внимание на простейший способ хранить авторизационную информацию отдельно от *playbook*-а.

Режим oauth2 с использованием Application ID & Application Secret

Данный режим задается установкой параметра authenticator: oauth2

В этом режиме необходимо указать параметры app_id и app_secret (либо задать соответствующие значения в переменных окружения *DECS_APP_ID* и *DECS_APP_SECRET*).

Чтобы воспользоваться этим режимом, вы должны заранее создать *Application ID* & *Application Secret* под своей учетной записью на OAuth2 сервере, который используется для авторизации в вашем DECS контроллере.

Вы можете сделать это самостоятельно в соответствующем разделе личного кабинета на OAuth2 сервере.

Помимо вышеуказанных параметров также необходимо задать oauth2_url и controller_url. В итоге *task* будет выглядеть следующим образом:

```
---
- hosts: ansible_master
  tasks:
  - name: manage VM in DECS cloud
    decs_vm:
      authenticator: oauth2
      app_id: "{{ my_app_id }}"
      app_secret: "{{ my_app_secret }}"
      oauth2_url: "https://sso.decs.online"
      controller_url: "https://cloud.digitalenergy.online"
    <<<дальнейшие детали опущены>>>
...
```

Режим jwt с использованием заранее полученного JSON Web Token (JWT)

Данный режим задается установкой параметра authenticator: jwt

Режим *jwt* может оказаться полезным, когда вы планируете создавать множество виртуальных машин (например, в асинхнронном режиме вызова модуля *decs_vm*), так как дает экономию по количеству API вызовов в адрес OAuth2 провайдера и контроллера облачной инфраструктуры.

Чтобы воспользоваться этим режимом вам сначала потребуется получить действительный JWT, например, от OAuth2 сервера, который используется для авторизации в вашем DECS контроллере. Для получения JWT можно воспользоваться вспомогательным модулем *decs_jwt*.

Помимо задания параметра jwt (либо сохранения JWT в переменной окружения *DECS_JWT*) также необходимо задать oauth2_url и controller_url. В итоге *task* будет выглядеть следующим образом:

```
---
- hosts: ansible_master
  tasks:
```

```
 - name: manage VM in DECS cloud
   decs_vm:
     authenticator: jwt
     jwt: "{{ my_jwt }}"
     oauth2_url: "https://sso.decs.online"
     controller_url: "https://cloud.digitalenergy.online"
   <<<дальнейшие детали опущены>>>
...
```

Режим legacy с использованием имени пользователя и пароля

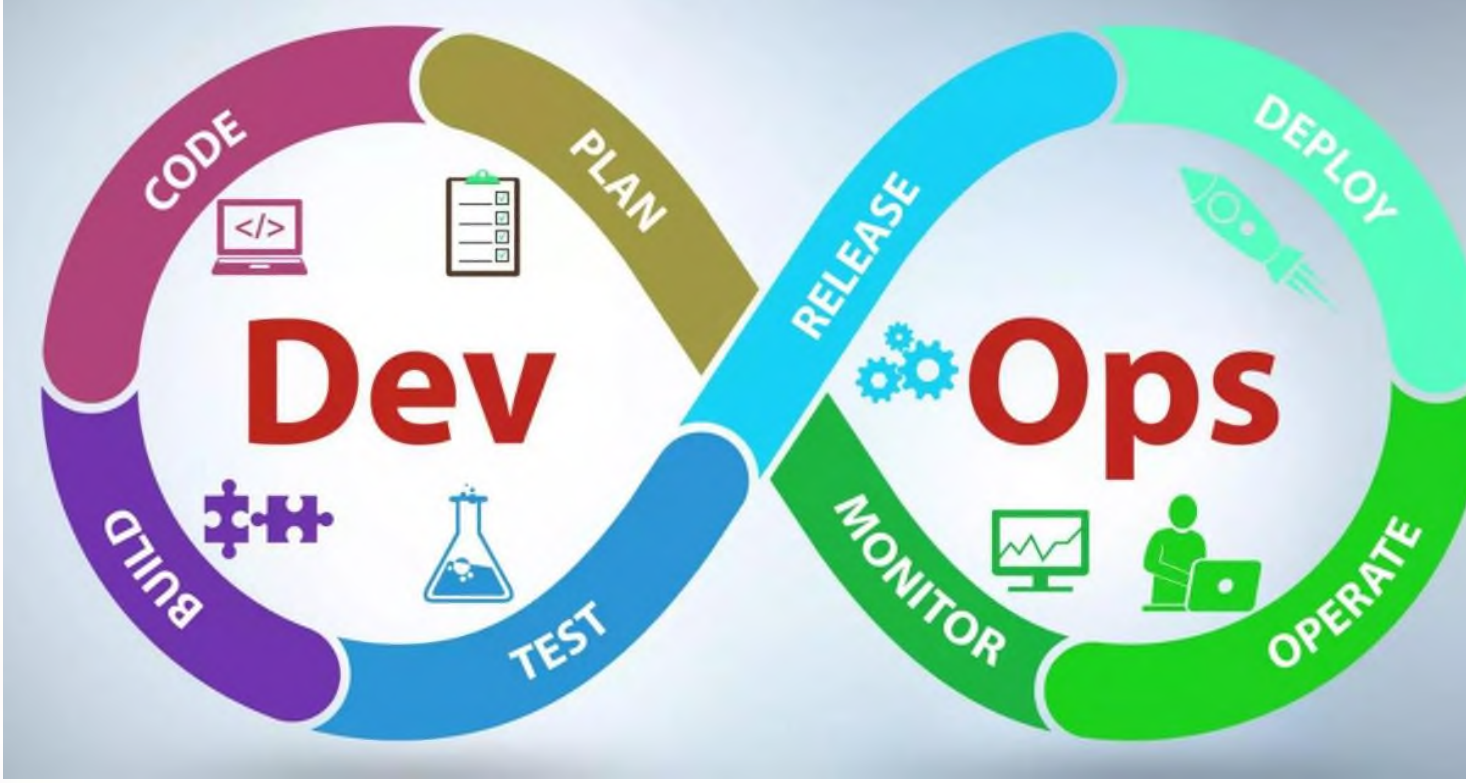
Данный режим задается установкой параметра authenticator: legacy

В этом режиме необходимо указать параметры user и password (либо задать соответствующие значения в переменных окружения *DECS_USER* и *DECS_PASSWORD*).

Чтобы воспользоваться этим режимом вам потребуется учетная запись (*legacy user*), заданная непосредственно на вашем DECS контроллере. Для создания такой учетной записи обратитесь к администратору облачной платформы.

Помимо вышеуказанных параметров также необходимо задать controller_url. В итоге *task* будет выглядеть следующим образом:

```
---
- hosts: ansible_master
  tasks:
  - name: manage VM in DECS cloud
    decs_vm:
      authenticator: legacy
      user: "{{ legacy_user_name }}"
      password: "{{ legacy_user_password }}"
      controller_url: "https://cloud.digitalenergy.online"
    <<<дальнейшие детали опущены>>>
...
```



МОДУЛЬ 13. Системы управления исходным кодом (SCM)

1. Виды и типы SCM и для чего применяются
2. Что такое Git?
3. Основы Git
4. Ветвления в Git. Хранилища репозиториев
5. Инструменты Git

1. Виды и типы SCM и для чего применяются

О системе контроля версий

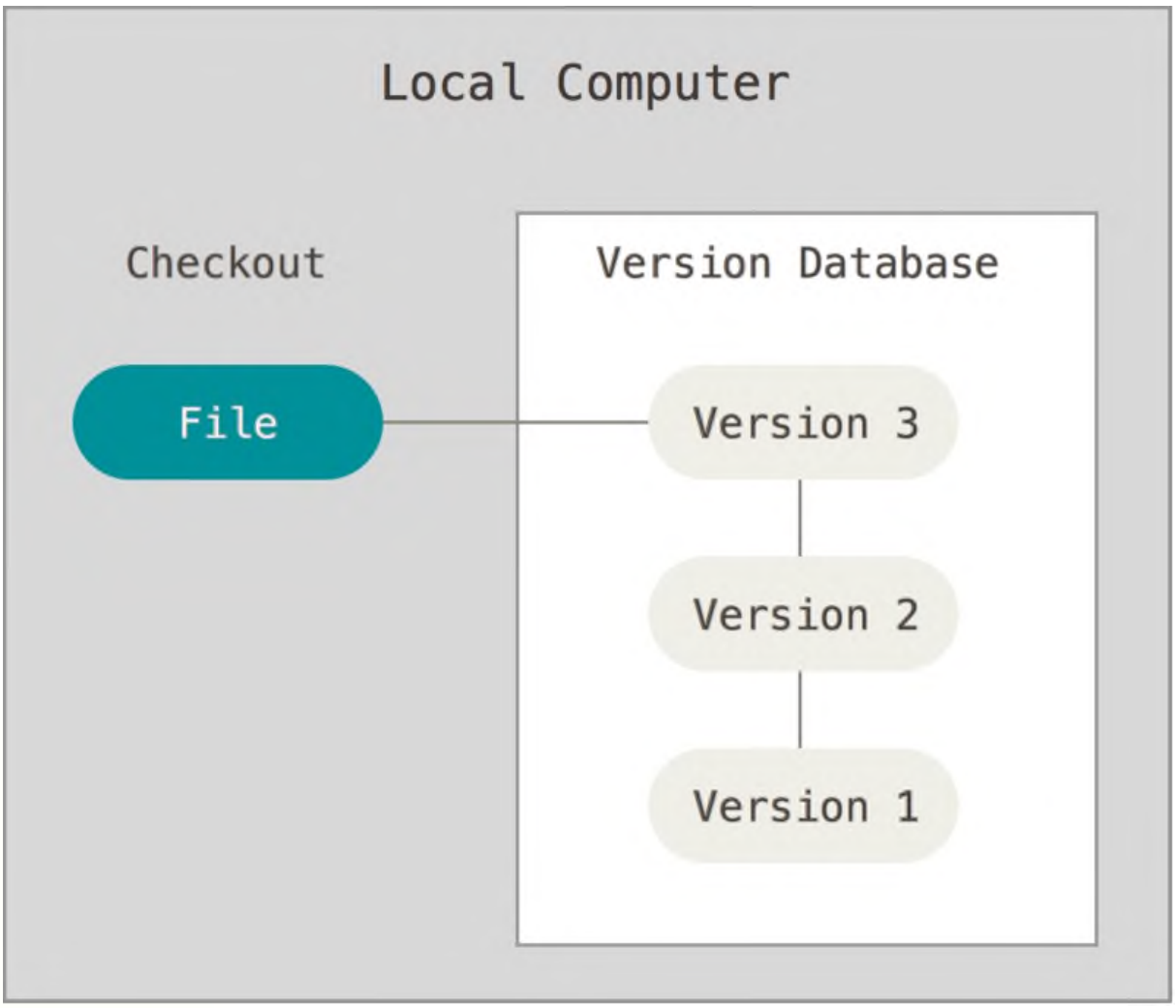
Что такое «система контроля версий» и почему это важно? Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в этой книге в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) — как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование СКВ также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий.

Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельный каталог (возможно даже, каталог с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть в каком каталоге вы находитесь и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи обо всех изменениях в файлах, осуществляя тем самым контроль ревизий.

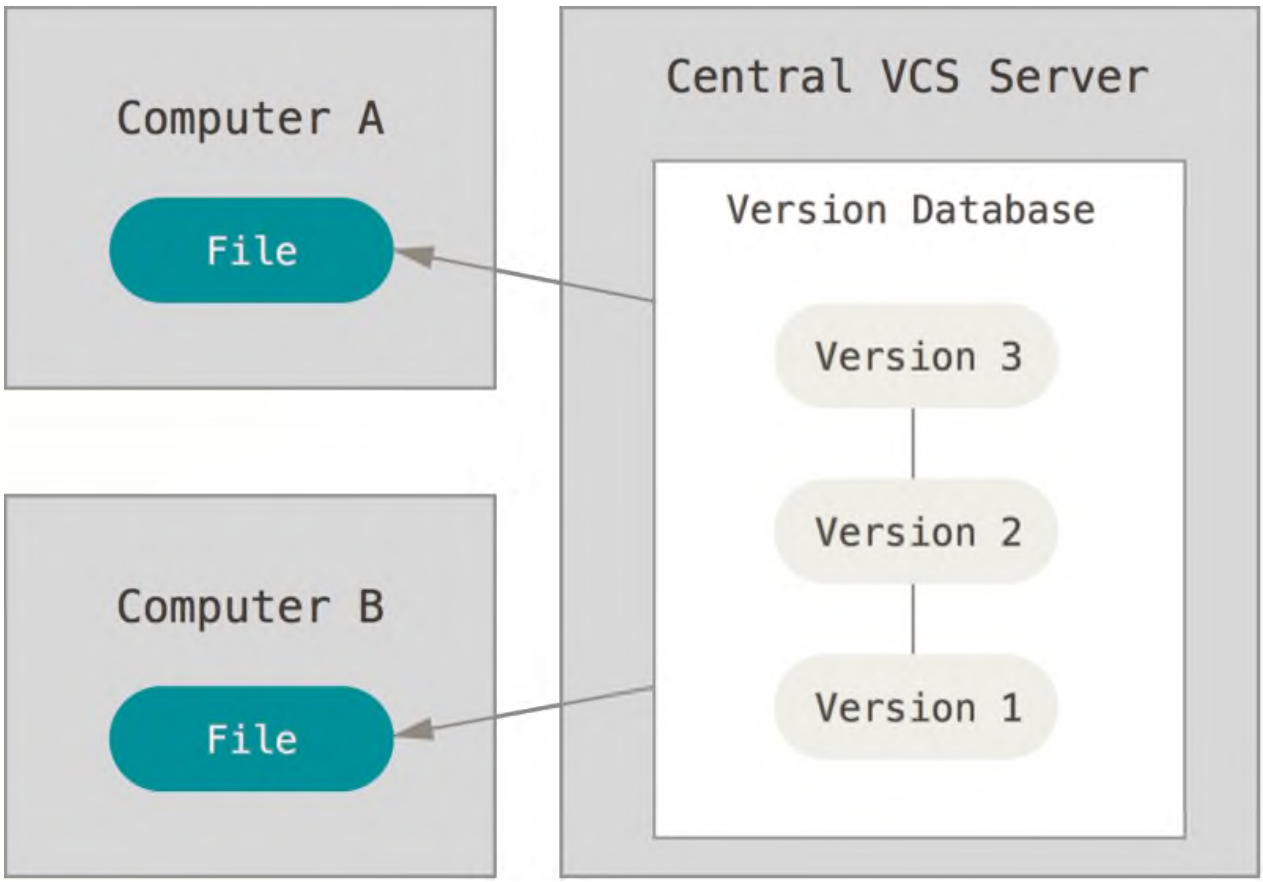


Локальный контроль версий

Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

Централизованные системы контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.



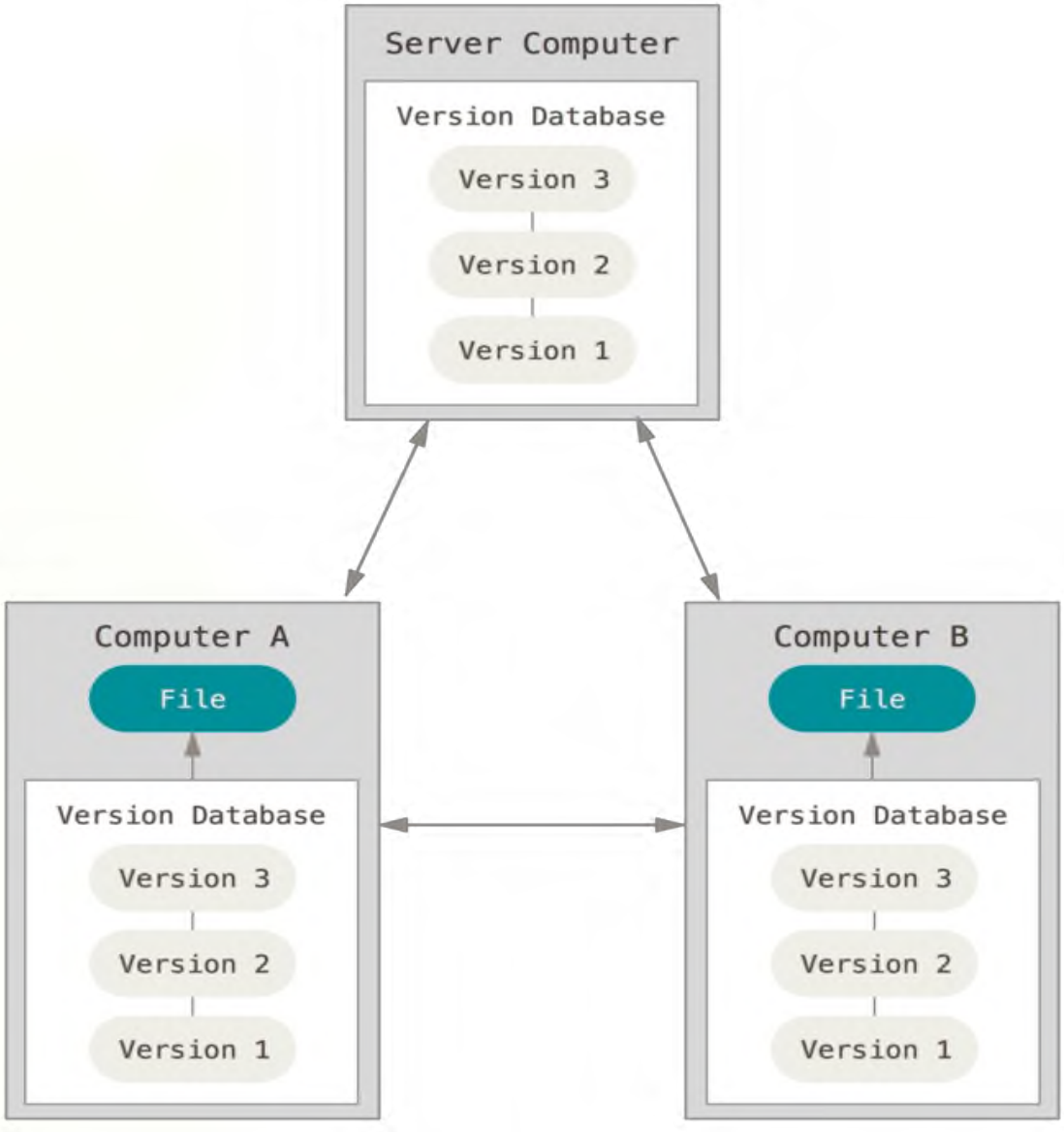
Централизованный контроль версий

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

Здесь в игру вступают распределённые системы контроля версий (РСКВ). В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.



Распределённый контроль версий

Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы

единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

Системы контроля версий стали неотъемлемой частью жизни не только разработчиков программного обеспечения, но и всех людей, столкнувшихся с проблемой управления интенсивно изменяющейся информацией, и желающих облегчить себе жизнь. Вследствие этого появилось большое число различных продуктов, предлагающих широкие возможности и предоставляющих обширные инструменты для управления версиями. В этой статье будут кратко рассмотрены наиболее популярные из них, приведены их достоинства и недостатки.

Для сравнения были выбраны наиболее распространенные системы контроля версий: RCS, CVS, Subversion, Aegis, Monoton, Git, Bazaar, Arch, Perforce, Mercurial, TFS.

RCS - система управления пересмотрами версий.

(www.gnu.org/software/rcs/rcs.html)

Начнем наш обзор с одной из первых систем контроля версий – RCS (Revision Control System – система управления пересмотрами версий), разработанной в 1985 году. Она пришла на смену популярной в то время системы контроля версий SCCS (Source Code Control System – система управления исходным кодом).

На данный момент RCS активно вытесняется более мощной системой контроля версий CVS, но все еще - достаточно популярна, и является частью проекта GNU.

RCS позволяет работать только с отдельными файлами, создавая для каждого историю изменений. Для текстовых файлов сохраняются не все версии файла, а только последняя версия и все изменение, внесенные в нее. RCS также может отслеживать изменения в бинарных файлах, но при этом каждое изменение хранится в виде отдельной версии файла.

Когда изменения в файл вносит один из пользователей, для всех остальных этот файл остается заблокированным. Они не могут запросить его из репозитория для редактирования, пока первый пользователь не закончит работу и не зафиксирует изменения.

Рассмотрим основные преимущества и недостатки системы контроля версий RCS.

Преимущества:

1. RCS - проста в использовании и хорошо подходит для ознакомления с принципами работы систем контроля версий.

2. Хорошо подходит для резервного копирования отдельных файлов, не требующих частого изменения группой пользователей.

3. Широко распространена и предустановленна в большинстве свободно распространяемых операционных системах.

Недостатки:

1. Отслеживает изменения только отдельных файлов, что не позволяет использовать ее для управления версиями больших проектов.

2. Не позволяет одновременно вносить изменения в один и тот же файл несколькими пользователями.

3. Низкая функциональность, по сравнению с современными системами контроля версий.

Выводы:

Система контроля версий RCS предоставляет слишком слабый набор инструментов для управления разрабатываемыми проектами и подходит разве что для ознакомления с технологией контроля версий или ведения небольшой истории откатов отдельных файлов.

CVS - система управления параллельными версиями.

(www.nongnu.org/cvs)

Система управления параллельными версиями (Concurrent Versions System) – логическое развитие системы управления пересмотрами версий (RCS), использующая ее стандарты и алгоритмы по управлению версиями, но значительно более функциональная, и позволяющая работать не только с отдельными файлами, но и с целыми проектами.

CVS основана на технологии клиент-сервер, взаимодействующих по сети. Клиент и сервер также могут располагаться на одной машине, если над проектом работает только один человек, или требуется вести локальный контроль версий.

Работа CVS организована следующим образом. Последняя версия и все сделанные изменения хранятся в репозитории сервера. Клиенты, подключаясь к серверу, проверяют отличия локальной версии от последней версии, сохраненной в репозитории, и, если есть отличия, загружают их в свой локальный проект. При необходимости решают конфликты и вносят требуемые изменения в разрабатываемый продукт. После этого все изменения загружаются в репозиторий сервера. CVS, при необходимости, позволяет откатываться на нужную версию разрабатываемого проекта и вести управление несколькими проектами одновременно.

Приведем основные достоинства и недостатки системы управления параллельными версиями.

Достоинства:

1. Несколько клиентов могут одновременно работать над одним и тем же проектом.

2. Позволяет управлять не одним файлом, а целыми проектами.

3. Обладает огромным количеством удобных графических интерфейсов, способных удовлетворить практически любой, даже самый требовательный вкус.

4. Широко распространена и поставляется по умолчанию с большинством операционных систем Linux.

5. При загрузке тестовых файлов из репозитория передаются только изменения, а не весь файл целиком.

Недостатки:

1. При перемещении или переименовании файла или директории теряются все, привязанные к этому файлу или директории, изменения.

2. Сложности при ведении нескольких параллельных веток одного и того же проекта.

3. Ограниченная поддержка шрифтов.

4. Для каждого изменения бинарного файла сохраняется вся версия файла, а не только внесенное изменение.

5. С клиента на сервер измененный файл всегда передается полностью.

6. Ресурсоемкие операции, так как требуют частого обращения к репозиторию, и сохраняемые копии имеют некоторую избыточность.

Выводы:

Несмотря на то, что CVS устарела и обладает серьезными недостатками, она все еще является одной из самых популярных систем контроля версий и отлично подходит для управления небольшими проектами, не требующих создания нескольких параллельных версий, которые надо периодически объединять. CVS можно порекомендовать, как промежуточный шаг в освоении работы систем контроля версий, ведущий к более мощным и современным видам таких программ.

Система управления версиями Subversion. (www.subversion.tigris.org)

Subversion – эта централизованная система управления версиями, созданная в 2000 году и основанная на технологии клиент-сервер. Она обладает всеми достоинствами CVS и решает основные ее проблемы (переименование и перемещение файлов и каталогов, работа с двоичными файлами и т. д.). Часто ее называют по имени клиентской части – SVN.

Принцип работы с Subversion очень походит на работу с CVS. Клиенты копируют изменения из репозитория и объединяют их с локальным проектом пользователя. Если возникают конфликты локальных изменений и изменений, сохраненных в репозитории, то такие ситуации разрешаются вручную. Затем в локальный проект вносятся изменения, и полученный результат сохраняется в репозитории.

При работе с файлами, не позволяющими объединять изменения, может использоваться следующий принцип:

1. Файл скачивается из репозитория и блокируется (запрещается его скачивание из репозитория).

2. Вносятся необходимые изменения.

3. Загружается файл в репозиторий и разблокируется (разрешается его скачивание из репозитория другим клиентам).

Во многом, из-за простаты и схожести в управлении с CVS, но в основном, из-за своей широкой функциональности, Subversion с успехом конкурирует с CVS и даже успешно ее вытесняет.

Однако, и у Subversion есть недостатки. Давайте рассмотрим ее слабые и сильные стороны для сравнения с другими системами управления версиями.

Достоинства:

1. Система команд, схожая с CVS.

2. Поддерживается большинство возможностей CVS.

3. Разнообразные графические интерфейсы и удобная работа из консоли.

4. Отслеживается история изменения файлов и каталогов даже после их переименования и перемещения.

5. Высокая эффективность работы, как с текстовыми, так и с бинарными файлами.

6. Встроенная поддержка во многие интегрированные средства разработки, такие как KDevelop, Zend Studio и многие другие.

7. Возможность создания зеркальных копий репозитория.

8. Два типа репозитория – база данных или набор обычных файлов.

9. Возможность доступа к репозиторию через Apache с использованием протокола WebDAV.

10. Наличие удобного механизма создания меток и ветвей проектов.

11. Можно с каждым файлом и директорией связать определенный набор свойств, облегчающий взаимодействие с системой контроля версии.

12. Широкое распространение позволяет быстро решить большинство возникающих проблем, обратившись к данным, накопленным Интернет-сообществом.

Недостатки:

1. Полная копия репозитория хранится на локальном компьютере в скрытых файлах, что требует достаточно большого объема памяти.

2. Существуют проблемы с переименованием файлов, если переименованный локально файл одним клиентом был в это же время изменен другим клиентом и загружен в репозиторий.

3. Слабо поддерживаются операции слияния веток проекта.

4. Сложности с полным удалением информации о файлах, попавших в репозиторий, так как в нем всегда остается информация о предыдущих изменениях файла, и непредусмотренно никаких штатных средств для полного удаления данных о файле из репозитория.

Выводы:

Subversion – современная система контроля версий, обладающая широким набором инструментов, позволяющих удовлетворить любые нужды для управления версиями проекта с помощью централизованной системы контроля. В Интернете множество ресурсов посвящено особенностям Subversion, что позволяет быстро и качественно решать все возникающие в ходе работы проблемы.

Простота установки, подготовки к работе и широкие возможности позволяют ставить subversion на одну из лидирующих позиций в конкурентной гонке систем контроля версий.

Система управления версиями Aegis. (www.aegis.sourceforge.net)

Aegis, созданная Питером Миллером в 1991 году, является первой альтернативой централизованным системам управления версиями. Все операции в ней производятся через файловую систему Unix. К сожалению, в Aegis нет встроенной поддержки работы по сети, но взаимодействия можно осуществлять, используя такие протоколы, как NFS, HTTP, FTP.

Основная особенность Aegis – это способ контроля вносимых в репозиторий изменений.

Во-первых, перед занесением каких-либо изменений, они должны обязательно пройти ряд тестов. И если нововведения в исходный код программы не проходят тесты, то требуется либо добавлять новые тесты, либо исправлять возможные ошибки в исходном коде.

Во-вторых, перед внесением изменений в основную ветку разрабатываемого проекта, они должны быть одобрены обозревателем.

В-третьих, предусмотрена иерархия доступа к репозиторию, основанная на системе прав доступа Unix-подобных операционных систем к файлам.

Все это делает использование системы контроля версий Aegis надежным, но крайне сложным, и даже хорошо проработана документация не сильно это облегчает.

Выделим основные достоинства и недостатки системы контроля версий Aegis.

Достоинства:

1. Надежный контроль корректности загружаемых изменений.
2. Возможность предоставлять различные уровни доступа к фалам репозитория, что дает приличный уровень безопасности.
3. Качественная документация.
4. Возможность переименовывать файлы, сохраненные в репозитории, без потери истории изменений.
5. Возможность работы с локальным репозиторием, если отсутствует сетевой доступ к главному репозиторию.

Недостатки:

1. Отсутствие встроенной поддержки сетевого взаимодействия.
2. Сложность настройки и работы с репозиторием.
3. Слабые графические интерфейсы.

Выводы:

Сложность работы Aegis может оттолкнуть пользователей от использования систем контроля версий, поэтому ее нельзя рекомендовать для ознакомления или ведения небольших программных проектов. Однако, она имеет ряд преимуществ, которые могут быть полезны в некоторых специфических ситуациях, особенно когда требуется жесткий контроль за качеством разрабатываемого программного обеспечения.

Система управления версиями Monotone. (monotone.ca)

Monotone – еще одна децентрализованная система управления версиями, разработанная Грейдоном Хоэм. В ней каждый клиент сам отвечает за синхронизацию версий разрабатываемого продукта с другими клиентами.

Работа с этой системой контроля версий - достаточно проста, а многие команды - схожи с командами, используемыми в Subversion и CVS. Отличия, в основном, заключаются в организации слияния ветвей проектов различных разработчиков.

Работа с Monotone строится следующим образом. В первую очередь, создается база данных проекта SQLite, и генерируются ключи с использованием алгоритма хеширования SHA1 (Secure Hash Algorithm 1).

Затем, по ходу корректировки проекта пользователем, все изменения сохраняются в этой базе данных, аналогично сохранению изменений в репозитории других систем контроля версий.

Для синхронизации проекта с другими пользователями необходимо:

- Экспортировать ключ (хэш-код последней версии проекта) и получить аналогичные ключи от других клиентов.

- Сохранить всем клиентам полученные ключи в связке ключей своих локальных проектах Monotone.

- Теперь каждый, зарегистрированный таким образом пользователь, может синхронизировать разработку со своими коллегами, используя простой набор команд.

Обобщим достоинства и недостатки системы контроля версий Monotone.

Достоинства:

1. Простой и понятный набор команд, схожий с командами Subversion и CVS.

2. Поддерживает переименование и перемещение файлов и директорий.

3. Качественная документация, значительно облегчающая использование системы контроля версий.

Недостатки:

1. Низкая скорость работы.

2. Отсутствие мощных графических оболочек.

3. Возможные (но чрезвычайно низкие) совпадения хэш-кода отличных по содержанию ревизий.

Выводы:

Monotone — это мощный и удобный инструмент для управления версиями разрабатываемого проекта. Набор команд - продуман и интуитивно понятен, особенно он будет удобен для пользователей, привыкших к работе с Subversion и CVS. Прекрасно оформленная и полная документация позволит быстро освоиться и использовать все возможности Subversion на полную мощность.

Однако, относительно низкая скорость работы и отсутствие мощных графических оболочек, возможно, сделает работу с большими проектами несколько затруднительной. Поэтому, если вам требуется система контроля

версий для поддержки сложных и объемных продуктов, стоит обратить внимание на Git или Mercurial.

Система управления версиями Git. (www.git-scm.com)

С февраля 2002 года для разработки ядра Linux'а большинством программистов стала использоваться система контроля версий BitKeeper. Довольно долгое время с ней не возникало проблем, но в 2005 году Лари МакВоем (разработчик BitKeeper'а) отозвал бесплатную версию программы.

Разрабатывать проект масштаба Linux без мощной и надежной системы контроля версий – невозможно. Одним из кандидатов и наиболее подходящим проектом оказалась система контроля версий Monotine, но Торвальдса Линуса не устроила ее скорость работы. Так как особенности организации Monatone не позволяли значительно увеличить скорость обработки данных, то 3 апреля 2005 года Линус приступил к разработке собственной системы контроля версий – Git.

Практически одновременно с Линусом (на три дня позже), к разработке новой системы контроля версий приступил, и Мэтт Макал. Свой проект Мэтт назвал Mercurial, но об этом позже, а сейчас вернемся к распределенной системе контроля версий Git.

Git – это гибкая, распределенная (без единого сервера) система контроля версий, дающая массу возможностей не только разработчикам программных продуктов, но и писателям для изменения, дополнения и отслеживания изменения «рукописей» и сюжетных линий, и учителям для корректировки и развития курса лекций, и администраторам для ведения документации, и для многих других направлений, требующих управления историей изменений.

У каждого разработчика, использующего Git, есть свой локальный репозиторий, позволяющий локально управлять версиями. Затем, сохраненными в локальный репозиторий данными, можно обмениваться с другими пользователями.

Часто при работе с Git создают центральный репозиторий, с которым остальные разработчики синхронизируются. Пример организации системы с центральным репозиторием – это проект разработки ядра Linux'а (http://www.kernel.org).

В этом случае все участники проекта ведут свои локальны разработки и беспрепятственно скачивают обновления из центрального репозитория. Когда необходимые работы отдельными участниками проекта выполнены

и отлажены, они, после удостоверения владельцем центрального репозитория в корректности и актуальности проделанной работы, загружают свои изменения в центральный репозиторий.

Наличие локальных репозиторием также значительно повышает надежность хранения данных, так как, если один из репозиториев выйдет из строя, данные могут быть легко восстановлены из других репозиториев.

Работа над версиями проекта в Git может вестись в нескольких ветках, которые затем могут с легкостью полностью или частично объединяться, уничтожаться, откатываться и разрастаться во все новые и новые ветки проекта.

Можно долго обсуждать возможности Git’а, но для краткости и более простого восприятия приведем основные достоинства и недостатки этой системы управления версиями

Достоинства:

1. Надежная система сравнения ревизий и проверки корректности данных, основанные на алгоритме хеширования SHA1 (Secure Hash Algorithm 1).

2. Гибкая система ветвления проектов и слияния веток между собой.

3. Наличие локального репозитория, содержащего полную информацию обо всех изменениях, позволяет вести полноценный локальный контроль версий и заливать в главный репозиторий только полностью прошедшие проверку изменения.

4. Высокая производительность и скорость работы.

5. Удобный и интуитивно понятный набор команд.

6. Множество графических оболочек, позволяющих быстро и качественно вести работы с Git’ом.

7. Возможность делать контрольные точки, в которых данные сохраняются без дельты компрессии, а полностью. Это позволяет уменьшить скорость восстановления данных, так как за основу берется ближайшая контрольная точка, и восстановление идет от нее. Если бы контрольные точки отсутствовали, то восстановление больших проектов могло бы занимать часы.

8. Широкая распространенность, легкая доступность и качественная документация.

9. Гибкость системы позволяет удобно ее настраивать и даже создавать специализированные контроля системы или пользовательские интерфейсы на базе git.

10. Универсальный сетевой доступ с использованием протоколов http, ftp, rsync, ssh и др.

Недостатки:

1. Unix – ориентированность. На данный момент отсутствует зрелая реализация Git, совместимая с другими операционными системами.

2. Возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по содержанию ревизий.

3. Не отслеживается изменение отдельных файлов, а только всего проекта целиком, что может быть неудобно при работе с большими проектами, содержащими множество несвязных файлов.

4. При начальном (первом) создании репозитория и синхронизации его с другими разработчиками, потребуется достаточно длительное время для скачивания данных, особенно если проект большой, так как требуется скопировать на локальный компьютер весь репозиторий.

Выводы:

Git – гибкая, удобная и мощная система контроля версий, способная удовлетворить абсолютное большинство пользователей. Существующие недостатки постепенно удаляются и не приносят серьезных проблем пользователям. Если вы ведете большой проект, территориально удаленный, и тем более, если часто приходится разрабатывать программное обеспечение, не имея доступа к другим разработчикам (например, вы не хотите терять время при перелете из страны в страну или во время поездки на работу), можно делать любые изменения и сохранять их в локальном репозитории, откатываться, переключаться между ветками и т. д.). Git – один из лидеров систем контроля версий.

Система управления версиями Mercurial. (mercurial.selenic.com)

Распределенная система контроля версий Mercurial разрабатывалась Мэттом Макалом параллельно с системой контроля версий Git, созданной Торвальдсом Линусом.

Первоначально, она была создана для эффективного управления большими проектами под Linux’ом, а поэтому была ориентирована на быструю и надежную работу с большими репозиториями. На данный момент mercurial адаптирован для работы под Windows, Mac OS X и большинство Unix систем.

Большая часть системы контроля версий написана на языке Python, и только отдельные участки программы, требующие наибольшего быстродействия, написаны на языке Си.

Идентификация ревизий происходит на основе алгоритма хеширования SHA1 (Secure Hash Algorithm 1), однако, также предусмотрена возможность присвоения ревизиям индивидуальных номеров.

Так же, как и в git'е, поддерживается возможность создания веток проекта с последующим их слиянием.

Для взаимодействия между клиентами используются протоколы HTTP, HTTPS или SSH.

Набор команд - простой и интуитивно понятный, во многом схожий с командами subversion. Так же имеется ряд графических оболочек и доступ к репозиторию через веб-интерфейс. Немаловажным является и наличие утилит, позволяющих импортировать репозитории многих других систем контроля версий.

Рассмотрим основные достоинства и недостатки Mercurial.

Достоинства:

1. Быстрая обработка данных.
2. Кроссплатформенная поддержка.
3. Возможность работы с несколькими ветками проекта.
4. Простота в обращение.
5. Возможность конвертирования репозиториев других систем поддержки версий, таких как CVS, Subversion, Git, Darcs, GNU Arch, Bazaar и др.

Недостатки:

1. Возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по содержанию ревизий.
2. Ориентирован на работу в консоли.

Выводы:

Простой и отточенный интерфейс, и набор команд, возможность импортировать репозитории с других систем контроля версий, - сделают переход на Mercurial и обучение основным особенностями безболезненным и быстрым. Вряд ли это займет больше нескольких дней.

Надежность и скорость работы позволяют использовать его для контроля версий огромных проектов. Все это делает mercurial достойным конкурентом git'а.

Система управления версиями Bazaar. (bazaar.canonical.com)

Bazaar – распределенная, свободно распространяемая система контроля версий, разрабатываемая при поддержке компании Canonical Ltd. Написана на языке Python и работает под управлением операционных систем Linux, Mac OS X и Windows.

В отличие от Git и Mercurial, создаваемых для контроля версий ядра операционной системы Linux, а поэтому ориентированных на максимальное быстродействие при работе с огромным числом файлов, Bazaar ориентировался на удобный и дружественный интерфейс пользователя. Оптимизация скорости работы производилось уже на втором этапе, когда первые версии программы уже появились.

Как и во многих других системах контроля версий, система команд Bazaar'a - очень похожа на команды CVS или Subversion, что, впрочем, неудивительно, так как обеспечивает удобный, простой и интуитивно понятный интерфейс взаимодействия с программой.

Приятно, что огромное внимание уделяется работе с ветками проектов (создание, объединение веток и т. д.), что очень важно при разработке серьезных проектов и позволяет проводить доработки и эксперименты без угрозы потери основной версии программного обеспечения.

Большой плюс этой системе контроля версий дает возможность работы с репозиториями других систем контроля версий, таких как Subversion или Git.

Кратко приведем наиболее значительные достоинства и недостатки этой системы контроля версий.

Достоинства:

1. Кросплатформенная поддержка.
2. Удобный и интуитивно понятный интерфейс.
3. Простая работа с ветками проекта.
4. Возможность работы с репозиториями других систем контроля версий.
5. Великолепная документация.
6. Удобный графический интерфейс.
7. Чрезвычайная гибкость, позволяющая подстроится под нужды конкретного пользователя.

Недостатки:

1. Более низкая скорость работы, по сравнению с git и mercurial, но эта ситуация постепенно исправляется.
2. Для полноценного функционирования необходимо устанавливать достаточно большое количество плагинов, позволяющих полностью раскрыть все возможности системы контроля версий.

Выводы:

Bazaar – удобная система контроля версий с приятным интерфейсом. Хорошо подходит для пользователей, которых отталкивает перспектива работы с командной строкой. Множество дополнительных опций и расширений позволит настроить программу под свои нужды. Схожесть системы команд с Git и Subversion, и возможность работы напрямую с их репозиториями, - сделает переход на Bazaar быстрым и безболезненным. Об успешности базара говорит и тот факт, что ей пользуются разработчики Ubuntu Linux.

Система управления версиями Arch.

Arch – распределенная система контроля версий, созданная Томом Лордом. Изначально она создавалась для решения проблем CVS, что им вполне удалось.

Arch осуществляет атомарные операции по сохранению изменений в репозиторий, т. е. исключает ситуацию скачивания репозитория, когда часть изменений загружена, а часть еще не успела загрузиться.

Поддерживаются возможности ветвления версий проекта и объединение отдельных веток, переименование и перемещение файлов и каталогов с сохранением истории изменений, и многие другие приятные возможности.

Не требует специального сервиса для сетевого репозитория и может использовать такие протоколы, как FTP, SFTP или WebDAV и так далее.

Но, к сожалению, поддерживается только UNIX – системами, однако, перевод Arch под другие операционные системы не должен составлять труда.

Трудно отметить какие то принципиально лучшие качества, по сравнению с другими распределенным системами контроля версий, такими как git, mercurial, bazaar, так что если есть выбор, то лучше использовать что-то более мощное и распространенное.

Система управления версиями Perforce. (www.perforce.com)

Продолжим обзор систем контроля версий и перейдем к коммерческим программам. Начнем с централизованной системы контроля версий – Perforce, разработанной компанией Perforce Software.

Система Perforce имеет клиент-серверную организацию и позволяет одновременно управлять несколькими проектами, создавая для каждого проекта свой репозиторий.

Perforce – кроссплатформенная система. Существуют версии, способные работать под управлением операционных систем Unix, Mac OS X, Microsoft Windows.

Для работы с системой контроля версий можно использовать, как консоль, так и специально разработанный графический интерфейс.

Серьезное преимущество Perforce'у дает возможность интегрироваться со множеством средств разработки программного обеспечения и такими приложениями, как Autodesk 3D Studio Max, Maya, Adobe Photoshop, Microsoft Office, Eclipse, emacs и многими другими.

Поддержка возможности создания веток версий проекта, гибко управлять ими, объединять, откатываться на предыдущие ревизии, - делает Perforce вполне конкурентно способной системой и способствуют ее широкому распространению. Однако, это продукт - коммерческий, что несколько сужает область его применения и сдерживает распространение. В основном, он используется в больших коммерческих компаниях, для которых важна не только функциональность, но и своевременная техническая поддержка.

Система управления версиями Team Foundation Server. **(msdn.microsoft.com/en-us/library/ms364061.aspx)**

Собственно говоря, нельзя назвать Team Foundation Server (TFC) просто системой контроля версий – это некое комплексное решение, в состав которого входит и система управления версиями, и система сбора данных, и построения отчетов, и другие полезные функции.

Управляемый проект при работе с TFC представляет собой ветки исходного кода проекта, наборы отчетов и пользовательские элементы. При создании проекта заранее выбираются его параметры, которые можно, как выбирать самостоятельно, так и использовать шаблоны. Шаблоны позволяют определить путь развития проекта, сделать его гибким или жестко формализованным, заложить стратегию развития, учесть необходимые заготовки документов и отчетов.

TFC легко интегрируется с Microsoft Excel и Microsoft Project, что значительно облегчает создание и отслеживание элементов контролируемых проектов.

В качестве системы контроля версий TFC позволяет:

- совместно корректировать файлы проекта;
- разрешать конфликты;
- создавать ветки проектов, а затем объединять их;
- управлять доступом к репозиторию;

- откатываться на предыдущие версии;

- создавать отложенные изменения - изменения, которые, непосредственно, не добавлены в репозиторий, но их могут видеть другие пользователи, причем скачать эти изменения можно, только получив специальное разрешение от владельца изменений;

- помечать отдельные версии файлов в репозитории и группировать их;

Для сохранения данных и репозиториев разрабатываемых проектов используются базы данных SQL Server 2005.

TFC - мощный и удобный инструмент, позволяющий не только управлять версиями исходного кода, но и полностью организовывать весь цикл разработки проекта от написания программ до их документации. Однако, эта мощная и сложная система больше подходит для ведения больших проектов, требующих сложного и досконального управления разработкой. Если у вас - небольшая разработка, то имеет смысл использовать менее мощный инструмент, а еще лучше свободно распространяемый, так как это вам сэкономит время, деньги и нервы.

Обобщение.

Большой выбор систем контроля версий позволяет удовлетворить любые требования и организовать работу так, как вам необходимо. Однако, среди всего многообразия систем есть явные лидеры. Так, если необходимо управлять огромным проектом, состоящим из десятков тысяч файлов и над которым работу ведут тысячи человек, то лучше всего выбор остановить на Git или Mercurial. Если для вас главное – удобный интерфейс, а разрабатываемый проект - не очень большой, то для вас предпочтительна система Bazaar.

Для программистов-одиночек или небольших проектов, не требующих ветвления и создания множества версий, лучше всего подойдет Subversion.

Но, в конечном итоге, выбор — это дело вкуса, так как сейчас существует множество систем контроля версий, которые дадут вам все необходимое. Так что выбирайте и не пожалеете. Системы контроля версий – это абсолютно необходимое программное обеспечения для каждого разработчика и не только.

Краткая история Git

Как и многие вещи в жизни, Git начинался с капелькой творческого хаоса и бурных споров.

Ядро Linux — это достаточно большой проект с открытым исходным кодом. Большую часть времени разработки ядра Linux (1991–2002 гг.) изменения передавались между разработчиками в виде патчей и архивов. В 2002 году проект ядра Linux начал использовать проприетарную децентрализованную СКВ BitKeeper.

В 2005 году отношения между сообществом разработчиков ядра Linux и коммерческой компанией, которая разрабатывала BitKeeper, прекратились, и бесплатное использование утилиты стало невозможным. Это сподвигло сообщество разработчиков ядра Linux (а в частности, Линуса Торвальдса — создателя Linux) разработать свою собственную утилиту, учитывая уроки, полученные при работе с BitKeeper. Некоторыми целями, которые преследовала новая система, были:

- Скорость
- Простая архитектура
- Хорошая поддержка нелинейной разработки (тысячи параллельных веток)
- Полная децентрализация
- Возможность эффективного управления большими проектами, такими как ядро Linux (скорость работы и разумное использование дискового пространства)

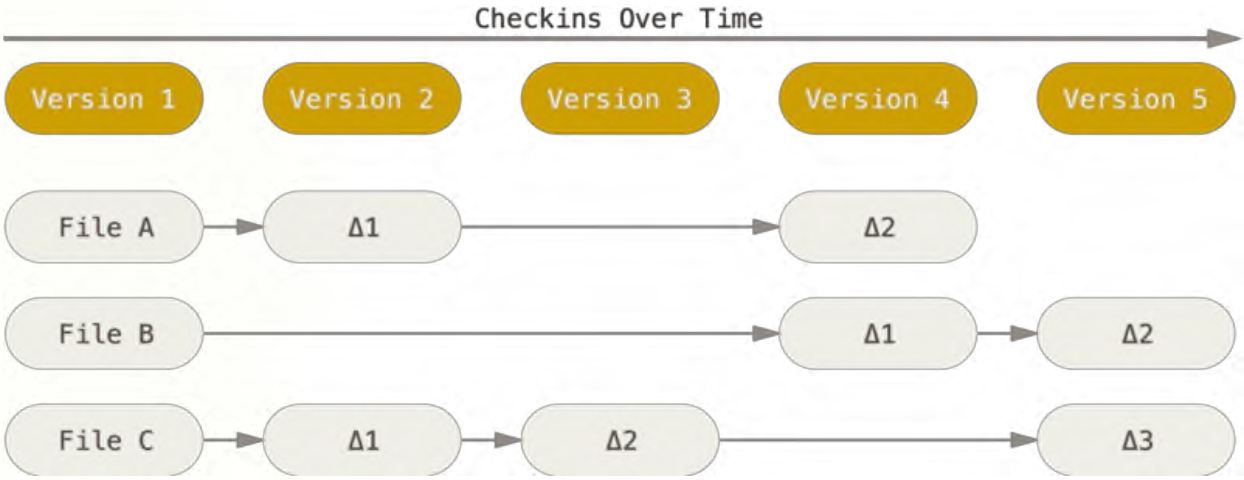
С момента своего появления в 2005 году, Git развился в простую в использовании систему, сохранив при этом свои изначальные качества. Он удивительно быстр, эффективен в работе с большими проектами и имеет великолепную систему веток для нелинейной разработки.

2. Что такое Git?

Что же такое Git, если говорить коротко? Очень важно понять эту часть материала, потому что если вы поймёте, что такое Git и основы того, как он работает, тогда, возможно, вам будет гораздо проще его использовать. Пока вы изучаете Git, попробуйте забыть всё, что вы знаете о других СКВ, таких как Subversion и Perforce. Это позволит вам избежать определённых проблем при использовании инструмента. Git хранит и использует информацию совсем иначе по сравнению с другими системами, даже несмотря на то, что интерфейс пользователя достаточно похож, и понимание этих различий поможет вам избежать путаницы во время использования.

Снимки, а не различия

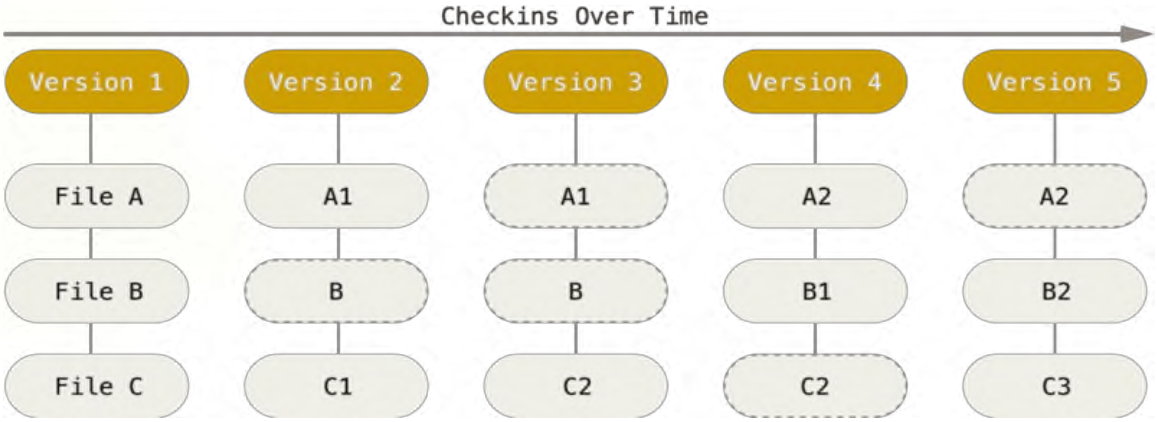
Основное отличие Git от любой другой СКВ (включая Subversion и её собратьев) — это подход к работе со своими данными. Концептуально, большинство других систем хранят информацию в виде списка изменений в файлах. Эти системы (CVS, Subversion, Perforce, Bazaar и т. д.) представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, **основанным на различиях**).



Хранение данных как набора изменений относительно первоначальной версии каждого из файлов.

Git не хранит и не обрабатывает данные таким способом. Вместо этого, подход Git к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не

запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, **поток снимков**.



Хранение данных как снимков проекта во времени.

Это очень важное отличие между Git и почти любой другой СКВ. Git переосмысливает практически все аспекты контроля версий, которые были скопированы из предыдущего поколения большинством других систем. Это делает Git больше похожим на миниатюрную файловую систему с удивительно мощными утилитами, надстроенными над ней, нежели просто на СКВ. Когда мы будем рассматривать управление ветками в главе Ветвление в Git, мы увидим, какие преимущества вносит такой подход к работе с данными в Git.

Почти все операции выполняются локально

Для работы большинства операций в Git достаточно локальных файлов и ресурсов — в основном, системе не нужна никакая информация с других компьютеров в вашей сети. Если вы привыкли к ЦСКВ, где большинство операций страдают от задержек из-за работы с сетью, то этот аспект Git заставит вас думать, что боги скорости наделили Git несказанной мощью. Так как вся история проекта хранится прямо на вашем локальном диске, большинство операций кажутся чуть ли не мгновенными.

Для примера, чтобы посмотреть историю проекта, Git не нужно соединяться с сервером для её получения и отображения — система просто считывает данные напрямую из локальной базы данных. Это означает, что вы увидите историю проекта практически моментально. Если вам необходимо посмотреть изменения, сделанные между текущей версией файла и версией, созданной месяц назад, Git может найти файл месячной давности и локально вычислить изменения вместо того, чтобы запрашивать удалённый сервер выполнить эту операцию, либо вместо получения старой версии файла с сервера и выполнения операции локально.

Это также означает, что есть лишь небольшое количество действий, которые вы не сможете выполнить, если вы находитесь оффлайн или не имеете доступа к VPN в данный момент. Если вы в самолёте или в поезде и хотите немного поработать, вы сможете создавать коммиты без каких-либо проблем (в вашу **локальную** копию, помните?): когда будет возможность подключиться к сети, все изменения можно будет синхронизировать. Если вы ушли домой и не можете подключиться через VPN, вы всё равно сможете работать. Добиться такого же поведения во многих других системах либо очень сложно, либо вовсе невозможно. В Perforce, для примера, если вы не подключены к серверу, вам не удастся сделать многого; в Subversion и CVS вы можете редактировать файлы, но вы не сможете сохранить изменения в базу данных (потому что вы не подключены к БД). Всё это может показаться не таким уж и значимым, но вы удивитесь, какое большое значение это может иметь.

Целостность Git

В Git для всего вычисляется хеш-сумма, и только потом происходит сохранение. В дальнейшем обращение к сохранённым объектам происходит по этой хеш-сумме. Это значит, что невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Данная функциональность встроена в Git на низком уровне и является неотъемлемой частью его философии. Вы не потеряете информацию во время её передачи и не получите повреждённый файл без ведома Git.

Механизм, которым пользуется Git при вычислении хеш-сумм, называется SHA-1 хеш. Это строка длиной в 40 шестнадцатеричных символов (0-9 и a-f), она вычисляется на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

24b9da6552252987aa493b52f8696cd6d3b00373

Вы будете постоянно встречать хеши в Git, потому что он использует их повсеместно. На самом деле, Git сохраняет все объекты в свою базу данных не по имени, а по хеш-сумме содержимого объекта.

Git обычно только добавляет данные

Когда вы производите какие-либо действия в Git, практически все из них только **добавляют** новые данные в базу Git. Очень сложно заставить систему удалить данные либо сделать что-то, что нельзя впоследствии отменить. Как и в любой другой СКВ, вы можете потерять или испортить свои изменения, пока они не зафиксированы, но после того, как вы зафиксируете снимок в Git, будет очень сложно что-либо потерять,

особенно если вы регулярно синхронизируете свою базу с другим репозиторием.

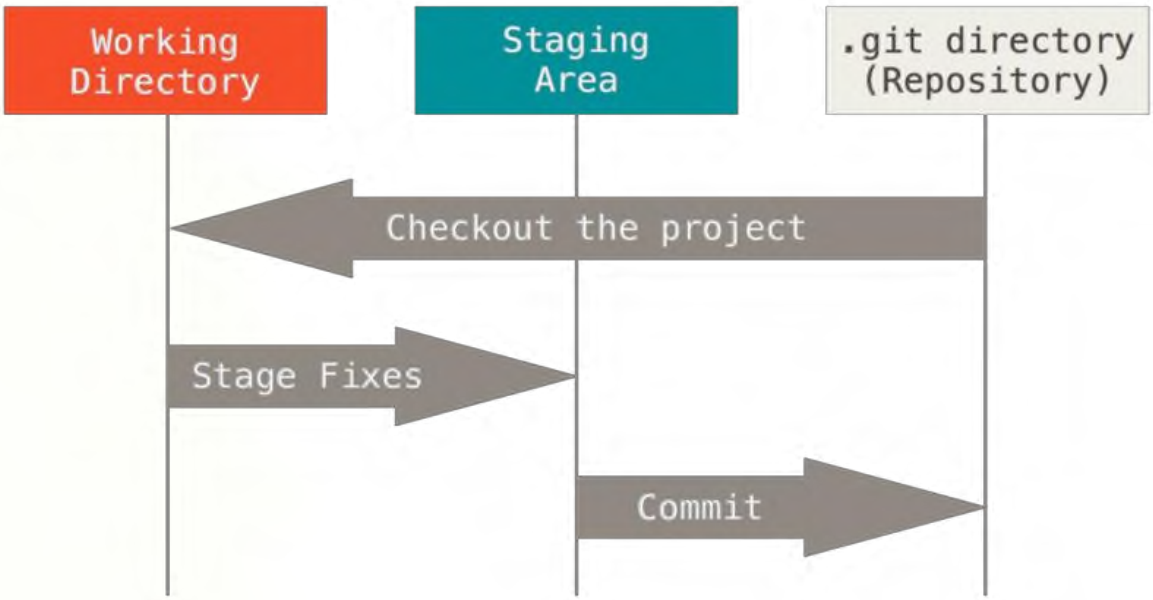
Всё это превращает использование Git в одно удовольствие, потому что мы знаем, что можем экспериментировать, не боясь серьёзных проблем. Для более глубокого понимания того, как Git хранит свои данные и как вы можете восстановить данные, которые кажутся утерянными, см. Операции отмены.

Три состояния

Это самая важная вещь, которую нужно запомнить о Git, если вы хотите, чтобы остаток процесса обучения прошёл гладко. У Git есть три основных состояния, в которых могут находиться ваши файлы: **изменён** (modified), **индексирован** (staged) и **зафиксирован** (committed):

- К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.
- Индексированный — это изменённый файл в его текущей версии, отмеченный для включения в следующий коммит.
- Зафиксированный значит, что файл уже сохранён в вашей локальной базе.

Мы подошли к трём основным секциям проекта Git: рабочая копия (working tree), область индексирования (staging area) и каталог Git (Git directory).



Рабочая копия, область индексирования и каталог Git.

Рабочая копия является снимком одной версии проекта. Эти файлы извлекаются из сжатой базы данных в каталоге Git и помещаются на диск, для того чтобы их можно было использовать или редактировать.

Область индексирования — это файл, обычно находящийся в каталоге Git, в нём содержится информация о том, что попадёт в следующий коммит. Её техническое название на языке Git — «индекс», но фраза «область индексирования» также работает.

Каталог Git — это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git и это та часть, которая копируется при **клонировании** репозитория с другого компьютера.

Базовый подход в работе с Git выглядит так:

1. Изменяете файлы вашей рабочей копии.
2. Выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки **только** этих изменений в индекс.
3. Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в ваш каталог Git.

Если определённая версия файла есть в каталоге Git, эта версия считается **зафиксированной** (committed). Если файл был изменён и добавлен в индекс, значит, он **индексирован** (staged). И если файл был изменён с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается **изменённым** (modified). В главе Основы Git вы узнаете больше об этих состояниях и какую пользу вы можете извлечь из них или как полностью пропустить часть с индексом.

3. Основы Git

Если вы хотите начать работать с Git, прочитав всего одну главу, то эта глава — то, что вам нужно. Здесь рассмотрены все базовые команды, необходимые вам для решения подавляющего большинства задач, возникающих при работе с Git. После прочтения этой главы вы научитесь настраивать и инициализировать репозиторий, начинать и прекращать контроль версий файлов, а также подготавливать и фиксировать изменения. Мы также продемонстрируем вам, как настроить в Git игнорирование отдельных файлов или их групп, как быстро и просто отменить ошибочные изменения, как просмотреть историю вашего проекта и изменения между отдельными коммитами (commit), а также как отправлять (push) и получать (pull) изменения в/из удалённого (remote) репозитория.

Создание Git-репозитория

Обычно вы получаете репозиторий Git одним из двух способов:

1. Вы можете взять локальный каталог, который в настоящее время не находится подверсионным контролем, и превратить его в репозиторий Git, либо
2. Вы можете **клонировать** существующий репозиторий Git из любого места.

В обоих случаях вы получите готовый к работе Git репозиторий на вашем компьютере.

Создание репозитория в существующем каталоге

Если у вас уже есть проект в каталоге, который не находится под версионным контролем Git, то для начала нужно перейти в него. Если вы не делали этого раньше, то для разных операционных систем это выглядит по-разному:

для Linux:

```
$ cd /home/user/my_project
```

для macOS:

```
$ cd /Users/user/my_project
```

для Windows:

```
$ cd C:/Users/user/my_project
```

а затем выполните команду:

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем `.git`, содержащий все необходимые файлы репозитория — структуру Git

репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. Подробное описание файлов, содержащихся в только что созданном вами каталоге .git, приведено в главе Git изнутри

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс и осуществить первый коммит изменений. Добиться этого вы сможете, запустив команду git add несколько раз, указав индексируемые файлы, а затем выполнив git commit:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Initial project version'
```

Мы разберем, что делают эти команды чуть позже. Теперь у вас есть Git-репозиторий с отслеживаемыми файлами и начальным коммитом.

Клонирование существующего репозитория

Для получения копии, существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду git clone. Если вы знакомы с другими системами контроля версий, такими как Subversion, то заметите, что команда называется «clone», а не «checkout». Это важное различие — вместо того, чтобы просто получить рабочую копию, Git получает копию практически всех данных, которые есть на сервере. При выполнении git clone с сервера забирается (pulled) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования (вы можете потерять часть серверных хуков (server-side hooks) и т. п., но все данные, помещённые под версионный контроль, будут сохранены.

Клонирование репозитория осуществляется командой git clone <url>. Например, если вы хотите клонировать библиотеку libgit2, вы можете сделать это следующим образом:

```
$ git clone https://github.com/libgit2/libgit2
```

Эта команда создаёт каталог libgit2, инициализирует в нём подкаталог .git, скачивает все данные для этого репозитория и извлекает рабочую копию последней версии. Если вы перейдёте в только что созданный каталог libgit2, то увидите в нём файлы проекта, готовые для работы или использования. Для того, чтобы клонировать репозиторий в

каталог с именем, отличающимся от libgit2, необходимо указать желаемое имя, как параметр командной строки:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван mylibgit.

В Git реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол https://, вы также можете встретить git:// или user@server:path/to/repo.git, использующий протокол передачи SSH. В разделе Установка Git на сервер главы 4 мы познакомимся со всеми доступными вариантами конфигурации сервера для обеспечения доступа к вашему Git репозиторию, а также рассмотрим их достоинства и недостатки.

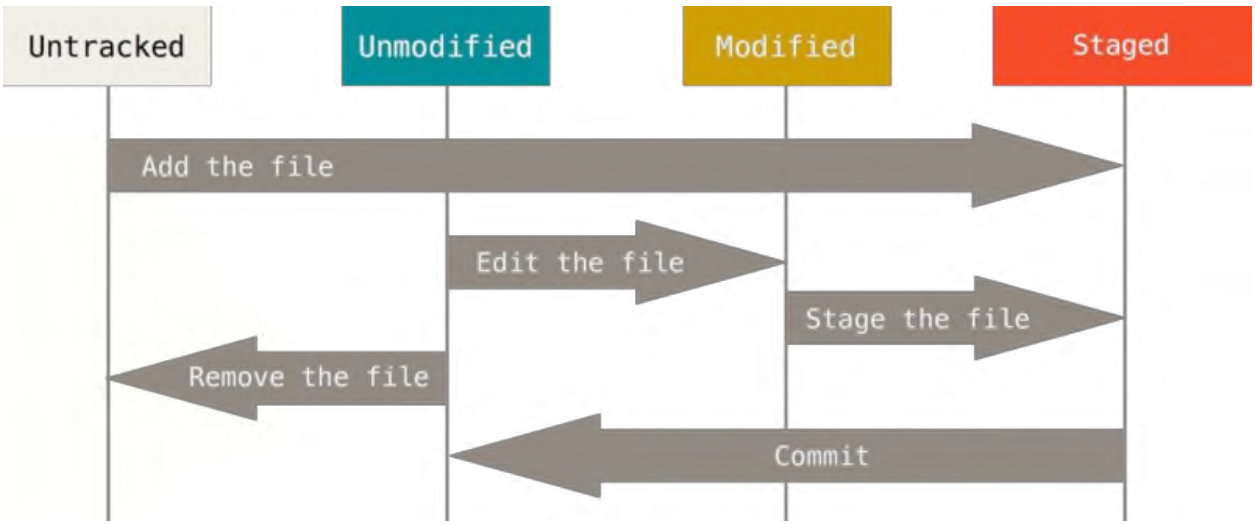
Запись изменений в репозиторий

Итак, у вас имеется настоящий Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать «снимки» состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем снимке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы — это те файлы, о которых знает Git.

Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что Git только что их извлек и вы ничего пока не редактировали.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется.



Жизненный цикл состояний файлов

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда `git status`. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Это означает, что у вас чистый рабочий каталог, другими словами — в нем нет отслеживаемых измененных файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам на какой ветке вы находитесь и сообщает вам, что она не расходится с веткой на сервере. Пока что это всегда ветка `master`, ветка по умолчанию; в этой главе это не важно. В главе Ветвление в Git будут рассмотрены ветки и ссылки более детально.

Предположим, вы добавили в свой проект новый файл, простой файл `README`. Если этого файла раньше не было, и вы выполните `git status`, вы увидите свой неотслеживаемый файл — вот так:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Понять, что новый файл README неотслеживаемый можно по тому, что он находится в секции «Untracked files» в выводе команды status. Статус Untracked означает, что Git видит файл, которого не было в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить README, так давайте сделаем это.

Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда git add. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

```
$ git add README
```

Если вы снова выполните команду status, то увидите, что файл README теперь отслеживаемый и добавлен в индекс:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

    new file:   README
```

Вы можете видеть, что файл проиндексирован, так как он находится в секции «Changes to be committed». Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды git add, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили git init, затем вы выполнили git add (файлы) — это было сделано для того, чтобы добавить файлы в вашем каталоге под версионный контроль. Команда git add принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет все файлы из указанного каталога в индекс.

Индексация изменённых файлов

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл CONTRIBUTING.md и после этого снова выполните команду git status, то результат будет примерно следующим:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Файл CONTRIBUTING.md находится в секции «Changes not staged for commit» — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду git add. Это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния. Вам может быть понятнее, если вы будете думать об этом как «добавить этот контент в следующий коммит», а не как «добавить этот файл в проект». Выполним git add, чтобы проиндексировать CONTRIBUTING.md, а затем снова выполним git status:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

```
    modified:   CONTRIBUTING.md
```

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в CONTRIBUTING.md до коммита. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним git status:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Что за чёрт? Теперь CONTRIBUTING.md отображается как проиндексированный и непроиндексированный одновременно. Как такое возможно? Такая ситуация демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду git add. Если вы выполните коммит сейчас, то файл CONTRIBUTING.md попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду git add , а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения git commit. Если вы изменили файл после выполнения git add, вам придётся снова выполнить git add, чтобы проиндексировать последнюю версию файла:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
```

```
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Сокращенный вывод статуса

Вывод команды git status довольно всеобъемлющий и многословный. Git также имеет флаг вывода сокращенного статуса, так что вы можете увидеть изменения в более компактном виде. Если вы выполните git status -s или git status --short вы получите гораздо более упрощенный вывод:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Новые неотслеживаемые файлы помечены ?? слева от них, файлы добавленные в отслеживаемые помечены A, отредактированные файлы помечены M и так далее. В выводе содержится два столбца — в левом указывается статус файла, а в правом модифицирован ли он после этого. К примеру, в нашем выводе, файл README модифицирован в рабочем каталоге, но не проиндексирован, а файл lib/simplegit.rb модифицирован и проиндексирован. Файл Rakefile модифицирован, проиндексирован и ещё раз модифицирован, таким образом на данный момент у него есть те изменения, которые попадут в коммит, и те, которые не попадут.

Игнорирование файлов

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т. п.). В таком случае вы можете создать файл .gitignore. с перечислением шаблонов, соответствующих таким файлам. Вот пример файла .gitignore:

```
$ cat .gitignore
*.[oa]
*~
```

Первая строка предписывает Git игнорировать любые файлы заканчивающиеся на «.o» или «.a» — объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы, заканчивающиеся на тильду (~), которая используется во многих текстовых редакторах, например Emacs, для обозначения временных файлов. Вы можете также включить каталоги log, tmp или pid; автоматически создаваемую документацию; и т. д. и т. п. Хорошая практика заключается в настройке файла .gitignore до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с #, игнорируются.
- Стандартные шаблоны являются глобальными и применяются рекурсивно для всего дерева каталогов.
- Чтобы избежать рекурсии используйте символ слеш (/) в начале шаблона.
- Чтобы исключить каталог добавьте слеш (/) в конец шаблона.
- Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами. Символ (*) соответствует 0 или более символам; последовательность [abc] — любому символу из указанных в скобках (в данном примере a, b или c); знак вопроса (?) соответствует одному символу; и квадратные скобки, в которые заключены символы, разделённые дефисом ([0-9]), соответствуют любому символу из интервала (в данном случае от 0 до 9). Вы также можете использовать две звёздочки, чтобы указать на вложенные каталоги: a/**/z соответствует a/z, a/b/z, a/b/c/z, и так далее.

Вот ещё один пример файла .gitignore:

```
# Исключить все файлы с расширением .a
*.a

# Но отслеживать файл lib.a даже если он подпадает под исключение выше
!lib.a
```

```
# Исключить файл TODO в корневом каталоге, но не файл в subdir/TODO
/TODO

# Игнорировать все файлы в каталоге build/
build/

# Игнорировать файл doc/notes.txt, но не файл doc/server/arch.txt
doc/*.txt

# Игнорировать все .txt файлы в каталоге doc/
doc/**/*.txt
```

Подсказка	GitHub поддерживает довольно полный список примеров .gitignore файлов для множества проектов и языков https://github.com/github/gitignore это может стать отправной точкой для .gitignore в вашем проекте.
Примечание	В простейшем случае репозиторий будет иметь один файл .gitignore в корневом каталоге, правила из которого будут рекурсивно применяться ко всем подкаталогам. Так же возможно использовать .gitignore файлы в подкаталогах. Правила из этих файлов будут применяться только к каталогам, в которых они находятся. Например, репозиторий исходного кода ядра Linux содержит 206 файлов .gitignore. Детальное рассмотрение использования нескольких .gitignore файлов выходит за пределы этой книги; детали доступны в справке man gitignore.

Просмотр индексированных и неиндексированных изменений

Если результат работы команды git status недостаточно информативен для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду git diff. Позже мы рассмотрим команду git diff подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и

собираетесь включить в коммит. Если git status отвечает на эти вопросы в самом общем виде, перечисляя имена файлов, git diff показывает вам непосредственно добавленные и удалённые строки — патч как он есть.

Допустим, вы снова изменили и проиндексировали файл README, а затем изменили файл CONTRIBUTING.md без индексирования. Если вы выполните команду git status, вы опять увидите что-то вроде:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите git diff без аргументов:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
  Please include a nice description of your changes when you submit your PR;
  if we have to read the whole diff to figure out why you're contributing
  in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

  If you are starting to work on a particular area, feel free to submit a PR
```

```
that highlights your work in progress (and note in the PR title that it's
```

Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить `git diff --staged`. Эта команда сравнивает ваши проиндексированные изменения с последним коммитом:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Важно отметить, что `git diff` сама по себе не показывает все изменения, сделанные с последнего коммита — только те, что ещё не проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то `git diff` ничего не вернёт.

Другой пример: вы проиндексировали файл `CONTRIBUTING.md` и затем изменили его, вы можете использовать `git diff` для просмотра как проиндексированных изменений в этом файле, так и тех, что пока не проиндексированы. Если наше окружение выглядит вот так:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
        modified:   CONTRIBUTING.md
```

Используйте `git diff` для просмотра непроиндексированных изменений

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

а также `git diff --cached` для просмотра проиндексированных изменений (`--staged` и `--cached` синонимы):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Примечание	**Git Diff во внешних инструментах** Мы будем продолжать использовать команду `git diff` различными способами на протяжении всей книги. Существует еще один способ просматривать эти изменения, если вы предпочитаете графический просмотр или внешнюю программу просмотра различий, вместо консоли. Выполнив команду `git difftool` вместо `git diff`, вы сможете просмотреть изменения в файле с помощью таких программ как emerge, vimdiff и других (включая коммерческие продукты). Выполните `git difftool --tool-help` чтобы увидеть какие из них уже установлены в вашей системе.

Коммит изменений

Теперь, когда ваш индекс находится в таком состоянии, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили `git add` после редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли `git status`, вы видели, что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать `git commit`:

```
$ git commit
```

Эта команда откроет выбранный вами текстовый редактор.

Примечание	Редактор устанавливается переменной окружения `EDITOR` — обычно это vim или emacs, хотя вы можете установить любой другой с помощью команды `git config --global core.editor`.

В редакторе будет отображён следующий текст (это пример окна Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
```

```
# new file:   README
# modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы команды `git status` и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете.

Примечание	Для ещё более подробного напоминания, что же именно вы поменяли, можете передать аргумент `-v` в команду `git commit`. Это приведёт к тому, что в комментарий будет также помещена дельта/diff изменений, таким образом вы сможете точно увидеть все изменения, которые вы совершили.

Когда вы выходите из редактора, Git создаёт для вас коммит с этим сообщением, удаляя комментарии и вывод команды `diff`.

Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой `commit` указав его после параметра `-m`, как в следующем примере:

```
$ git commit -m "Story 182: fix benchmarks for speed"
[master 463dc4f] Story 182: fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Итак, вы создали свой первый коммит! Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (`master`), какая контрольная сумма SHA-1 у этого коммита (`463dc4f`), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и висит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы

сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Игнорирование индексации

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, Git предоставляет простой способ. Добавление параметра `-a` в команду `git commit` заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без `git add`:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'Add new benchmarks'
[master 83e38c7] Add new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Обратите внимание, что в данном случае перед коммитом вам не нужно выполнять `git add` для файла `CONTRIBUTING.md`, потому что флаг `-a` включает все файлы. Это удобно, но будьте осторожны: флаг `-a` может включить в коммит нежелательные изменения.

Удаление файлов

Для того чтобы удалить файл из Git, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда `git rm`, которая также удаляет файл из вашего рабочего каталога, так что в следующий раз вы не увидите его как «неотслеживаемый».

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции «Changes not staged for commit» (измененные, но не проиндексированные) вывода команды `git status`:

```
$ rm PROJECTS.md
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Затем, если вы выполните команду `git rm`, удаление файла попадёт в индекс:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы должны использовать принудительное удаление с помощью параметра `-f`. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из Git.

Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на жёстком диске, но перестать отслеживать изменения в нём. Это особенно полезно, если вы забыли добавить что-то в файл `.gitignore` и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию `--cached`:

```
$ git rm --cached README
```

В команду `git rm` можно передавать файлы, каталоги или шаблоны. Это означает, что вы можете сделать что-то вроде:

```
$ git rm log/\*.log
```

Обратите внимание на обратный слеш (\) перед *. Он необходим из-за того, что Git использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы, имеющие расширение .log и находящиеся в каталоге log/. Или же вы можете сделать вот так:

```
$ git rm \*~
```

Эта команда удаляет все файлы, имена которых заканчиваются на ~.

Перемещение файлов

В отличие от многих других систем контроля версий, Git не отслеживает перемещение файлов явно. Когда вы переименовываете файл в Git, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения перемещений постфактум — мы рассмотрим обнаружение перемещения файлов чуть позже.

Таким образом, наличие в Git команды mv выглядит несколько странным. Если вам хочется переименовать файл в Git, вы можете сделать что-то вроде:

```
$ git mv file_from file_to
```

и это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на статус, вы увидите, что Git считает, что произошло переименование файла:

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Однако, это эквивалентно выполнению следующих команд:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду mv.

Единственное отличие состоит лишь в том, что `mv` — одна команда вместо трёх — это функция для удобства. Важнее другое — вы можете использовать любой удобный способ для переименования файла, а затем воспользоваться командами add/rm перед коммитом.

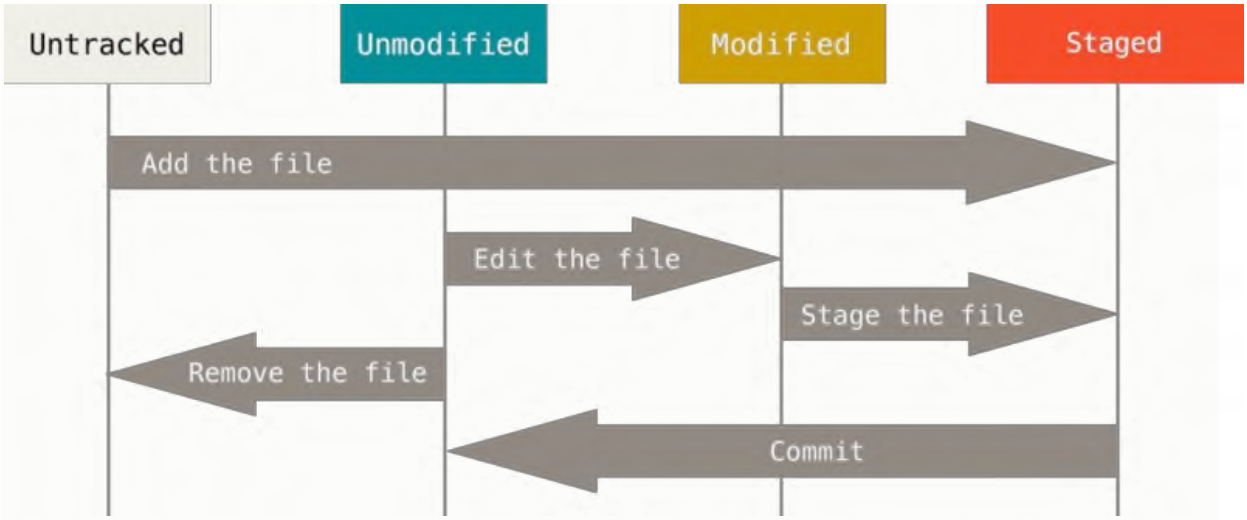
Запись изменений в репозиторий

Итак, у вас имеется настоящий Git-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать «снимки» состояния (snapshots) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы — это те файлы, которые были в последнем снимке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы — это те файлы, о которых знает Git.

Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что Git только что их извлек и вы ничего пока не редактировали.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется.



Жизненный цикл состояний файлов

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда `git status`. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Это означает, что у вас чистый рабочий каталог, другими словами — в нем нет отслеживаемых измененных файлов. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам на какой ветке вы находитесь и сообщает вам, что она не расходится с веткой на сервере. Пока что это всегда ветка `master`, ветка по умолчанию; в этой главе это не важно. В главе Ветвление в Git будут рассмотрены ветки и ссылки более детально.

Предположим, вы добавили в свой проект новый файл, простой файл `README`. Если этого файла раньше не было, и вы выполните `git status`, вы увидите свой неотслеживаемый файл — вот так:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

Понять, что новый файл `README` неотслеживаемый можно по тому, что он находится в секции «Untracked files» в выводе команды `status`. Статус `Untracked` означает, что Git видит файл, которого не было в предыдущем снимке состояния (коммите); Git не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить README, так давайте сделаем это.

Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда `git add`. Чтобы начать отслеживание файла `README`, вы можете выполнить следующее:

```
$ git add README
```

Если вы снова выполните команду `status`, то увидите, что файл `README` теперь отслеживаемый и добавлен в индекс:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

    new file:   README
```

Вы можете видеть, что файл проиндексирован, так как он находится в секции «Changes to be committed». Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды `git add`, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили `git init`, затем вы выполнили `git add` (файлы) — это было сделано для того, чтобы добавить файлы в вашем каталоге под версионный контроль. Команда `git add` принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет все файлы из указанного каталога в индекс.

Индексация изменённых файлов

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл `CONTRIBUTING.md` и после этого снова выполните команду `git status`, то результат будет примерно следующим:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Файл CONTRIBUTING.md находится в секции «Changes not staged for commit» — это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду git add. Это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например для указания файлов с исправленным конфликтом слияния. Вам может быть понятнее, если вы будете думать об этом как «добавить этот контент в следующий коммит», а не как «добавить этот файл в проект». Выполним git add, чтобы проиндексировать CONTRIBUTING.md, а затем снова выполним git status:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в CONTRIBUTING.md до коммита. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним git status:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

```

```
   new file:   README
   modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   CONTRIBUTING.md
```

Что за чёрт? Теперь CONTRIBUTING.md отображается как проиндексированный и непроиндексированный одновременно. Как такое возможно? Такая ситуация наглядно демонстрирует, что Git индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду git add. Если вы выполните коммит сейчас, то файл CONTRIBUTING.md попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду git add , а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения git commit. Если вы изменили файл после выполнения git add, вам придётся снова выполнить git add, чтобы проиндексировать последнюю версию файла:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   README
   modified:   CONTRIBUTING.md
```

Сокращенный вывод статуса

Вывод команды git status довольно всеобъемлющий и многословный. Git также имеет флаг вывода сокращенного статуса, так что вы можете увидеть изменения в более компактном виде. Если вы выполните git status -s или git status --short вы получите гораздо более упрощенный вывод:

```
$ git status -s
 M README
MM Rakefile
```

```
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Новые неотслеживаемые файлы помечены ?? слева от них, файлы добавленные в отслеживаемые помечены A, отредактированные файлы помечены M и так далее. В выводе содержится два столбца — в левом указывается статус файла, а в правом модифицирован ли он после этого. К примеру, в нашем выводе, файл README модифицирован в рабочем каталоге, но не проиндексирован, а файл lib/simplegit.rb модифицирован и проиндексирован. Файл Rakefile модифицирован, проиндексирован и ещё раз модифицирован, таким образом на данный момент у него есть те изменения, которые попадут в коммит, и те, которые не попадут.

Игнорирование файлов

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т. п.). В таком случае вы можете создать файл .gitignore. с перечислением шаблонов соответствующих таким файлам. Вот пример файла .gitignore:

```
$ cat .gitignore
*.[oa]
*~
```

Первая строка предписывает Git игнорировать любые файлы заканчивающиеся на «.o» или «.a» — объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы, заканчивающиеся на тильду (~), которая используется во многих текстовых редакторах, например Emacs, для обозначения временных файлов. Вы можете также включить каталоги log, tmp или pid; автоматически создаваемую документацию; и т. д. и т. п. Хорошая практика заключается в настройке файла .gitignore до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с #, игнорируются.

- Стандартные шаблоны являются глобальными и применяются рекурсивно для всего дерева каталогов.
- Чтобы избежать рекурсии используйте символ слеш (/) в начале шаблона.
- Чтобы исключить каталог добавьте слеш (/) в конец шаблона.
- Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами. Символ (*) соответствует 0 или более символам; последовательность [abc] — любому символу из указанных в скобках (в данном примере a, b или c); знак вопроса (?) соответствует одному символу; и квадратные скобки, в которые заключены символы, разделённые дефисом ([0-9]), соответствуют любому символу из интервала (в данном случае от 0 до 9). Вы также можете использовать две звёздочки, чтобы указать на вложенные каталоги: a/**/z соответствует a/z, a/b/z, a/b/c/z, и так далее.

Вот ещё один пример файла .gitignore:

```
# Исключить все файлы с расширением .a
*.a

# Но отслеживать файл lib.a даже если он подпадает под исключение выше
!lib.a

# Исключить файл TODO в корневом каталоге, но не файл в subdir/TODO
/TODO

# Игнорировать все файлы в каталоге build/
build/

# Игнорировать файл doc/notes.txt, но не файл doc/server/arch.txt
doc/*.txt

# Игнорировать все .txt файлы в каталоге doc/
doc/**/*.txt
```

Подсказка	GitHub поддерживает довольно полный список примеров .gitignore файлов для множества проектов и языков https://github.com/github/gitignore это

	может стать отправной точкой для .gitignore в вашем проекте.
Примечание	В простейшем случае репозиторий будет иметь один файл .gitignore в корневом каталоге, правила из которого будут рекурсивно применяться ко всем подкаталогам. Так же возможно использовать .gitignore файлы в подкаталогах. Правила из этих файлов будут применяться только к каталогам, в которых они находятся. Например, репозиторий исходного кода ядра Linux содержит 206 файлов .gitignore. Детальное рассмотрение использования нескольких .gitignore файлов выходит за пределы этой книги; детали доступны в справке man gitignore.

Просмотр индексированных и неиндексированных изменений

Если результат работы команды git status недостаточно информативен для вас — вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены — вы можете использовать команду git diff. Позже мы рассмотрим команду git diff подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь включить в коммит. Если git status отвечает на эти вопросы в самом общем виде, перечисляя имена файлов, git diff показывает вам непосредственно добавленные и удалённые строки — патч как он есть.

Допустим, вы снова изменили и проиндексировали файл README, а затем изменили файл CONTRIBUTING.md без индексирования. Если вы выполните команду git status, вы опять увидите что-то вроде:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
```

```
 (use "git add <file>..." to update what will be committed)
 (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   CONTRIBUTING.md
```

Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите `git diff` без аргументов:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить `git diff --staged`. Эта команда сравнивает ваши проиндексированные изменения с последним коммитом:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Важно отметить, что `git diff` сама по себе не показывает все изменения, сделанные с последнего коммита — только те, что ещё не

проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то git diff ничего не вернёт.

Другой пример: вы проиндексировали файл CONTRIBUTING.md и затем изменили его, вы можете использовать git diff для просмотра как проиндексированных изменений в этом файле, так и тех, что пока не проиндексированы. Если наше окружение выглядит вот так:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Используйте git diff для просмотра непроиндексированных изменений

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects list](https://github.com/libgit2/libgit2/blob/development/PR
OJECTS.md).
+# test line
```

а также git diff --cached для просмотра проиндексированных изменений (--staged и --cached синонимы):

```
$ git diff --cached
```

```
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Примечание	**Git Diff во внешних инструментах** Мы будем продолжать использовать команду `git diff` различными способами на протяжении всей книги. Существует еще один способ просматривать эти изменения, если вы предпочитаете графический просмотр или внешнюю программу просмотра различий, вместо консоли. Выполнив команду `git difftool` вместо `git diff`, вы сможете просмотреть изменения в файле с помощью таких программ как emerge, vimdiff и других (включая коммерческие продукты). Выполните `git difftool --tool-help` чтобы увидеть какие из них уже установлены в вашей системе.

Коммит изменений

Теперь, когда ваш индекс находится в таком состоянии, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили `git add` после редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли `git status`, вы видели, что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения — это набрать `git commit`:

```
$ git commit
```

Эта команда откроет выбранный вами текстовый редактор.

Примечание	Редактор устанавливается переменной окружения EDITOR — обычно это vim или emacs, хотя вы можете установить любой другой с помощью команды git config --global core.editor, как было показано в главе Введение).

В редакторе будет отображён следующий текст (это пример окна Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#       new file:   README
#       modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы команды git status и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете.

Примечание	Для ещё более подробного напоминания, что же именно вы поменяли, можете передать аргумент -v в команду git commit. Это приведёт к тому, что в комментарий будет также помещена дельта/diff изменений, таким образом вы сможете точно увидеть все изменения, которые вы совершили.

Когда вы выходите из редактора, Git создаёт для вас коммит с этим сообщением, удаляя комментарии и вывод команды diff.

Есть и другой способ — вы можете набрать свой комментарий к коммиту в командной строке вместе с командой commit указав его после параметра -m, как в следующем примере:

```
$ git commit -m "Story 182: fix benchmarks for speed"
[master 463dc4f] Story 182: fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Итак, вы создали свой первый коммит! Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (`master`), какая контрольная сумма SHA-1 у этого коммита (`463dc4f`), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и висит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Игнорирование индексации

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, Git предоставляет простой способ. Добавление параметра `-a` в команду `git commit` заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без `git add`:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'Add new benchmarks'
[master 83e38c7] Add new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Обратите внимание, что в данном случае перед коммитом вам не нужно выполнять git add для файла CONTRIBUTING.md, потому что флаг -a включает все файлы. Это удобно, но будьте осторожны: флаг -a может включить в коммит нежелательные изменения.

Удаление файлов

Для того чтобы удалить файл из Git, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда git rm, которая также удаляет файл из вашего рабочего каталога, так что в следующий раз вы не увидите его как «неотслеживаемый».

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции «Changes not staged for commit» (измененные, но не проиндексированные) вывода команды git status:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Затем, если вы выполните команду git rm, удаление файла попадёт в индекс:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы

должны использовать принудительное удаление с помощью параметра -f. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из Git.

Другая полезная штука, которую вы можете захотеть сделать — это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на жёстком диске, но перестать отслеживать изменения в нём. Это особенно полезно, если вы забыли добавить что-то в файл .gitignore и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию --cached:

```
$ git rm --cached README
```

В команду git rm можно передавать файлы, каталоги или шаблоны. Это означает, что вы можете сделать что-то вроде:

```
$ git rm log/\*.log
```

Обратите внимание на обратный слеш (\) перед *. Он необходим из-за того, что Git использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы, имеющие расширение .log и находящиеся в каталоге log/. Или же вы можете сделать вот так:

```
$ git rm \*~
```

Эта команда удаляет все файлы, имена которых заканчиваются на ~.

Перемещение файлов

В отличие от многих других систем контроля версий, Git не отслеживает перемещение файлов явно. Когда вы переименовываете файл в Git, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован. Однако, Git довольно умён в плане обнаружения перемещений постфактум — мы рассмотрим обнаружение перемещения файлов чуть позже.

Таким образом, наличие в Git команды mv выглядит несколько странным. Если вам хочется переименовать файл в Git, вы можете сделать что-то вроде:

```
$ git mv file_from file_to
```

и это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на статус, вы увидите, что Git считает, что произошло переименование файла:

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Однако, это эквивалентно выполнению следующих команд:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду `mv`. Единственное отличие состоит лишь в том, что `mv` — одна команда вместо трёх — это функция для удобства. Важнее другое — вы можете использовать любой удобный способ для переименования файла, а затем воспользоваться командами add/rm перед коммитом.

Операции отмены

В любой момент вам может потребоваться что-либо отменить. Здесь мы рассмотрим несколько основных способов отмены сделанных изменений. Будьте осторожны, не все операции отмены в свою очередь можно отменить! Это одна из редких областей Git, где неверными действиями можно необратимо удалить результаты своей работы.

Отмена может потребоваться, если вы сделали коммит слишком рано, например, забыв добавить какие-то файлы или комментарий к коммиту. Если вы хотите переделать коммит — внесите необходимые изменения, добавьте их в индекс и сделайте коммит ещё раз, указав параметр `--amend`:

```
$ git commit --amend
```

Эта команда использует область подготовки (индекс) для внесения правок в коммит. Если вы ничего не меняли с момента последнего коммита (например, команда запущена сразу после предыдущего коммита), то снимок состояния останется в точности таким же, а всё что вы сможете изменить — это ваше сообщение к коммиту.

Запустится тот же редактор, только он уже будет содержать сообщение предыдущего коммита. Вы можете редактировать сообщение как обычно, однако, оно заменит сообщение предыдущего коммита.

Например, если вы сделали коммит и поняли, что забыли проиндексировать изменения в файле, который хотели добавить в коммит, то можно сделать следующее:

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```

В итоге получится единый коммит — второй коммит заменит результаты первого.

Примечание	Очень важно понимать, что, когда вы вносите правки в последний коммит, вы не столько исправляете его, сколько **заменяете** новым, который полностью его перезаписывает. В результате всё выглядит так, будто первоначальный коммит никогда не существовал, а также он больше не появится в истории вашего репозитория. Очевидно, смысл изменения коммитов в добавлении незначительных правок в последние коммиты и, при этом, во избежание засорения истории сообщениями вида «Ой, забыл добавить файл» или «Исправление грамматической ошибки».

Отмена индексации файла

Следующие два раздела демонстрируют как работать с индексом и изменениями в рабочем каталоге. Радует, что команда, которой вы определяете состояние этих областей, также подсказывает вам как отменять изменения в них. Например, вы изменили два файла и хотите добавить их в разные коммиты, но случайно выполнили команду `git add *` и добавили в индекс оба. Как исключить из индекса один из них? Команда `git status` напомнит вам:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Прямо под текстом «Changes to be committed» говорится: используйте git reset HEAD <file>… для исключения из индекса. Давайте последуем этому совету и отменим индексирование файла CONTRIBUTING.md:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M       CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Команда выглядит несколько странно, но — работает! Файл CONTRIBUTING.md изменен, но больше не добавлен в индекс.

Примечание	Команда git reset **может** быть опасной если вызвать её с параметром --hard. В приведенном примере файл не был затронут, следовательно команда относительно безопасна.

На текущий момент этот магический вызов — всё, что вам нужно знать о команде git reset.

Отмена изменений в файле

Что делать, если вы поняли, что не хотите сохранять свои изменения файла CONTRIBUTING.md? Как можно просто отменить изменения в нём — вернуть к тому состоянию, которое было в последнем коммите (или к начальному после клонирования, или еще как-то полученному)? Нам повезло, что git status подсказывает и это тоже.

В выводе команды из последнего примера список изменений выглядит примерно так:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Здесь явно сказано, как отменить существующие изменения. Давайте так и сделаем:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

   renamed:    README.md -> README
```

Как видите, откат изменений выполнен.

Важно	Важно понимать, что `git checkout -- <file>` — опасная команда. Все локальные изменения в файле пропадут — Git просто заменит его версией из последнего коммита. Ни в коем случае не используйте эту команду, если вы не уверены, что изменения в файле вам не нужны.

Если вы хотите сохранить изменения в файле, но прямо сейчас их нужно отменить, то есть способы получше, такие как ветвление и припрятывание — мы рассмотрим их в разделе Ветвление в Git.

Помните, все что попало в **коммит** почти всегда Git может восстановить. Можно восстановить даже коммиты из веток, которые были удалены, или коммиты, перезаписанные параметром `–amend`. Но всё, что не было включено в коммит и потеряно — скорее всего, потеряно навсегда.

Отмена действий с помощью git restore

Git версии 2.23.0 представил новую команду: `git restore`. По сути, это альтернатива git reset, которую мы только что рассмотрели. Начиная с версии 2.23.0, Git будет использовать `git restore` вместо `git reset` для многих операций отмены.

Давайте проследим наши шаги и отменим действия с помощью `git restore` вместо `git reset`.

Отмена индексации файла с помощью git restore

В следующих двух разделах показано, как работать с индексом и изменениями рабочей копии с помощью `git restore`. Приятно то, что команда, которую вы используете для определения состояния этих двух областей, также напоминает вам, как отменить изменения в них. Например, предположим, что вы изменили два файла и хотите зафиксировать их как два отдельных изменения, но случайно набираете `git add *` и индексируете их оба. Как вы можете убрать из индекса один из двух? Команда `git status` напоминает вам:

```
$ git add *
$ git status
On branch master
Changes to be committed:
 (use "git restore --staged <file>..." to unstage)
```

```
        modified:   CONTRIBUTING.md
        renamed:    README.md -> README
```

Прямо под текстом «Changes to be committed», написано использовать git restore --staged <file> … для отмены индексации файла. Итак, давайте воспользуемся этим советом, чтобы убрать из индекса файл CONTRIBUTING.md:

```
$ git restore --staged CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
 (use "git restore --staged <file>..." to unstage)
        renamed:    README.md -> README

Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
        modified:   CONTRIBUTING.md
```

Файл CONTRIBUTING.md изменен, но снова не индексирован.

Откат измененного файла с помощью git restore

Что, если вы поймете, что не хотите сохранять изменения в файле CONTRIBUTING.md? Как легко его откатить — вернуть обратно к тому, как он выглядел при последнем коммите (или изначально клонирован, или каким-либо образом помещён в рабочий каталог)? К счастью, git status тоже говорит, как это сделать. В выводе последнего примера неиндексированная область выглядит следующим образом:

```
Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
        modified:   CONTRIBUTING.md
```

Он довольно недвусмысленно говорит, как отменить сделанные вами изменения. Давайте сделаем то, что написано:

```
$ git restore CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
 (use "git restore --staged <file>..." to unstage)
        renamed:    README.md -> README
```

Важно	Важно понимать, что git restore <file> — опасная команда. Любые локальные изменения, внесенные в этот файл, исчезнут — Git просто заменит файл последней зафиксированной версией. Никогда не используйте эту команду, если

	точно не знаете, нужны ли вам эти несохраненные локальные изменения.

Работа с удалёнными репозиториями

Для того, чтобы внести вклад в какой-либо Git-проект, вам необходимо уметь работать с удалёнными репозиториями. Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее. В данном разделе мы рассмотрим некоторые из этих навыков.

Примечание	**Удаленный репозиторий может находиться на вашем локальном компьютере.** Вполне возможно, что удалённый репозиторий будет находиться на том же компьютере, на котором работаете вы. Слово «удалённый» не означает, что репозиторий обязательно должен быть где-то в сети или Интернет, а значит только — где-то ещё. Работа с таким удалённым репозиторием подразумевает выполнение стандартных операций отправки и получения, как и с любым другим удалённым репозиторием.

Просмотр удалённых репозиториев

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду `git remote`. Она выведет названия доступных удалённых репозиториев. Если вы клонировали репозиторий, то увидите, как минимум `origin` — имя по умолчанию, которое Git даёт серверу, с которого производилось клонирование:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
```

```
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Вы можете также указать ключ -v, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию:

```
$ git remote -v
origin      https://github.com/schacon/ticgit (fetch)
origin      https://github.com/schacon/ticgit (push)
```

Если у вас больше одного удалённого репозитория, команда выведет их все. Например, для репозитория с несколькими настроенными удалёнными репозиториями в случае совместной работы нескольких пользователей, вывод команды может выглядеть примерно так:

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Это означает, что мы можем легко получить изменения от любого из этих пользователей. Возможно, что некоторые из репозиториев доступны для записи и в них можно отправлять свои изменения, хотя вывод команды не даёт никакой информации о правах доступа.

Добавление удалённых репозиториев

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториев, сейчас рассмотрим эту операцию подробнее. Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), просто выполните команду git remote add <shortname> <url>:

```
$ git remote
origin
```

```
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin	https://github.com/schacon/ticgit (fetch)
origin	https://github.com/schacon/ticgit (push)
pb	https://github.com/paulboone/ticgit (fetch)
pb	https://github.com/paulboone/ticgit (push)
```

Теперь вместо указания полного пути вы можете использовать pb. Например, если вы хотите получить изменения, которые есть у Пола, но нету у вас, вы можете выполнить команду git fetch pb:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Ветка master из репозитория Пола сейчас доступна вам под именем pb/master. Вы можете слить её с одной из ваших веток или переключить на неё локальную ветку, чтобы просмотреть содержимое ветки Пола.

Получение изменений из удалённого репозитория — Fetch и Pull

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

```
$ git fetch [remote-name]
```

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта, которые вы можете просмотреть или слить в любой момент.

Когда вы клонируете репозиторий, команда clone автоматически добавляет этот удалённый репозиторий под именем «origin». Таким образом, git fetch origin извлекает все наработки, отправленные на этот сервер после того, как вы его клонировали (или получили изменения с помощью fetch). Важно отметить, что команда git fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный

момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если ветка настроена на отслеживание удалённой ветки, то вы можете использовать команду `git pull` чтобы автоматически получить изменения из удалённой ветки и слить их со своей текущей. Этот способ может для вас оказаться более простым или более удобным. К тому же, по умолчанию команда `git clone` автоматически настраивает вашу локальную ветку `master` на отслеживание удалённой ветки `master` на сервере, с которого вы клонировали репозиторий. Название веток может быть другим и зависит от ветки по умолчанию на сервере. Выполнение `git pull`, как правило, извлекает (fetch) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

Примечание	Начиная с версии 2.27, команда `git pull` выдаёт предупреждение, если настройка `pull.rebase` не установлена. Git будет выводить это предупреждение каждый раз пока настройка не будет установлена. Если хотите использовать поведение Git по умолчанию (простое смещение вперёд если возможно — иначе создание коммита слияния): `git config --global pull.rebase "false"` Если хотите использовать перебазирование при получении изменений: `git config --global pull.rebase "true"`

Отправка изменений в удаленный репозиторий (Push)

Когда вы хотите поделиться своими наработками, вам необходимо отправить их в удалённый репозиторий. Команда для этого действия простая: `git push <remote-name> <branch-name>`. Чтобы отправить вашу ветку master на сервер `origin` (повторимся, что клонирование обычно настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки ваших коммитов:

```
$ git push origin master
```

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду `push`. Если вы и кто-то ещё одновременно

клонируете, затем он выполняет команду push, а после него выполнить команду push попытаетесь вы, то ваш push точно будет отклонён. Вам придётся сначала получить изменения и объединить их с вашими и только после этого вам будет позволено выполнить push. Обратитесь к главе Ветвление в Git для более подробного описания, как отправлять изменения на удалённый сервер.

Просмотр удаленного репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду git remote show <remote>. Выполнив эту команду с некоторым именем, например, origin, вы получите следующий результат:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                       tracked
    dev-branch                   tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Она выдаёт URL удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что, если вы, находясь на ветке master, выполните git pull, ветка master с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Это был пример для простой ситуации, и вы наверняка встречались с чем-то подобным. Однако, если вы используете Git более интенсивно, вы можете увидеть гораздо большее количество информации от git remote show:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
```

```
    Push  URL: https://github.com/my-org/complex-project
    HEAD branch: master
    Remote branches:
      master                tracked
      dev-branch            tracked
      markdown-strip        tracked
      issue-43              new (next fetch will store in remotes/origin)
      issue-45              new (next fetch will store in remotes/origin)
      refs/remotes/origin/issue-11  stale (use 'git remote prune' to remove)
    Local branches configured for 'git pull':
      dev-branch merges with remote dev-branch
      master     merges with remote master
    Local refs configured for 'git push':
      dev-branch            pushes to dev-branch            (up to date)
      markdown-strip        pushes to markdown-strip        (up to date)
      master                pushes to master                (up to date)
```

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении `git push`. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере, и для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении `git pull`.

Удаление и переименование удалённых репозиториев

Для переименования удалённого репозитория можно выполнить `git remote rename`. Например, если вы хотите переименовать `pb` в `paul`, вы можете это сделать при помощи `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Стоит упомянуть, что это также изменит имена удалённых веток в вашем репозитории. То, к чему вы обращались как `pb/master`, теперь стало `paul/master`.

Если по какой-то причине вы хотите удалить удаленный репозиторий — вы сменили сервер или больше не используете определённое зеркало, или кто-то перестал вносить изменения — вы можете использовать `git remote rm`:

```
$ git remote remove paul
$ git remote
origin
```

При удалении ссылки на удалённый репозиторий все отслеживаемые ветки и настройки, связанные с этим репозиторием, так же будут удалены.

Работа с тегами

Как и большинство СКВ, Git имеет возможность помечать определённые моменты в истории как важные. Как правило, эта функциональность используется для отметки моментов выпуска версий (v1.0, и т. п.). Такие пометки в Git называются тегами. В этом разделе вы узнаете, как посмотреть имеющиеся теги, как создать новые или удалить существующие, а также какие типы тегов существуют в Git.

Просмотр списка тегов

Просмотреть список имеющихся тегов в Git можно очень просто. Достаточно набрать команду `git tag` (параметры `-l` и `--list` опциональны):

```
$ git tag
v1.0
v2.0
```

Данная команда перечисляет теги в алфавитном порядке; порядок их отображения не имеет существенного значения.

Так же можно выполнить поиск тега по шаблону. Например, репозиторий Git содержит более 500 тегов. Если вы хотите посмотреть теги выпусков 1.8.5, то выполните следующую команду:

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Примечание	**Для отображения тегов, согласно шаблону, требуются параметры -l или --list** Если вы хотите посмотреть весь список тегов, запуск команды git tag неявно подразумевает это и выводит полный список; использование параметров -l или --list в этом случае опционально. Если вы хотите отфильтровать список тегов согласно шаблону, использование параметров -l или --list становится обязательным.

Создание тегов

Git использует два основных типа тегов: легковесные и аннотированные.

Легковесный тег — это что-то очень похожее на ветку, которая не изменяется — просто указатель на определённый коммит.

А вот аннотированные теги хранятся в базе данных Git как полноценные объекты. Они имеют контрольную сумму, содержат имя автора, его e-mail и дату создания, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные теги, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные.

Аннотированные теги

Создание аннотированного тега в Git выполняется легко. Самый простой способ — это указать -a при выполнении команды tag:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

Опция -m задаёт сообщение, которое будет храниться вместе с тегом. Если не указать сообщение, то Git запустит редактор, чтобы вы смогли его ввести.

С помощью команды git show вы можете посмотреть данные тега вместе с коммитом:

```
$ git show v1.4
tag v1.4
```

```
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Здесь приведена информация об авторе тега, дате его создания и аннотирующее сообщение перед информацией о коммите.

Легковесные теги

Легковесный тег — это ещё один способ пометить коммит. По сути, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесного тега не передавайте опций `-a`, `-s` и `-m`, укажите только название:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

На этот раз при выполнении `git show` для этого тега вы не увидите дополнительной информации. Команда просто покажет коммит:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Отложенная расстановка тегов

Также возможно помечать уже пройденные коммиты. Предположим, история коммитов выглядит следующим образом:

```
$ git log --pretty=oneline
```

```
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

Теперь предположим, что вы забыли отметить версию проекта v1.2, которая была там, где находится коммит «Update rakefile». Вы можете добавить тег и позже. Для отметки коммита укажите его контрольную сумму (или её часть) как параметр команды:

```
$ git tag -a v1.2 9fceb02
```

Проверим, что коммит отмечен:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    Update rakefile
```

```
...
```

Обмен тегами

По умолчанию, команда `git push` не отправляет теги на удалённые сервера. После создания теги нужно отправлять явно на удалённый сервер. Процесс аналогичен отправке веток — достаточно выполнить команду `git push origin <tagname>`.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

Если у вас много тегов, и вам хотелось бы отправить все за один раз, то можно использовать опцию `--tags` для команды `git push`. В таком случае все ваши теги отправятся на удалённый сервер (если только их уже там нет).

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

Теперь, если кто-то клонирует (clone) или выполнит `git pull` из вашего репозитория, то он получит вдобавок к остальному и ваши метки.

Примечание	**`git push` отправляет оба типа тегов** Отправка тегов командой `git push <remote> --tags` не различает аннотированные и легковесные теги. В настоящее время не существует опции чтобы отправить только лёгковесные теги, но если использовать команду `git push <remote> --follow-tags`, то отправятся только аннотированные теги.

Удаление тегов

Для удаления тега в локальном репозитории достаточно выполнить команду `git tag -d <tagname>`. Например, удалить созданный ранее легковесный тег можно следующим образом:

```
$ git tag -d v1.4-lw
Deleted tag 'v1.4-lw' (was e7d5add)
```

Обратите внимание, что при удалении тега не происходит его удаления с внешних серверов. Существует два способа изъятия тега из внешнего репозитория.

Первый способ — это выполнить команду `git push <remote> :refs/tags/<tagname>`:

```
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
 - [deleted]         v1.4-lw
```

Это следует понимать как обновление внешнего тега пустым значением, что приводит к его удалению.

Второй способ убрать тег из внешнего репозитория более интуитивный:

```
$ git push origin --delete <tagname>
```

Переход на тег

Если вы хотите получить версии файлов, на которые указывает тег, то вы можете сделать `git checkout` для тега. Однако, это переведёт репозиторий в состояние «detached HEAD», которое имеет ряд неприятных побочных эффектов.

```
$ git checkout v2.0.0
Note: switching to 'v2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in
this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>
```

```
Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout v2.0-beta-0.1
Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-final
HEAD is now at df3f601... Add atlas.json and cover image
```

Если в состоянии «detached HEAD» внести изменения и сделать коммит, то тег не изменится, при этом новый коммит не будет относиться ни к какой из веток, а доступ к нему можно будет получить только по его хешу. Поэтому, если вам нужно внести изменения — исправить ошибку в одной из старых версий — скорее всего вам следует создать ветку:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Если сделать коммит в ветке `version2`, то она сдвинется вперед и будет отличаться от тега `v2.0.0`, так что будьте с этим осторожны.

Псевдонимы в Git

Прежде, чем закончить эту главу по основам Git, рассмотрим ещё одну маленькую хитрость, которая поможет сделать использование Git проще, легче, и более привычным: псевдонимы (aliases). Мы не будем ссылаться на них дальше или предполагать, что вы будете пользоваться ими по ходу чтения книги, но вам лучше было бы знать, как их использовать.

Git не будет пытаться сделать вывод о том, какую команду вы хотели ввести, если вы ввели её не полностью. Если вы не хотите печатать каждую команду для Git целиком, вы легко можете настроить псевдонимы (alias) для любой команды с помощью `git config`. Вот несколько примеров псевдонимов, которые вы, возможно, захотите задать:

```
$ git config --global alias.co checkout

$ git config --global alias.br branch
```

```
$ git config --global alias.ci commit

$ git config --global alias.st status
```

Это означает, что, например, вместо ввода `git commit`, вам достаточно набрать только `git ci`. По мере освоения Git вам, вероятно, придётся часто пользоваться и другими командами. В этом случае без колебаний создавайте новые псевдонимы.

Такой способ может также быть полезен для создания команд, которые, как вы думаете, должны существовать. Например, чтобы исправить неудобство, с которым мы столкнулись при исключении файла из индекса, можно добавить в Git свой собственный псевдоним `unstage`:

```
$ git config --global alias.unstage 'reset HEAD --'
```

Это делает эквивалентными следующие две команды:

```
$ git unstage fileA

$ git reset HEAD -- fileA
```

Такой вариант кажется немного более понятным. Также, обычно, добавляют команду `last` следующим образом:

```
$ git config --global alias.last 'log -1 HEAD'
```

Таким образом, можно легко просмотреть последний коммит:

```
$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date:   Tue Aug 26 19:48:51 2008 +0800


    Test for current head


    Signed-off-by: Scott Chacon <schacon@example.com>
```

Можно сказать, что Git просто заменяет эти команды на созданные вами псевдонимы (alias). Однако, возможно, вы захотите выполнить внешнюю команду, а не подкоманду Git. В этом случае следует начать команду с символа `!`. Это полезно, если вы пишете свои утилиты для работы

с Git-репозиторием. Продемонстрируем этот случай на примере создания псевдонима git visual для запуска gitk:

```
$ git config --global alias.visual '!gitk'
```

4. Ветвления в Git. Хранилища репозиториев

Почти каждая система контроля версий (СКВ) в какой-то форме поддерживает ветвление. Используя ветвление, Вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию. Во многих СКВ создание веток — это очень затратный процесс, часто требующий создания новой копии каталога с исходным кодом, что может занять много времени для большого проекта.

Некоторые люди, говоря о модели ветвления Git, называют ее «киллер-фича», что выгодно выделяет Git на фоне остальных СКВ. Что в ней такого особенного? Ветвление Git очень легковесно: операция создания ветки выполняется почти мгновенно, переключение между ветками туда-сюда, обычно, также быстро. В отличие от многих других СКВ, Git поощряет процесс работы, при котором ветвление и слияние выполняется часто, даже по несколько раз в день. Понимание и владение этой функциональностью дает вам уникальный и мощный инструмент, который может полностью изменить привычный процесс разработки.

О ветвлении в двух словах

Для точного понимания механизма ветвлений, необходимо вернуться назад и изучить то, как Git хранит данные.

Как вы можете помнить из Что такое Git?, Git не хранит данные в виде последовательности изменений, он использует набор снимков (snapshot).

Когда вы делаете коммит, Git сохраняет его в виде объекта, который содержит указатель на снимок (snapshot) подготовленных данных. Этот объект так же содержит имя автора и e-mail, сообщение и указатель на коммит или коммиты непосредственно предшествующие данному (его родителей): отсутствие родителя для первоначального коммита, один родитель для обычного коммита, и несколько родителей для результатов слияния двух и более веток.

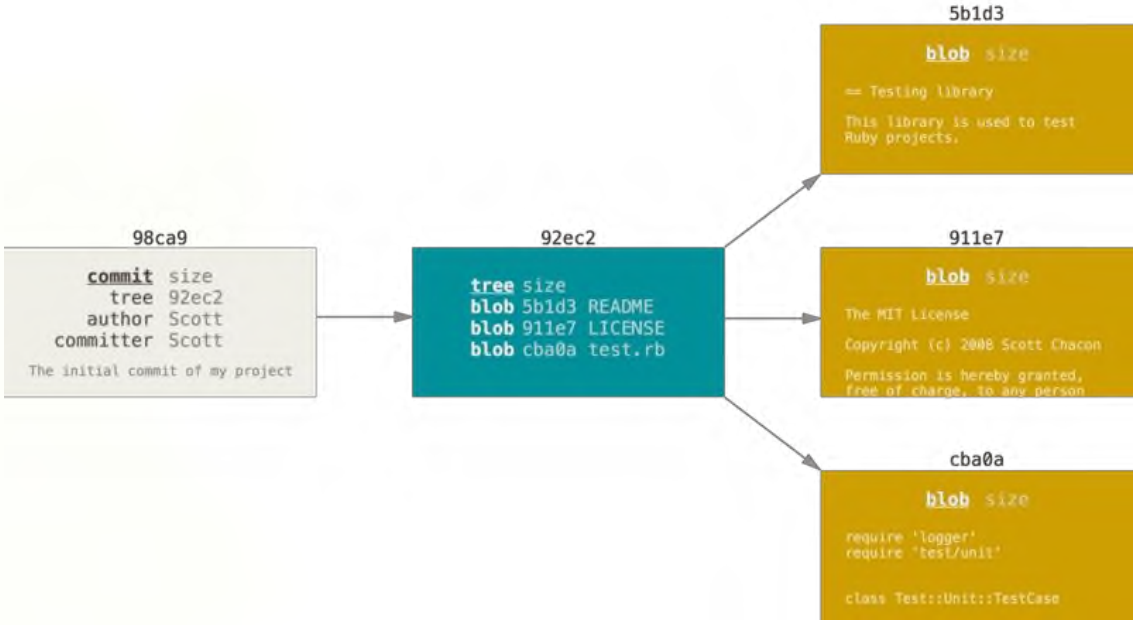
Предположим, у вас есть каталог с тремя файлами, и вы добавляете их все в индекс и создаёте коммит. Во время индексации вычисляется контрольная сумма каждого файла (SHA-1 как мы узнали из... Что такое Git?), затем каждый файл сохраняется в репозиторий (Git называет такой файл **блоб** — большой бинарный объект), а контрольная сумма попадёт в индекс:

```
$ git add README test.rb LICENSE
```

```
$ git commit -m 'Initial commit'
```

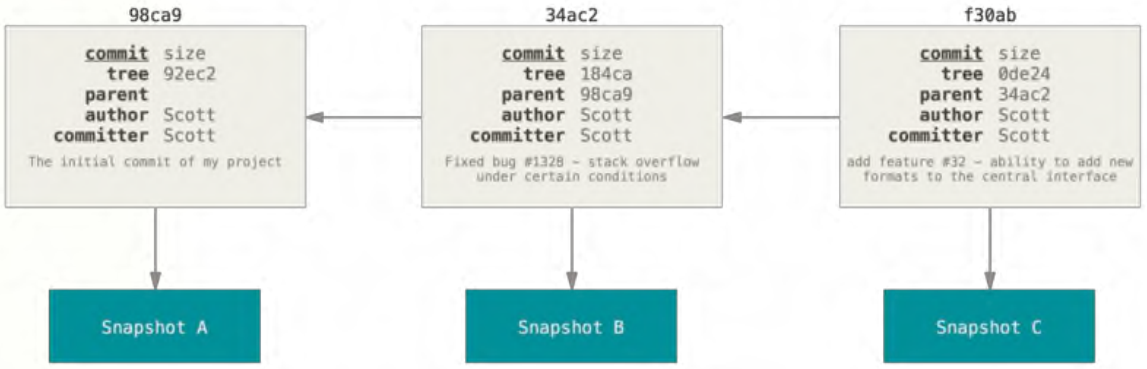
Когда вы создаёте коммит командой git commit, Git вычисляет контрольные суммы каждого подкаталога (в нашем случае, только основной каталог проекта) и сохраняет его в репозитории как объект дерева каталогов. Затем Git создаёт объект коммита с метаданными и указателем на основное дерево проекта для возможности воссоздать этот снимок в случае необходимости.

Ваш репозиторий Git теперь хранит пять объектов: три блоб объекта (по одному на каждый файл), объект **дерева** каталогов, содержащий список файлов и соответствующих им блобов, а также объект **коммита**, содержащий метаданные и указатель на объект дерева каталогов.



Коммит и его дерево

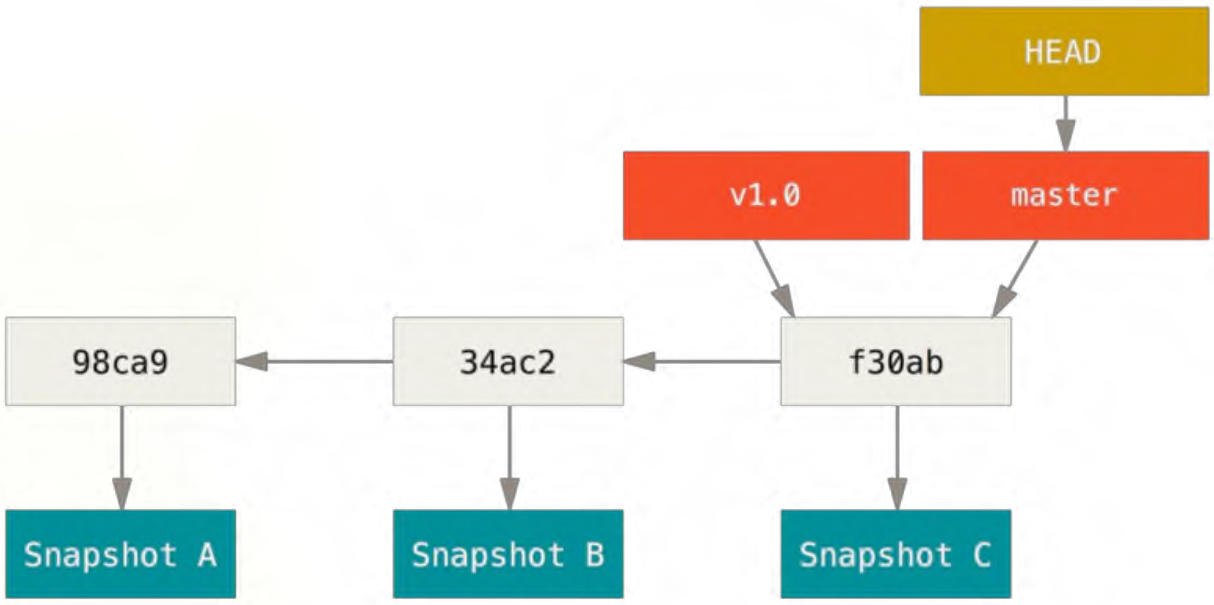
Если вы сделаете изменения и создадите ещё один коммит, то он будет содержать указатель на предыдущий коммит.



Коммит и его родители

Ветка в Git — это простой перемещаемый указатель на один из таких коммитов. По умолчанию, имя основной ветки в Git — master. Как только вы начнёте создавать коммиты, ветка master будет всегда указывать на последний коммит. Каждый раз при создании коммита указатель ветки master будет передвигаться на следующий коммит автоматически.

Примечание	Ветка «master» в Git — это не какая-то особенная ветка. Она точно такая же, как и все остальные ветки. Она существует почти во всех репозиториях только лишь потому, что её создаёт команда git init, а большинство людей не меняют её название.



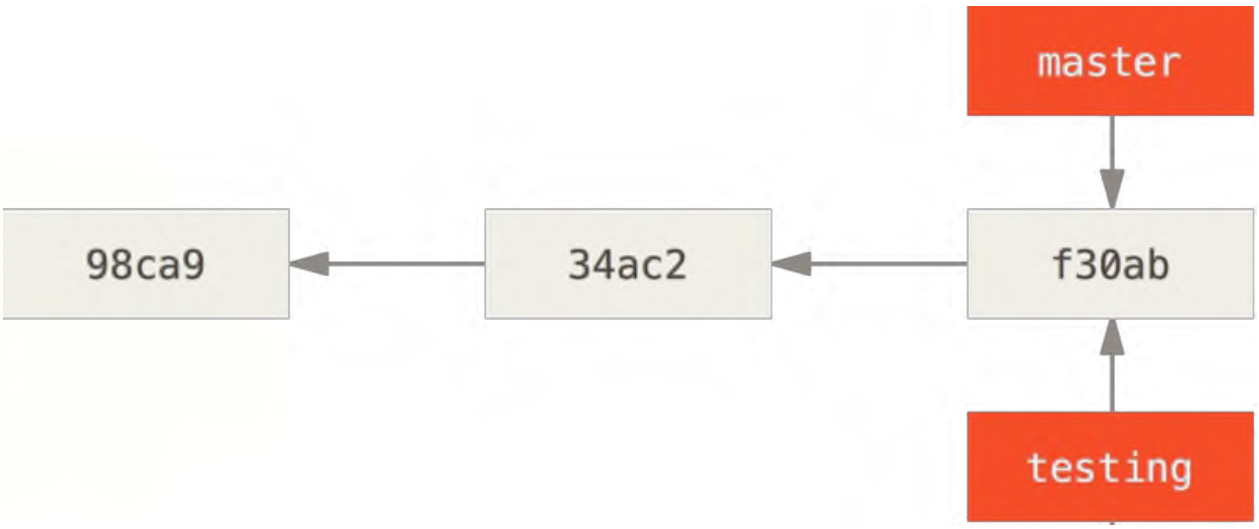
Ветка и история коммитов

Создание новой ветки

Что же на самом деле происходит при создании ветки? Всего лишь создаётся новый указатель для дальнейшего перемещения. Допустим вы хотите создать новую ветку с именем testing. Вы можете это сделать командой git branch :

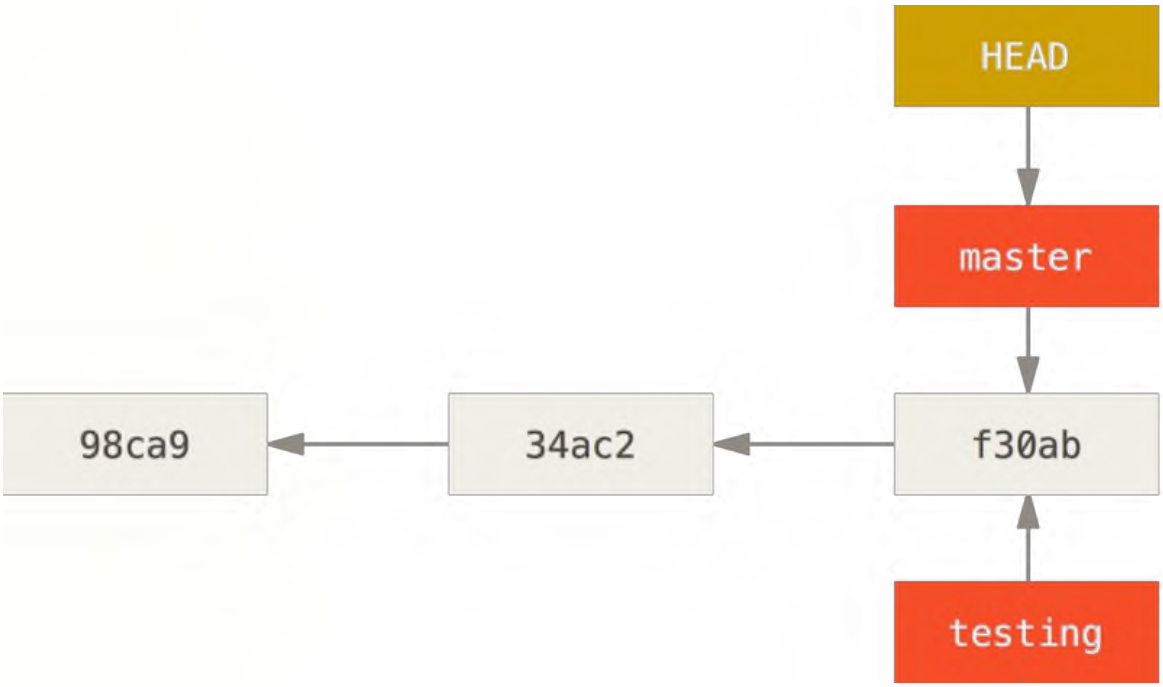
```
$ git branch testing
```

В результате создаётся новый указатель на текущий коммит.



Две ветки указывают на одну и ту же последовательность коммитов

Как Git определяет, в какой ветке вы находитесь? Он хранит специальный указатель HEAD. Имейте в виду, что в Git концепция HEAD значительно отличается от других систем контроля версий, которые вы могли использовать раньше (Subversion или CVS). В Git — это указатель на текущую локальную ветку. В нашем случае мы все еще находимся в ветке master. Команда git branch только **создаёт** новую ветку, но не переключает на неё.



HEAD указывает на ветку

Вы можете легко это увидеть при помощи простой команды git log, которая покажет вам куда указывают указатели веток. Эта опция называется --decorate.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

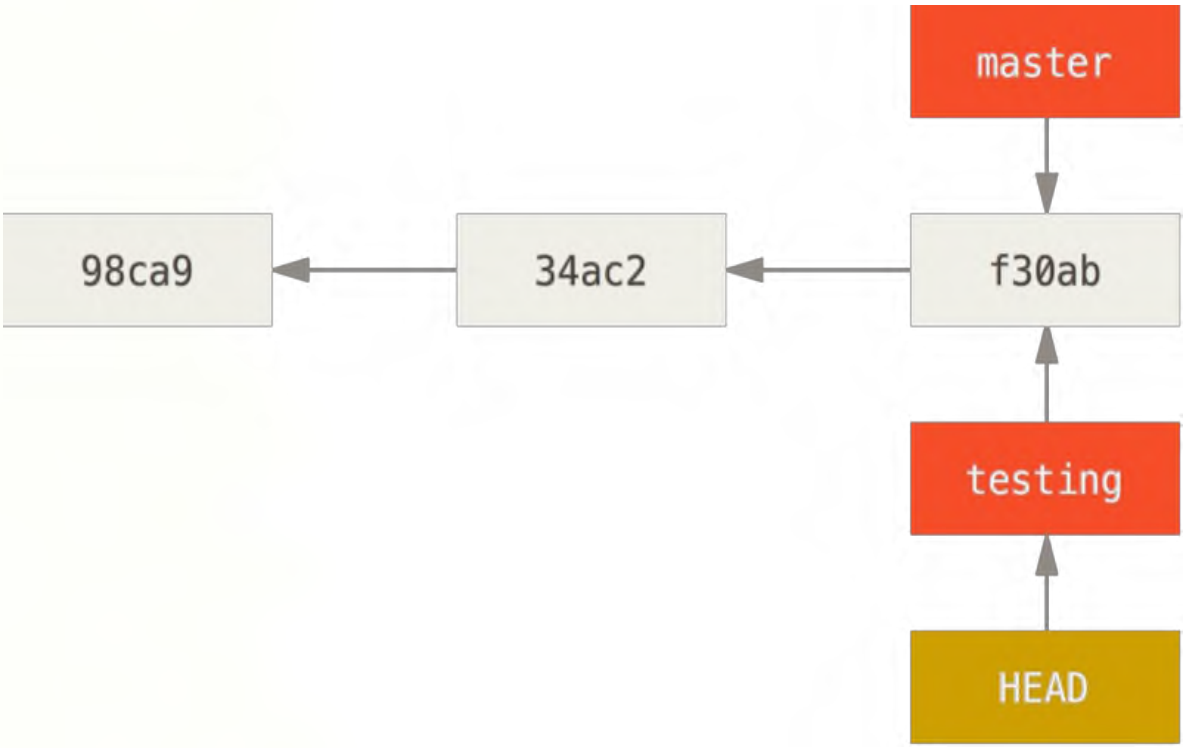
Здесь можно увидеть указывающие на коммит f30ab ветки: master и testing.

Переключение веток

Для переключения на существующую ветку выполните команду git checkout. Давайте переключимся на ветку testing:

```
$ git checkout testing
```

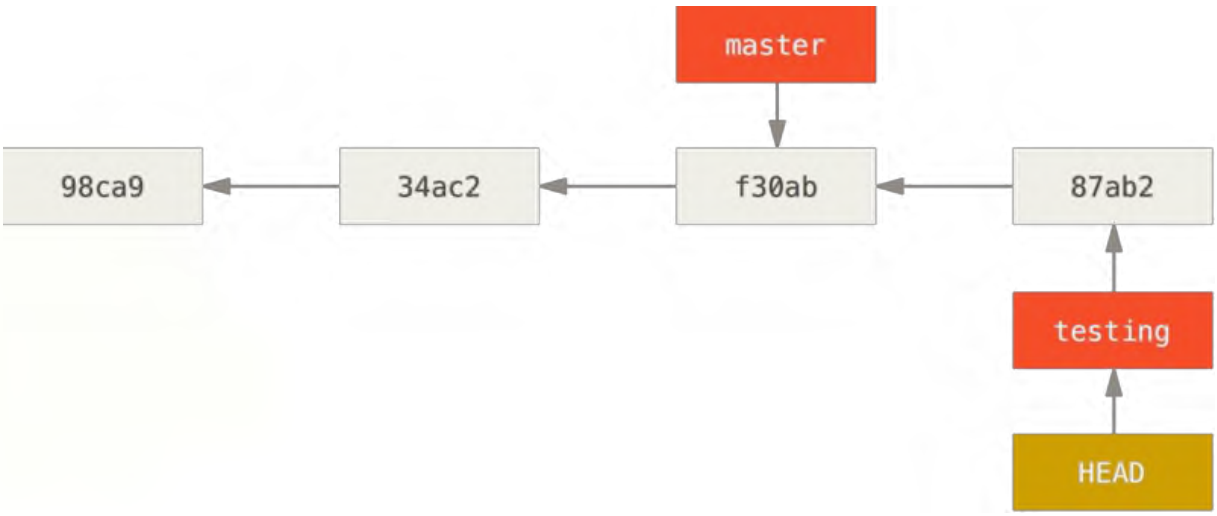
В результате указатель HEAD переместится на ветку testing.



HEAD указывает на текущую ветку

Какой в этом смысл? Давайте сделаем ещё один коммит:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

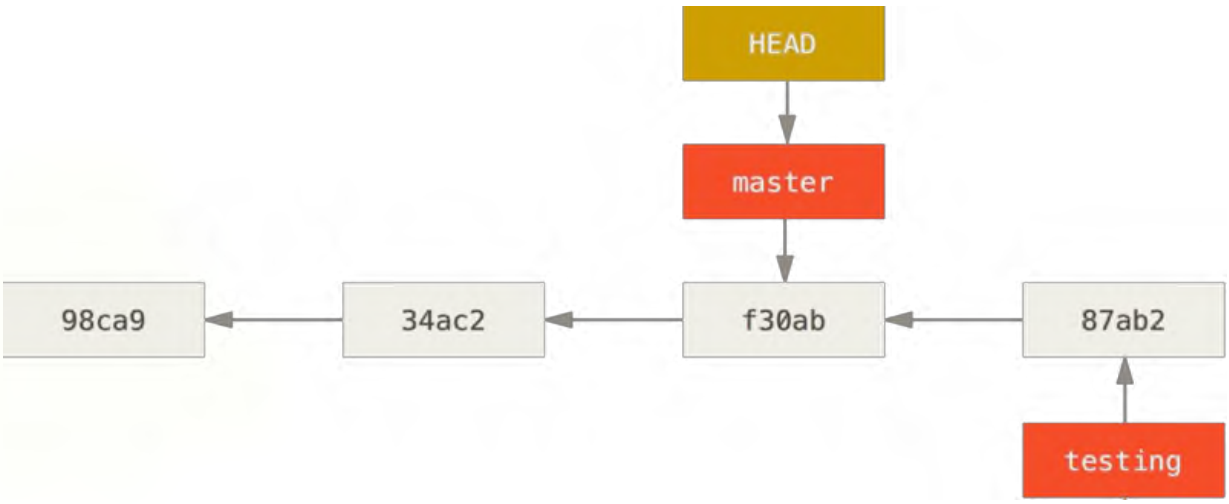


Указатель на ветку HEAD переместился вперёд после коммита

Интересная ситуация: указатель на ветку testing переместился вперёд, а master указывает на тот же коммит, где вы были до переключения веток командой git checkout. Давайте переключимся назад на ветку master:

```
$ git checkout master
```

Примечание	**git log не показывает все ветки по умолчанию** Если выполнить команду git log прямо сейчас, то в её выводе только что созданная ветка «testing» фигурировать не будет. Ветка никуда не исчезла; просто Git не знает, что именно она вас интересует, и выводит наиболее полезную, по его мнению, информацию. Другими словами, по умолчанию git log отобразит историю коммитов только для текущей ветки. Для просмотра истории коммитов другой ветки необходимо явно указать её имя: `Git Log Testing` Чтобы посмотреть историю по всем веткам — выполните команду с дополнительным флагом: `git log --all`.



HEAD перемещается, когда вы делаете checkout

Эта команда сделала две вещи: переместила указатель HEAD назад на ветку master и вернула файлы в рабочем каталоге в то состояние, на снимок которого указывает master. Это также означает, что все вносимые с этого момента изменения будут относиться к старой версии проекта. Другими словами, вы откатили все изменения ветки testing и можете продолжать в другом направлении.

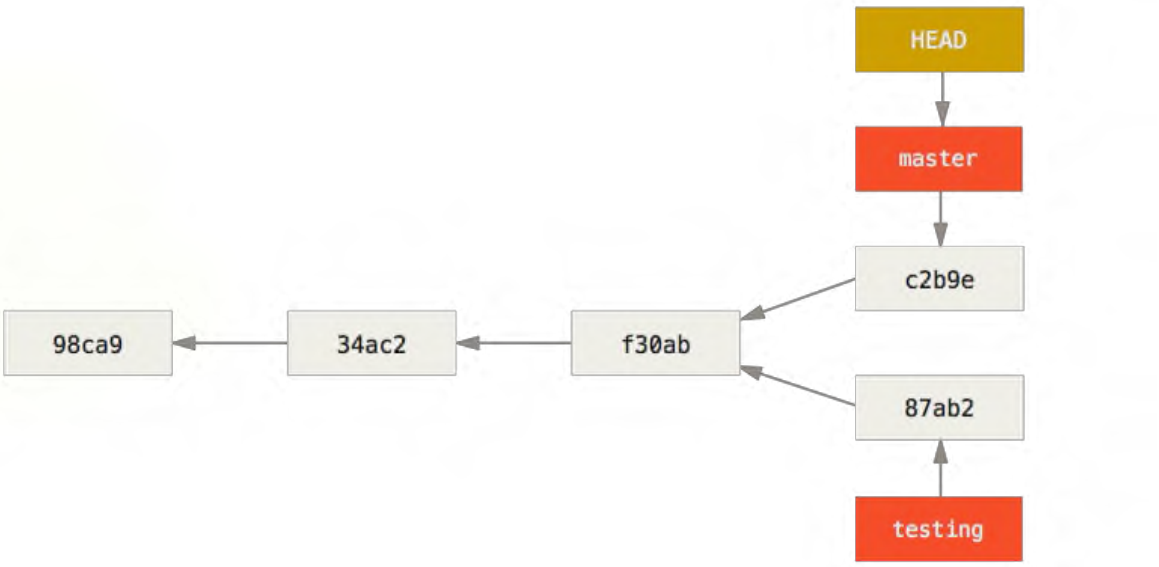
Примечание	**Переключение веток меняет файлы в рабочем каталоге** Важно запомнить, что при переключении веток в Git происходит изменение файлов в рабочем каталоге. Если вы переключаетесь на старую ветку, то рабочий каталог будет выглядеть так же, как выглядел на момент последнего коммита в ту ветку. Если Git по каким-то причинам не может этого сделать — он не позволит вам переключиться вообще.

Давайте сделаем еще несколько изменений и создадим очередной коммит:

```
$ vim test.rb

$ git commit -a -m 'made other changes'
```

Теперь история вашего проекта разошлась. Вы создали ветку и переключились на нее, поработали, а затем вернулись в основную ветку и поработали в ней. Эти изменения изолированы друг от друга: вы можете свободно переключаться туда и обратно, а когда понадобится — объединить их. И все это делается простыми командами: branch, checkout и commit.



Разветвлённая история

Все описанные действия можно визуализировать с помощью команды git log. Для отображения истории коммитов, текущего положения указателей веток и истории ветвления выполните команду git log --oneline --decorate --graph --all.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) Made other changes
| * 87ab2 (testing) Made a change
|/
* f30ab Add feature #32 - ability to add new formats to the central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Ветка в Git — это простой файл, содержащий 40 символов контрольной суммы SHA-1 коммита, на который она указывает; поэтому операции с ветками являются дешёвыми с точки зрения потребления ресурсов или времени. Создание новой ветки в Git происходит так же быстро и просто как запись 41 байта в файл (40 знаков и перевод строки).

Это принципиально отличает процесс ветвления в Git от более старых систем контроля версий, где все файлы проекта копируются в другой подкаталог. В зависимости от размера проекта, операции ветвления в таких системах могут занимать секунды или даже минуты, когда в Git эти операции мгновенны. Поскольку при коммите мы сохраняем указатель на родительский коммит, то поиск подходящей базы для слияния веток делается автоматически и, в большинстве случаев, очень прост. Эти возможности побуждают разработчиков чаще создавать и использовать ветки.

Давайте посмотрим, почему и вам имеет смысл делать так же.

	Одновременное создание новой ветки и переключение на неё
Примечание	Как правило, при создании новой ветки вы хотите сразу на неё переключиться — это можно сделать, используя команду `git checkout -b <newbranchname>`.
Примечание	Начиная с Git версии 2.23, вы можете использовать `git switch` вместо `git checkout`, чтобы: • Переключиться на существующую ветку: `git switch testing-branch`. • Создать новую ветку и переключиться на нее: `git switch -c new-branch`. Флаг `-c` означает создание, но также можно использовать полный формат: ` --create`. • Вернуться к предыдущей извлечённой ветке: `git switch -`.

Основы ветвления и слияния

Давайте рассмотрим простой пример рабочего процесса, который может быть полезен в вашем проекте. Ваша работа построена так:

1. Вы работаете над сайтом.
2. Вы создаете ветку для новой статьи, которую вы пишете.
3. Вы работаете в этой ветке.

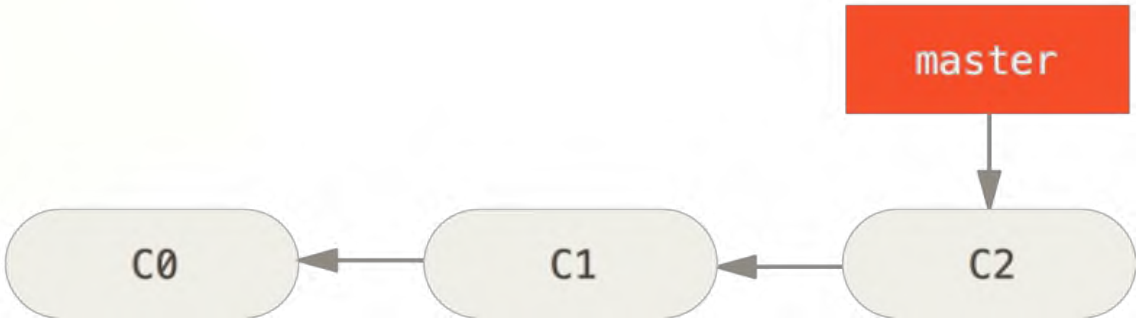
В этот момент вы получаете сообщение, что обнаружена критическая ошибка, требующая скорейшего исправления. Ваши действия:

1. Переключиться на основную ветку.
2. Создать ветку для добавления исправления.
3. После тестирования слить ветку, содержащую исправление с основной веткой.

4. Переключиться назад в ту ветку, где вы пишете статью и продолжить работать.

Основы ветвления

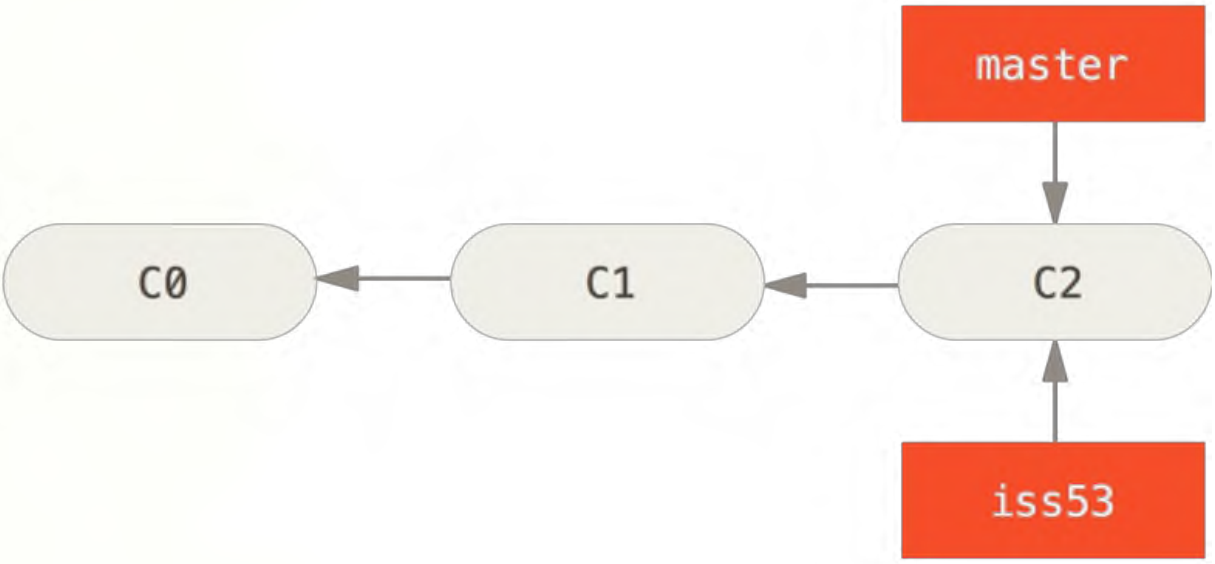
Предположим, вы работаете над проектом и уже имеете несколько коммитов.



Простая история коммитов

Вы решаете, что теперь вы будете заниматься проблемой #53 из вашей системы отслеживания ошибок. Чтобы создать ветку и сразу переключиться на нее, можно выполнить команду `git checkout` с параметром `-b`:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
Это то же самое что и:
$ git branch iss53
$ git checkout iss53
```

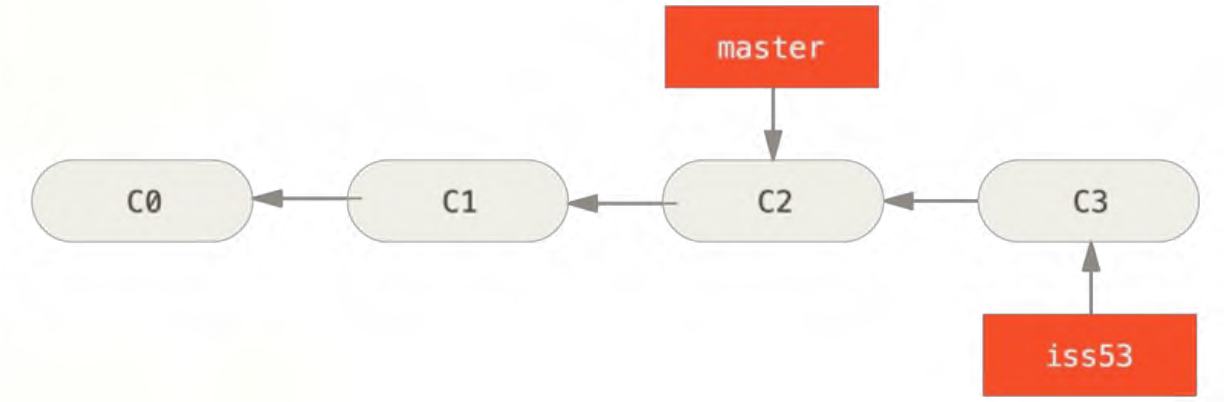


Создание нового указателя ветки

Вы работаете над своим сайтом и делаете коммиты. Это приводит к тому, что ветка iss53 движется вперед, так как вы переключились на нее ранее (HEAD указывает на нее).

```
$ vim index.html

$ git commit -a -m 'Create new footer [issue 53]'
```



Ветка iss53 двигается вперед

Тут вы получаете сообщение об обнаружении уязвимости на вашем сайте, которую нужно немедленно устранить. Благодаря Git, не требуется размещать это исправление вместе с тем, что вы сделали в iss53. Вам даже не придется прилагать усилий, чтобы откатить все эти изменения для начала работы над исправлением. Все, что вам нужно — переключиться на ветку master.

Но перед тем, как сделать это — имейте в виду, что если рабочий каталог либо индекс содержат незафиксированные изменения, конфликтующие с веткой, на которую вы хотите переключиться, то Git не позволит переключить ветки. Лучше всего переключаться из чистого рабочего состояния проекта. Есть способы обойти это (припрятать изменения (stash) или добавить их в последний коммит (amend)). Теперь предположим, что вы зафиксировали все свои изменения и можете переключиться на ветку master:

```
$ git checkout master

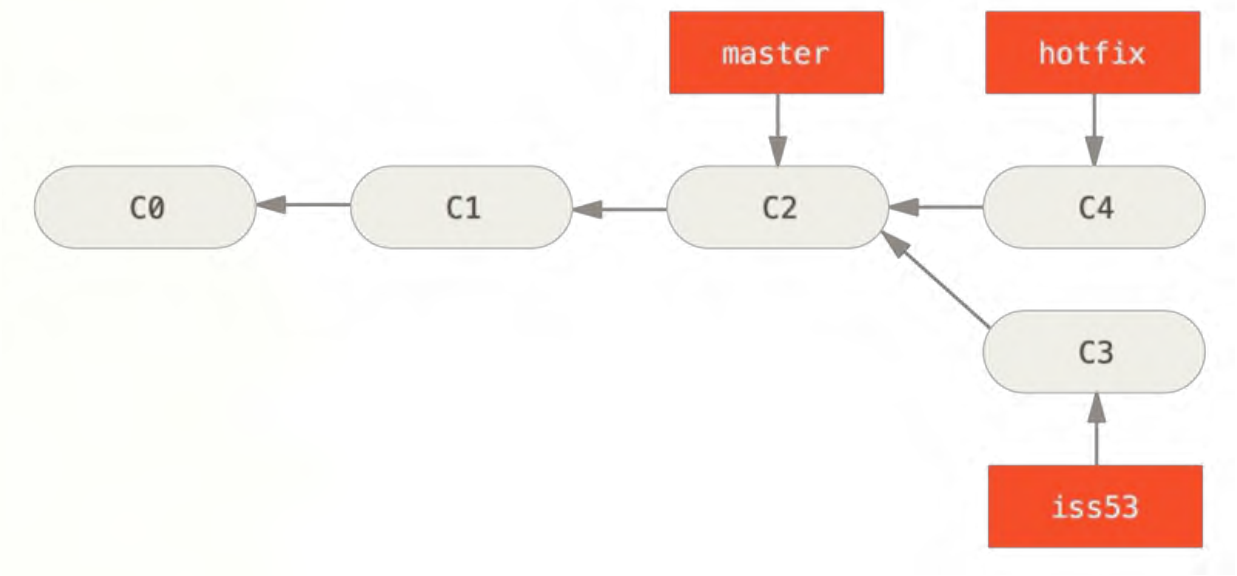
Switched to branch 'master'
```

С этого момента ваш рабочий каталог имеет точно такой же вид, какой был перед началом работы над проблемой #53, и вы можете сосредоточиться на работе над исправлением. Важно запомнить: когда вы переключаете ветки, Git возвращает состояние рабочего каталога к тому

виду, какой он имел в момент последнего коммита в эту ветку. Он добавляет, удаляет и изменяет файлы автоматически, чтобы состояние рабочего каталога соответствовало тому, когда был сделан последний коммит.

Теперь вы можете перейти к написанию исправления. Давайте создадим новую ветку для исправления, в которой будем работать, пока не закончим исправление.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'Fix broken email address'
[hotfix 1fb7853] Fix broken email address
 1 file changed, 2 insertions(+)
```



Ветка hotfix основана на ветке master

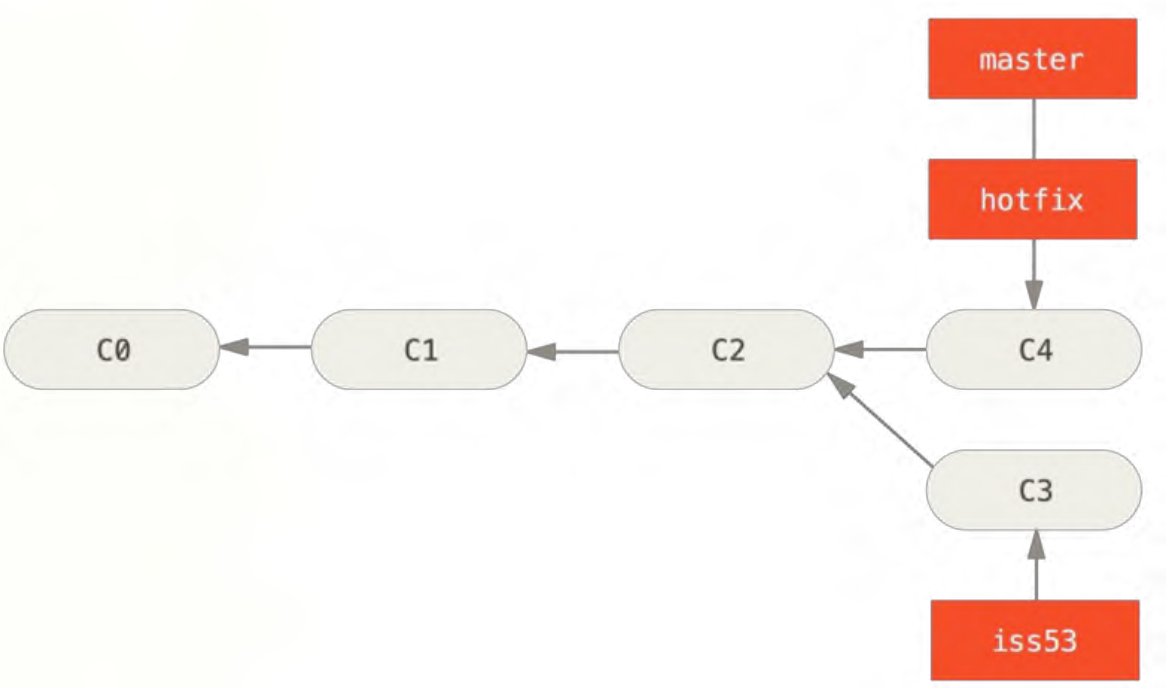
Вы можете прогнать тесты, чтобы убедиться, что ваше исправление делает именно то, что нужно. И если это так — выполнить слияние ветки `hotfix` с веткой `master` для включения изменений в продукт. Это делается командой `git merge`:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
```

```
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Заметили фразу «fast-forward» в этом слиянии? Git просто переместил указатель ветки вперед, потому что коммит C4, на который указывает слитая ветка hotfix, был прямым потомком коммита C2, на котором вы находились до этого. Другими словами, если коммит сливается с тем, до которого можно добраться, двигаясь по истории прямо, Git упрощает слияние просто перенося указатель ветки вперед, так как нет расхождений в изменениях. Это называется «fast-forward».

Теперь ваши изменения включены в коммит, на который указывает ветка master, и исправление можно внедрять.



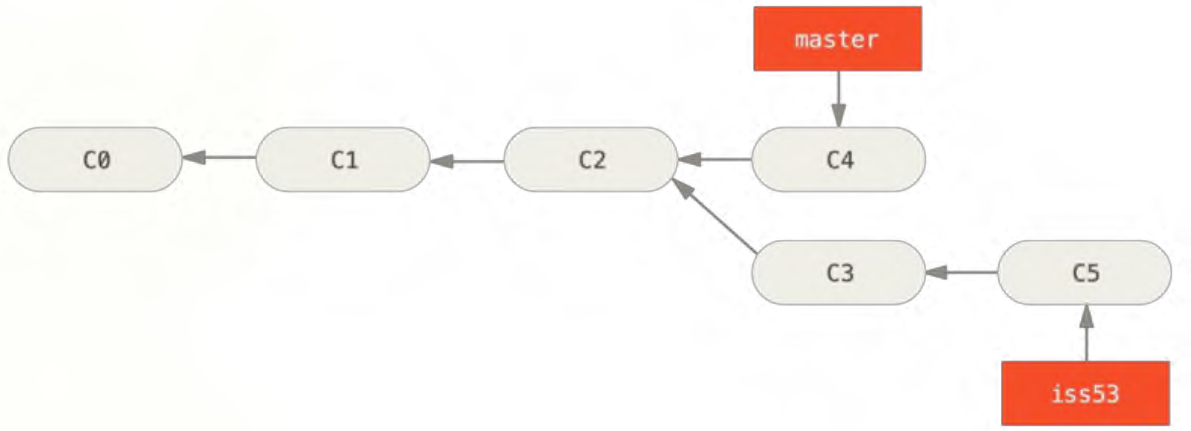
. `master` перемотан до `hotfix`

После внедрения вашего архиважного исправления вы готовы вернуться к работе над тем, что были вынуждены отложить. Но сначала нужно удалить ветку `hotfix`, потому что она больше не нужна — ветка `master` указывает на то же самое место. Для удаления ветки выполните команду `git branch` с параметром `-d`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Теперь вы можете переключиться обратно на ветку iss53 и продолжить работу над проблемой #53:

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'Finish the new footer [issue 53]'
[iss53 ad82d7a] Finish the new footer [issue 53]
1 file changed, 1 insertion(+)
```



Продолжение работы над iss53

Стоит обратить внимание на то, что все изменения из ветки hotfix не включены в вашу ветку iss53. Если их нужно включить, вы можете влить ветку master в вашу ветку iss53 командой git merge master, или же вы можете отложить слияние этих изменений до завершения работы, и затем влить ветку iss53 в master.

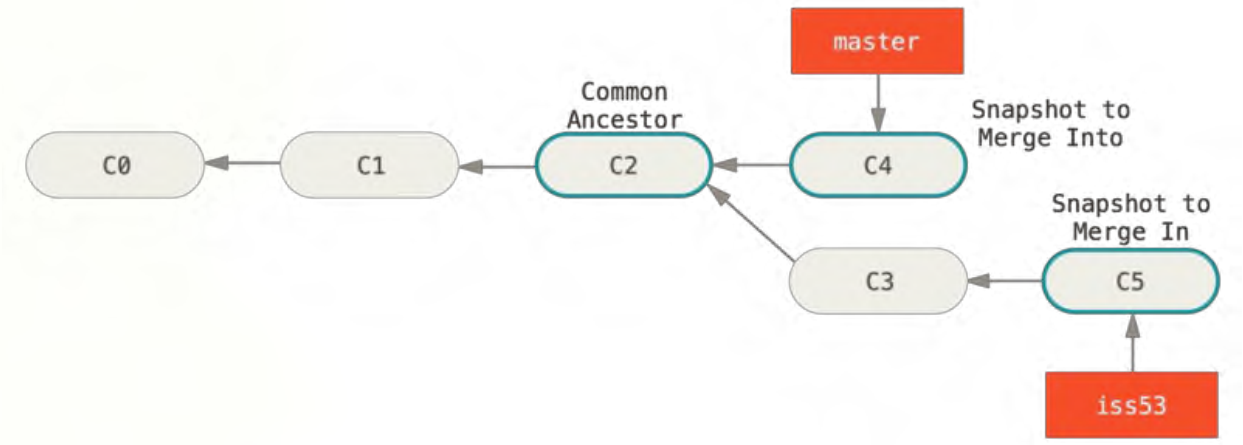
Основы слияния

Предположим, вы решили, что работа по проблеме #53 закончена и её можно влить в ветку master. Для этого нужно выполнить слияние ветки iss53 точно так же, как вы делали это с веткой hotfix ранее. Все, что нужно сделать — переключиться на ветку, в которую вы хотите включить изменения, и выполнить команду git merge:

```
$ git checkout master
Switched to branch 'master'
```

```
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Результат этой операции отличается от результата слияния ветки hotfix. В данном случае процесс разработки ответвился в более ранней точке. Так как коммит, на котором мы находимся, не является прямым родителем ветки, с которой мы выполняем слияние, Git придётся немного потрудиться. В этом случае Git выполняет простое трёхстороннее слияние, используя последние коммиты объединяемых веток и общего для них родительского коммита.



Использование трёх снимков при слиянии

Вместо того, чтобы просто передвинуть указатель ветки вперёд, Git создаёт новый результирующий снимок трёхстороннего слияния, а затем автоматически делает коммит. Этот особый коммит называют коммитом слияния, так как у него более одного предка.

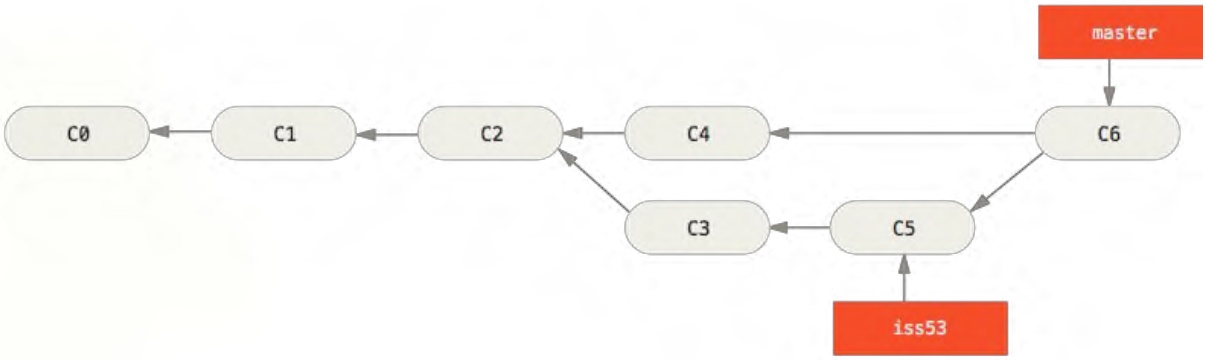


Рисунок 25. Коммит слияния

Теперь, когда изменения слиты, ветка iss53 больше не нужна. Вы можете закрыть задачу в системе отслеживания ошибок и удалить ветку:

```
$ git branch -d iss53
```

Основные конфликты слияния

Иногда процесс не проходит гладко. Если вы изменили одну и ту же часть одного и того же файла по-разному в двух объединяемых ветках, Git не сможет их чисто объединить. Если ваше исправление ошибки #53 потребовало изменить ту же часть файла что и hotfix, вы получите примерно такое сообщение о конфликте слияния:

```
$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.
```

Git не создал коммит слияния автоматически. Он остановил процесс до тех пор, пока вы не разрешите конфликт. Чтобы в любой момент после появления конфликта увидеть, какие файлы не объединены, вы можете запустить git status:

```
$ git status

On branch master

You have unmerged paths.

  (fix conflicts and run "git commit")


Unmerged paths:
```

```
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Всё, где есть неразрешённые конфликты слияния, перечисляется как неслитое. В конфликтующие файлы Git добавляет специальные маркеры конфликтов, чтобы вы могли исправить их вручную. В вашем файле появился раздел, выглядящий примерно так:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Это означает, что версия из HEAD (вашей ветки master, поскольку именно её вы извлекли перед запуском команды слияния) — это верхняя часть блока (всё, что над =======), а версия из вашей ветки iss53 представлена в нижней части. Чтобы разрешить конфликт, придётся выбрать один из вариантов, либо объединить содержимое по-своему. Например, вы можете разрешить конфликт, заменив весь блок следующим:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

В этом разрешении есть немного от каждой части, а строки <<<<<<<, ======= и >>>>>>> полностью удалены. Разрешив каждый конфликт во всех файлах, запустите git add для каждого файла, чтобы отметить конфликт как решённый. Добавление файла в индекс означает для Git, что все конфликты в нём исправлены.

Если вы хотите использовать графический инструмент для разрешения конфликтов, можно запустить git mergetool, который проведет вас по всем конфликтам:

```
$ git mergetool


This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html


Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Если вы хотите использовать инструмент слияния не по умолчанию (в данном случае Git выбрал opendiff, поскольку команда запускалась на Mac), список всех поддерживаемых инструментов представлен вверху после фразы «one of the following tools». Просто введите название инструмента, который хотите использовать.

После выхода из инструмента слияния Git спросит об успешности процесса. Если вы ответите скрипту утвердительно, то он добавит файл в индекс, чтобы отметить его как разрешенный. Теперь можно снова запустить git status, чтобы убедиться в отсутствии конфликтов:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)
```

```
Changes to be committed:

   modified:   index.html
```

Если это вас устраивает, и вы убедились, что все файлы, где были конфликты, добавлены в индекс — выполните команду git commit для создания коммита слияния. Комментарий к коммиту слияния по умолчанию выглядит примерно так:

```
Merge branch 'iss53'

Conflicts:
   index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#           .git/MERGE_HEAD
# and try again.



# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#           modified:   index.html
#
```

Если вы считаете, что коммит слияния требует дополнительных пояснений — опишите как были разрешены конфликты и почему были применены именно такие изменения, если это не очевидно.

Управление ветками

Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.

Команда `git branch` делает несколько больше, чем просто создаёт и удаляет ветки. При запуске без параметров вы получите простой список имеющихся у вас веток:

```
$ git branch
  iss53
* master
  testing
```

Обратите внимание на символ `*`, стоящий перед веткой `master`: он указывает на ветку, на которой вы находитесь в настоящий момент (т. е. ветку, на которую указывает `HEAD`). Это означает, что если вы сейчас сделаете коммит, ветка `master` переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду `git branch -v`:

```
$ git branch -v
  iss53   93b412c Fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 Add scott to the author list in the readme
```

Опции `--merged` и `--no-merged` могут отфильтровать этот список для вывода только тех веток, которые слиты или ещё не слиты в текущую ветку. Чтобы посмотреть те ветки, которые вы уже слили с текущей, можете выполнить команду `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Ветка iss53 присутствует в этом списке, потому что вы ранее слили её в master. Те ветки из этого списка, перед которыми нет символа *, можно смело удалять командой git branch -d; наработки из этих веток уже включены в другую ветку, так что ничего не потеряется.

Чтобы увидеть все ветки, содержащие наработки, которые вы пока ещё не слили в текущую ветку, выполните команду git branch --no-merged:

```
$ git branch --no-merged
  testing
```

Вы увидите оставшуюся ветку. Так как она содержит ещё не слитые наработки, попытка удалить её командой git branch -d приведёт к ошибке:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Если вы действительно хотите удалить ветку вместе со всеми наработками, используйте опцию -D, как указано в подсказке.

Подсказка	Если в качестве аргумента не указан коммит или ветка, то опции --merged и --no-merged покажут, что уже слито или не слито с вашей **текущей** веткой соответственно.
	Вы всегда можете указать дополнительный аргумент для вывода той же информации, но относительно указанной ветки, предварительно не извлекая и не переходя на неё.
	`$ git checkout testing`
	`$ git branch --no-merged master`
	`  topicA`
	`  featureB`

Переименование ветки

Внимание	Не переименовывайте ветки, которые всё ещё используются другими участниками. Не переименовывайте ветку в master/main/mainline, не прочитав раздел «Изменение имени главной ветки».

Предположим, у вас есть ветка с именем bad-branch-name, и вы хотите изменить её на corrected-branch-name, сохранив при этом всю историю. Вместе с этим вы также хотите изменить имя ветки на удалённом сервере (GitHub, GitLab или другой сервер). Как это сделать?

Переименуйте ветку локально с помощью команды git branch --move:

```
$ git branch --move bad-branch-name corrected-branch-name
```

Ветка bad-branch-name будет переименована в corrected-branch-name, но это изменение пока только локальное. Чтобы все остальные увидели исправленную ветку в удалённом репозитории, отправьте её туда:

```
$ git push --set-upstream origin corrected-branch-name
```

Теперь проверим, где мы сейчас находимся:

```
$ git branch --all
* corrected-branch-name
  main
  remotes/origin/bad-branch-name
  remotes/origin/corrected-branch-name
  remotes/origin/main
```

Обратите внимание, что текущая ветка corrected-branch-name, которая также присутствует и на удалённом сервере. Однако, старая ветка всё ещё по-прежнему там, но её можно удалить с помощью команды:

```
$ git push origin --delete bad-branch-name
```

Теперь старое имя ветки полностью заменено исправленным.

Изменение имени главной ветки

Предупреждение	Изменение имени ветки, например master/main/mainline/default, сломает интеграции, службы, вспомогательные утилиты и скрипты сборки, которые использует ваш репозиторий. Прежде чем сделать это, обязательно проконсультируйтесь с коллегами. Также убедитесь, что вы выполнили тщательный поиск в своём репозитории и обновили все ссылки на старое имя ветки в вашем коде или скриптах.

Переименуйте локальную ветку `master` в `main` с помощью следующей команды:

```
$ git branch --move master main
```

После этого, локальной ветки `master` больше не существует, потому что она была переименована в ветку `main`.

Чтобы все остальные могли видеть новую ветку `main`, вам нужно отправить её в общий репозиторий. Это делает переименованную ветку доступной в удалённом репозитории.

```
$ git push --set-upstream origin main
```

В итоге, состояние репозитория становится следующим:

```
git branch --all

* main

  remotes/origin/HEAD -> origin/master

  remotes/origin/main

  remotes/origin/master
```

Ваша локальная ветка `master` исчезла, так как она заменена веткой `main`. Ветка `main` доступна в удалённом репозитории. Старая ветка `master` всё ещё присутствует в удалённом репозитории. Остальные участники будут продолжать использовать ветку `master` в качестве основы для своей работы, пока вы не совершите ряд дополнительных действий.

Теперь, для завершения перехода на новую ветку перед вами стоят следующие задачи:

- Все проекты, которые зависят от текущего, должны будут обновить свой код и/или конфигурацию.
- Обновите конфигурацию всех запускаемых тестов.
- Исправьте скрипты сборки и публикации артефактов.
- Поправьте настройки репозитория на сервере: задайте новую ветку по умолчанию, обновите правила слияния, а также прочие настройки, которые зависят от имени веток.
- Обновите документацию, исправив ссылки, указывающие на старую ветку.
- Слейте или отмените запросы на слияние изменений, нацеленные на старую ветку.

После того, как вы выполнили все эти задачи и уверены, что ветка main работает так же, как ветка master, вы можете удалить ветку master:

```
$ git push origin --delete master
```

Работа с ветками

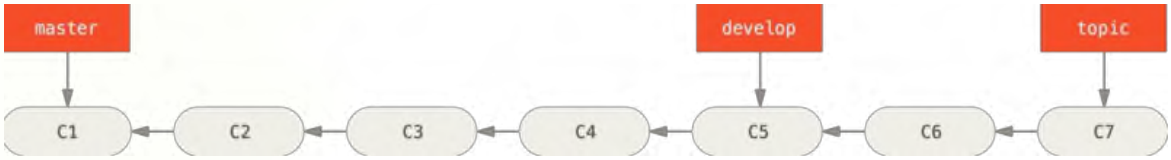
Теперь, когда вы познакомились с основами ветвления и слияния, возникает вопрос: что можно или нужно делать с этим? В этом разделе мы разберём некоторые основные рабочие процессы, ставшие возможными благодаря облегчённой процедуре ветвления, которые вы возможно захотите применить в собственном цикле разработки.

Долгоживущие ветки

Так как в Git применяется простое трёхстороннее слияние, ничто не мешает многократно объединять ветки в течение длительного времени. Это значит, что у вас, может быть, несколько постоянно открытых веток, которые вы используете для разных этапов вашего цикла разработки; вы можете регулярно сливать изменения из одной ветки в другую.

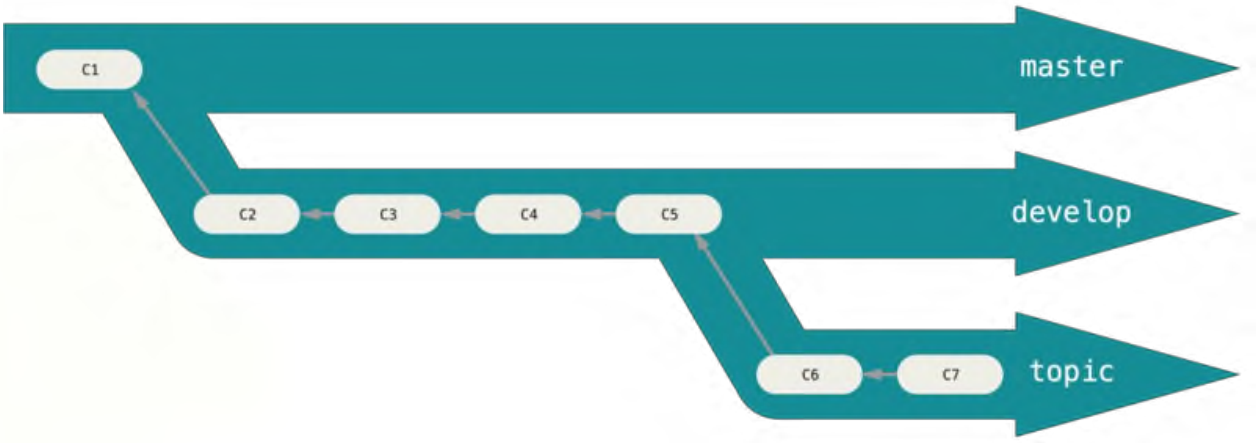
Многие разработчики, использующие Git, придерживаются именно такого подхода, оставляя полностью стабильный код только в ветке master — возможно, только тот код, который был или будет выпущен. При этом существует и параллельная ветка с именем develop или next, предназначенная для работы и тестирования стабильности; она не обязательно должна быть всегда стабильной, но при достижении стабильного состояния ее содержимое можно слить в ветку master. Она используется для слияния завершённых задач из тематических веток (временных веток наподобие iss53), чтобы гарантировать, что эти задачи проходят тестирование и не вносят ошибок.

По сути, мы говорим про указатели, перемещающиеся по линии создаваемых вами коммитов. Стабильные ветки находятся в нижнем конце истории коммитов, а самые свежие наработки — ближе к её верхней части



Линейное представление повышения стабильности веток

В общем случае это можно представить в виде накопителей, в которых наборы коммитов перемещаются на более стабильный уровень только после полного тестирования.



Представление диаграммы стабильности веток в виде многоуровневого накопителя

Число уровней стабильности можно увеличить. В крупных проектах зачастую появляется ветка proposed или pu (предлагаемые обновления), объединяющая ветки с содержимым, которое ещё не готово к включению в ветки next или master. Идея состоит в том, что каждая ветка представляет собой определённый уровень стабильности; как только он повышается, содержимое сливается в ветку уровнем выше. Разумеется, можно вообще обойтись без долгоживущих веток, но зачастую они имеют смысл, особенно при работе над большими и сложными проектами.

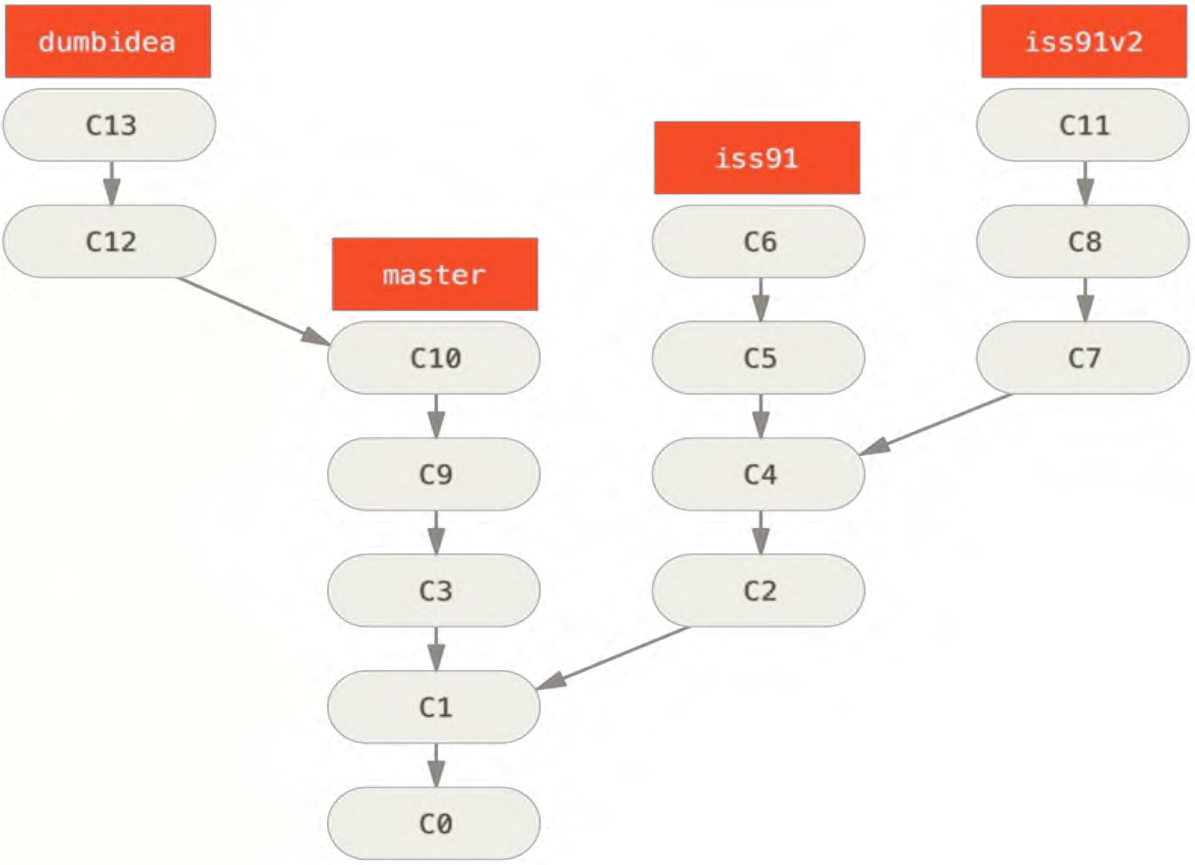
Тематические ветки

А вот такая вещь, как тематические ветки, полезна вне зависимости от величины проекта. Тематической веткой называется временная ветка, создаваемая и используемая для работы над конкретной функциональной возможностью или решения сопутствующих задач. Скорее всего, при работе с другими СКВ вы никогда ничего подобного не делали, так как там создание и слияние веток — затратные операции. Но в Git это обычное дело — много раз в день создавать ветки, работать с ними, сливать их и удалять.

Пример тематических веток вы видели в предыдущем разделе, когда мы создавали ветки iss53 и hotfix. В каждой из них было создано несколько коммитов, после чего, сразу же после слияния с основной веткой, они были удалены. Такая техника позволяет быстро и полностью осуществлять переключения контекста — так как работа разделена по уровням и все изменения в конкретной ветке относятся к определённой теме, что позволяет легко увидеть, что именно было сделано во время процедуры просмотра кода или аналогичной. Ветки с внесёнными в них изменениями можно хранить минуты, дни или даже месяцы, а слияние

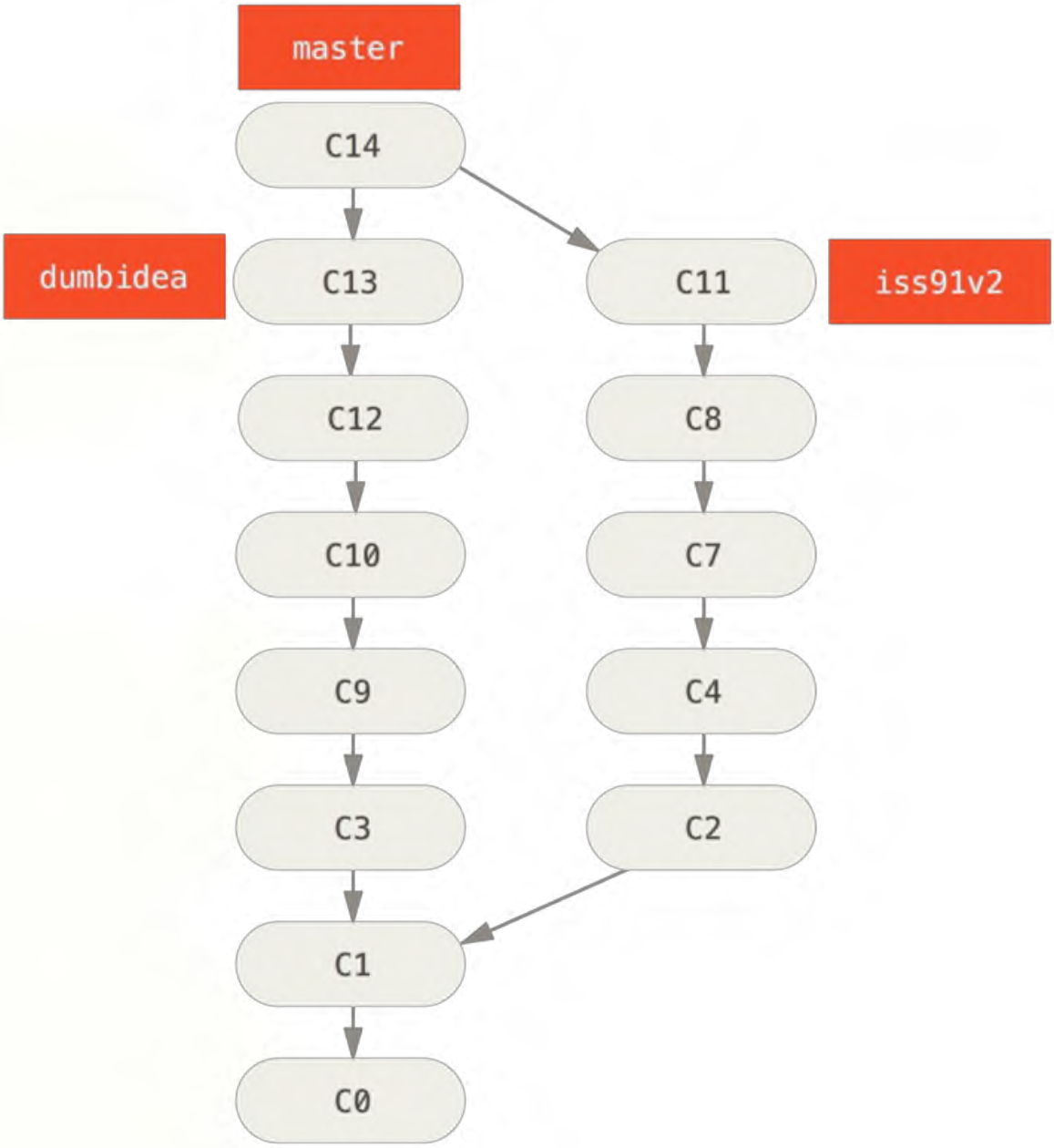
выполнить только когда это действительно потребуется, вне зависимости от порядка их создания.

Предположим, мы работаем в ветке master, ответвляемся для решения попутной проблемы (iss91), некоторое время занимаемся ею, затем создаём ветку, чтобы попробовать решить эту задачу другим способом (iss91v2), возвращаемся в ветку master и выполняем там некие действия, затем создаём новую ветку для изменений, в результате которых не уверены (ветка dumbidea). Результирующая история коммитов будет выглядеть примерно так:



Набор тематических веток

Предположим, вам больше нравится второй вариант решения задачи (iss91v2), а ветку dumbidea вы показали коллегам, и оказалось, что там содержится гениальная идея. Фактически вы можете удалить ветку iss91 (потеряв коммиты C5 и C6) и слить две другие ветки. После этого история будет выглядеть так:



История после слияния веток dumbidea и iss91v2

Важно помнить, что во время всех этих манипуляций ветки полностью локальны. Ветвления и слияния выполняются только в вашем Git репозитории — связь с сервером не требуется.

Удалённые ветки

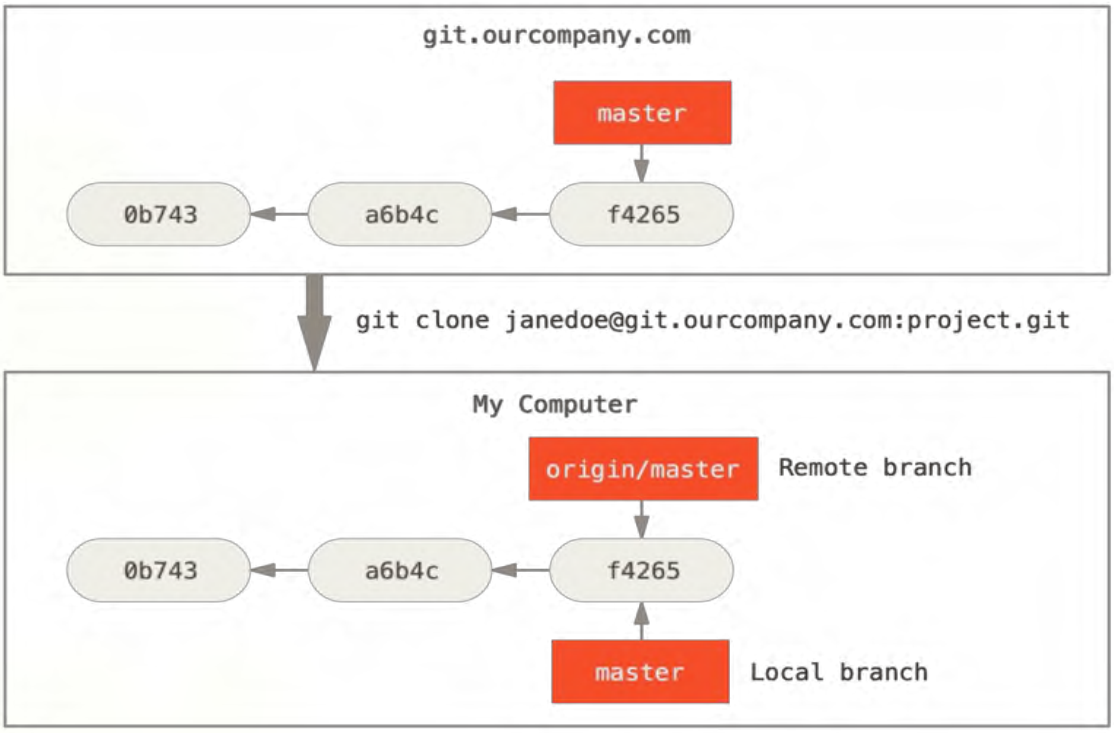
Удалённые ссылки — это ссылки (указатели) в ваших удалённых репозиториях, включая ветки, теги и так далее. Полный список удалённых ссылок можно получить с помощью команды `git ls-remote <remote>` или команды `git remote show <remote>` для получения удалённых веток и дополнительной информации. Тем не менее, более распространенным способом является использование веток слежения.

Ветки слежения — это ссылки на определённое состояние удалённых веток. Это локальные ветки, которые нельзя перемещать; Git перемещает их автоматически при любой коммуникации с удаленным репозиторием, чтобы гарантировать точное соответствие с ним. Представляйте их как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

Имена веток слежения имеют вид <remote>/<branch>. Например, если вы хотите посмотреть, как выглядела ветка master на сервере origin во время последнего соединения с ним, используйте ветку origin/master. Если вы с коллегой работали над одной задачей, и он отправил на сервер ветку iss53, при том что у вас может быть своя локальная ветка iss53, удалённая ветка будет представлена веткой слежения с именем origin/iss53.

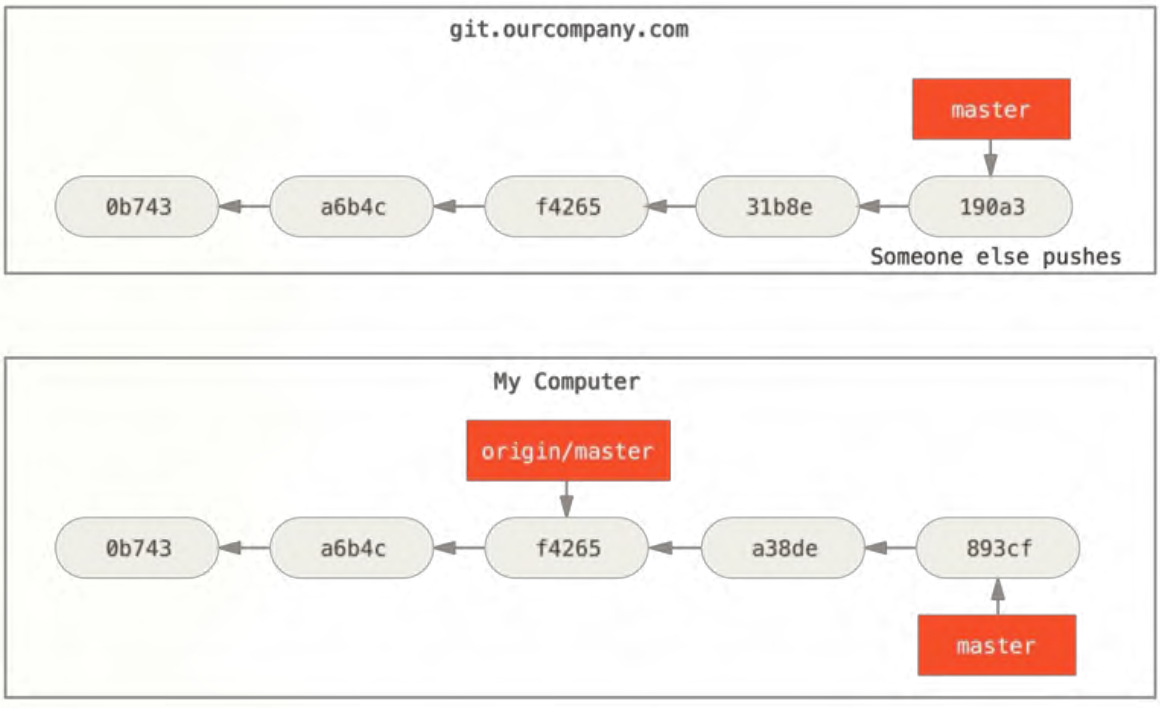
Возможно, всё это сбивает с толку, поэтому давайте рассмотрим на примере. Скажем, у вас в сети есть свой Git-сервер с адресом git.ourcompany.com. Если вы с него что-то клонируете, команда clone автоматически назовёт его origin, заберёт оттуда все данные, создаст указатель на то, на что там указывает ветка master, и назовёт его локально origin/master. Git также создаст вам локальную ветку master, которая будет начинаться там же, где и ветка master в origin, так что вам будет с чего начать.

Примечание	**«origin» — это не специальное название** Подобно названию ветки «master», «origin» не имеет какого-либо специального значения в Git. В то время как «master» — это название по умолчанию для ветки при выполнении git init только потому, что часто используется, «origin» — это название по умолчанию для удалённого сервера, когда вы запускаете git clone. Если вы выполните git clone -o booyah, то по умолчанию ветка слежения будет иметь вид booyah/master.



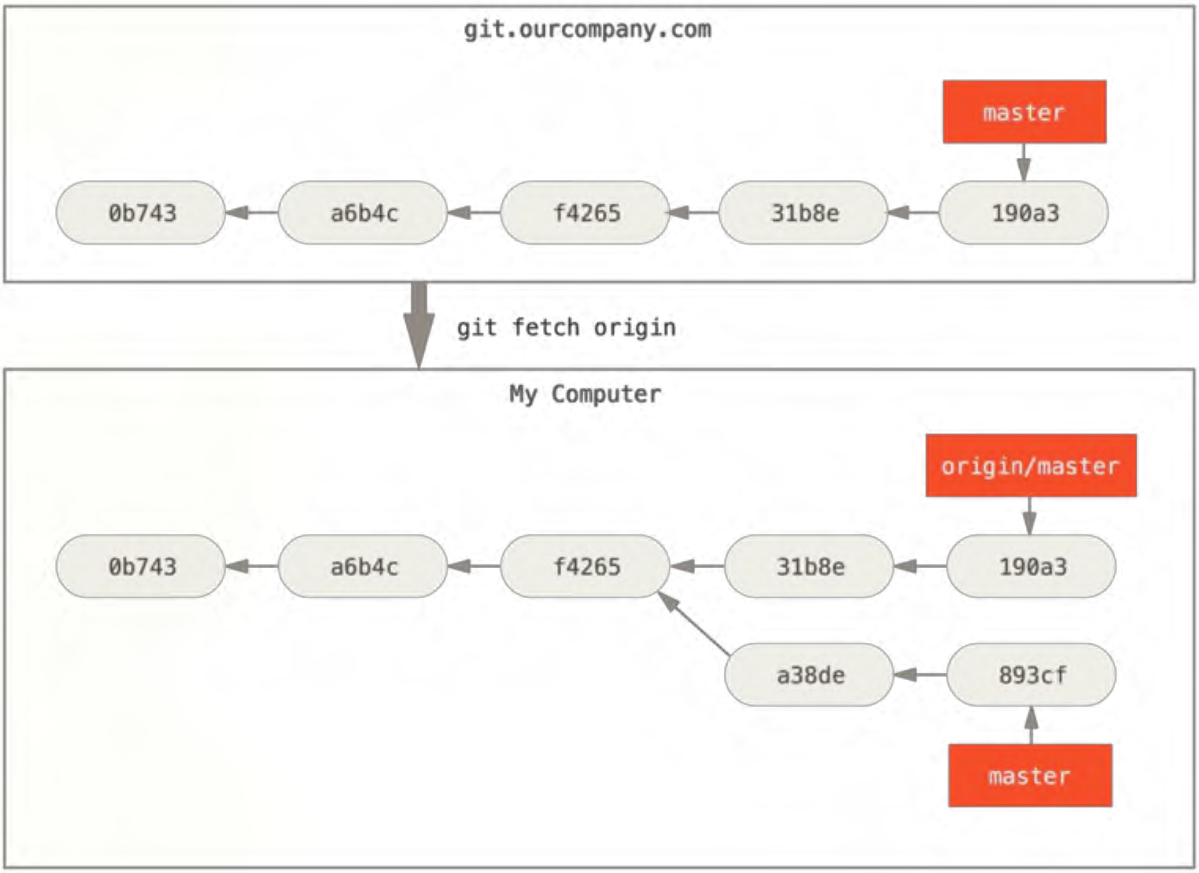
Серверный и локальный репозитории после клонирования

Если вы сделаете что-то в своей локальной ветке master, а тем временем кто-то отправит изменения на сервер git.ourcompany.com и обновит там ветку master, то ваши истории продолжатся по-разному. Пока вы не свяжетесь с сервером origin ваш указатель origin/master останется на месте.



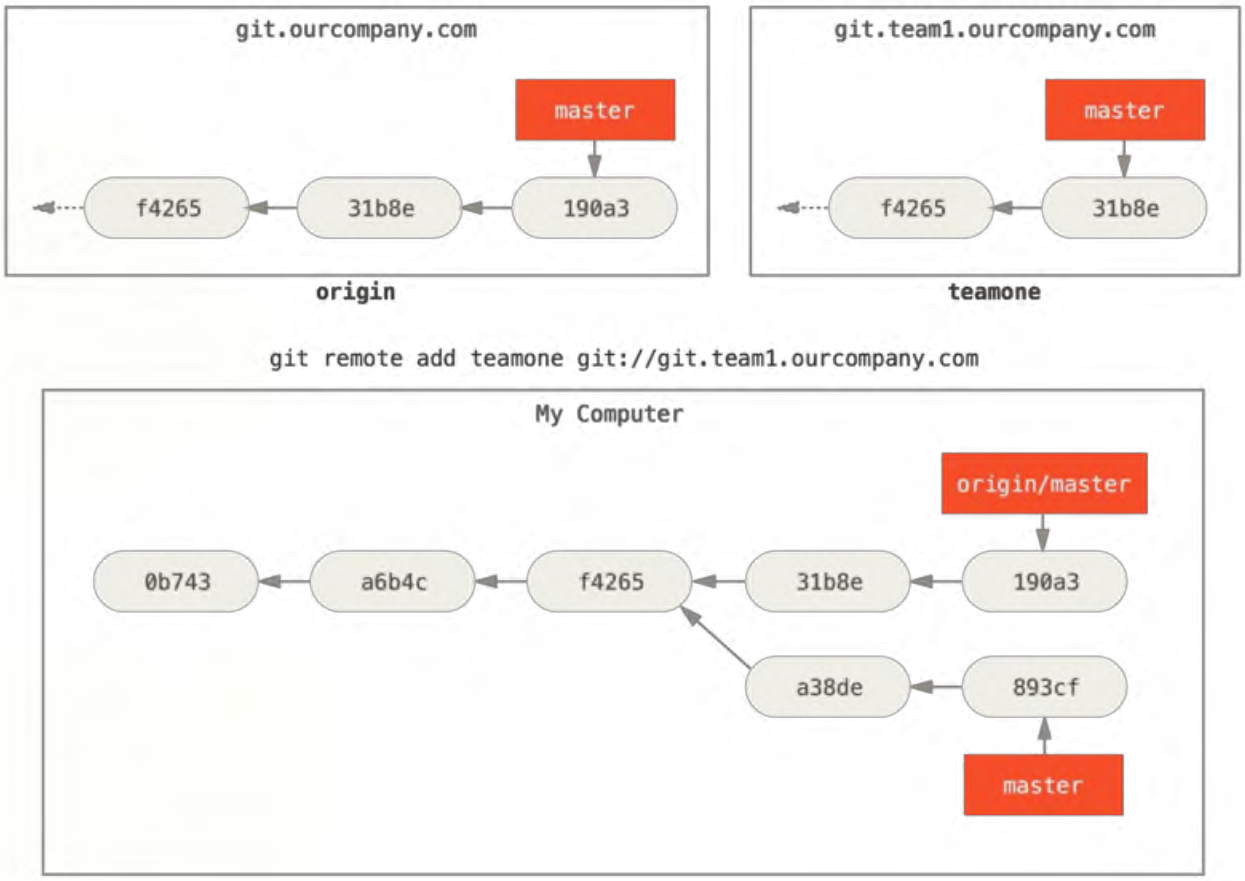
Локальная и удалённая работа может расходиться

Для синхронизации ваших изменений с удаленным сервером выполните команду `git fetch <remote>` (в нашем случае `git fetch origin`). Эта команда определяет какому серверу соответствует «origin» (в нашем случае это `git.ourcompany.com`), извлекает оттуда данные, которых у вас ещё нет, и обновляет локальную базу данных, сдвигая указатель `origin/master` на новую позицию.



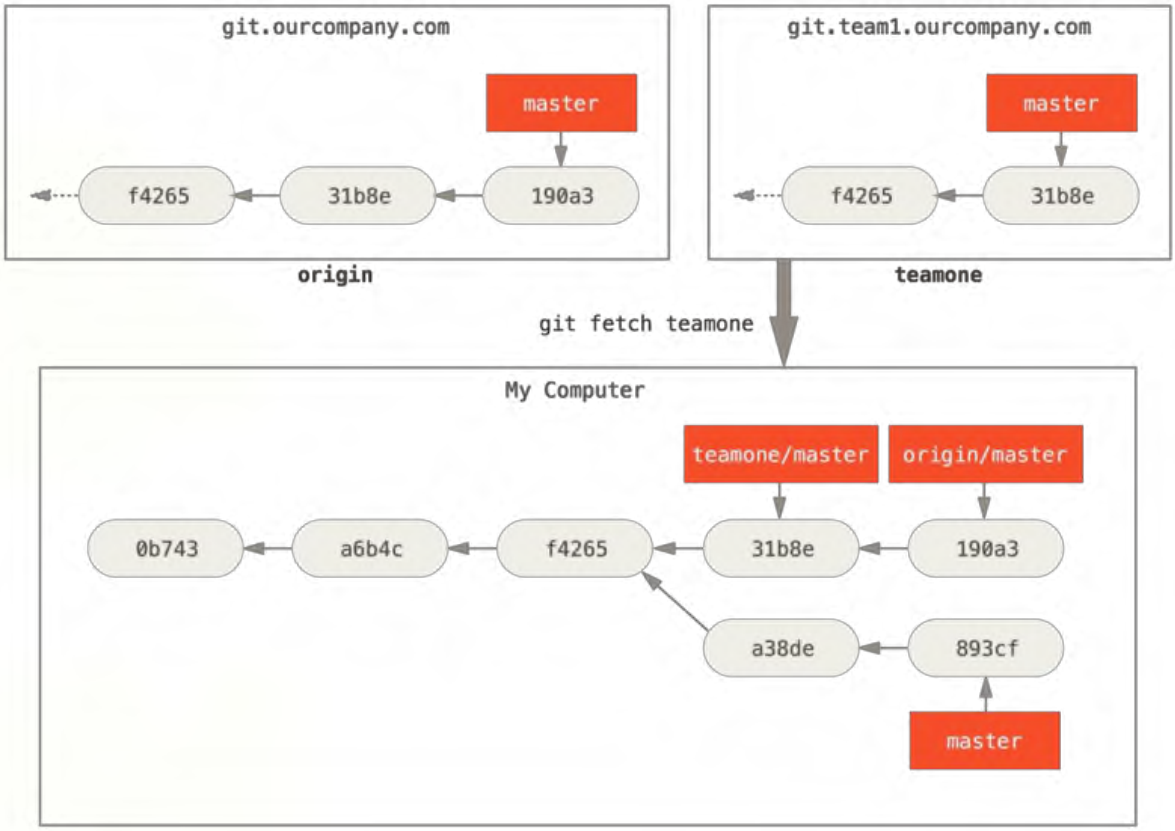
`git fetch` обновляет ветки слежения

Чтобы продемонстрировать, как будут выглядеть удалённые ветки в ситуации с несколькими удалёнными серверами, предположим, что у вас есть ещё один внутренний Git-сервер, который используется для разработки только одной из ваших команд разработчиков. Этот сервер находится на `git.team1.ourcompany.com`. Вы можете добавить его в качестве новой удалённой ссылки для текущего проекта с помощью команды `git remote add`. Назовите этот удалённый сервер `teamone` — это имя будет сокращением вместо полного URL.



Добавление ещё одного сервера в качестве удалённой ветки

Теперь вы можете выполнить команду `git fetch teamone` для получения всех изменений с сервера `teamone`, которых у вас нет локально. Так как в данный момент на этом сервере есть только те данные, что содержит сервер `origin`, Git ничего не получит, но создаст ветку слежения с именем `teamone/master`, которая будет указывать на тот же коммит, что и ветка `master` на сервере `teamone`.



Ветка слежения teamone/master

Отправка изменений

Когда вы хотите поделиться веткой, вам необходимо отправить её на удалённый сервер, где у вас есть права на запись. Ваши локальные ветки автоматически не синхронизируются с удалёнными при отправке — вам нужно явно указать те ветки, которые вы хотите отправить. Таким образом, вы можете использовать свои личные ветки для работы, которую не хотите показывать, а отправлять только те тематические ветки, над которыми вы хотите работать с кем-то совместно.

Если у вас есть ветка serverfix, над которой вы хотите работать с кем-то ещё, вы можете отправить её точно так же, как вы отправляли вашу первую ветку. Выполните команду git push <remote> <branch>:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
```

```
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Это в некотором роде сокращение. Git автоматически разворачивает имя ветки serverfix до refs/heads/serverfix:refs/heads/serverfix, что означает «возьми мою локальную ветку serverfix и обнови ей удалённую ветку serverfix». Вы также можете выполнить git push origin serverfix:serverfix — произойдёт то же самое — здесь говорится «возьми мою ветку serverfix и сделай её удалённой веткой serverfix». Можно использовать этот формат для отправки локальной ветки в удалённую ветку с другим именем. Если вы не хотите, чтобы на удалённом сервере ветка называлась serverfix, то вместо предыдущей команды выполните git push origin serverfix:awesomebranch, которая отправит локальную ветку serverfix в ветку awesomebranch удалённого репозитория.

Примечание	**Не вводите каждый раз свой пароль** Если вы используете HTTPS URL для отправки изменений, Git-сервер будет спрашивать имя пользователя и пароль для аутентификации. По умолчанию вам будет предложено ввести эти данные в терминале, чтобы сервер мог определить разрешена ли вам отправка изменений. Если вы не хотите вводить свои данные каждый раз при отправке изменений, вы можете настроить «credential cache». Проще всего держать их в памяти несколько минут, это легко настроить с помощью команды git config --global credential.helper cache.

В следующий раз, когда один из ваших соавторов будет получать обновления с сервера, он получит ссылку на то, на что указывает serverfix на сервере, как удалённую ветку origin/serverfix:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
```

```
 * [new branch]      serverfix    -> origin/serverfix
```

Необходимо отметить, что при получении данных создаются ветки слежения, вы не получаете автоматически для них локальных редактируемых копий. Другими словами, в нашем случае вы не получите новую ветку serverfix — только указатель origin/serverfix, который вы не можете изменять.

Чтобы слить эти наработки в свою текущую рабочую ветку, выполните git merge origin/serverfix. Если вам нужна локальная ветка serverfix, в которой вы сможете работать, то вы можете создать её на основе ветки слежения:

```
$ git checkout -b serverfix origin/serverfix
```

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

Это даст вам локальную ветку, в которой можно работать и которая будет начинаться там же, где и origin/serverfix.

Отслеживание веток

Получение локальной ветки из удалённой ветки автоматически создаёт то, что называется «веткой слежения» (а ветка, за которой следит, локальная называется «upstream branch»). Ветки слежения — это локальные ветки, которые напрямую связаны с удалённой веткой. Если, находясь на ветке слежения, выполнить git pull, то Git уже будет знать с какого сервера получать данные и какую ветку использовать для слияния.

При клонировании репозитория, как правило, автоматически создаётся ветка master, которая следит за origin/master. Однако, при желании вы можете настроить отслеживание и других веток — следить за ветками на других серверах или отключить слежение за веткой master. Вы только что видели простейший пример, что сделать это можно с помощью команды git checkout -b <branch> <remote>/<branch>. Это часто используемая команда, поэтому Git предоставляет сокращённую форму записи в виде флага --track:

```
$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'
```

В действительности это настолько распространённая команда, что существует сокращение для этого сокращения. Если вы пытаетесь извлечь

ветку, которая не существует, но существует только одна удалённая ветка с точно таким же именем, то Git автоматически создаст ветку слежения:

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Чтобы создать локальную ветку с именем, отличным от имени удалённой ветки, просто укажите другое имя:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

Теперь ваша локальная ветка sf будет автоматически получать изменения из origin/serverfix.

Если у вас уже есть локальная ветка и вы хотите настроить ее на слежение за удалённой веткой, которую вы только что получили, или хотите изменить используемую upstream-ветку, то воспользуйтесь параметрами -u или --set-upstream-to для команды git branch, чтобы явно установить новое значение.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

Примечание	**Сокращение Upstream** Если у вас настроена отслеживаемая ветка, вы можете ссылаться на нее с помощью сокращений @{upstream} или @{u}. Итак, если вы находитесь на ветке master и она следит за origin/master, при желании вы можете использовать git merge @{u} вместо git merge origin/master.

Если вы хотите посмотреть, как у вас настроены ветки слежения, воспользуйтесь опцией -vv для команды git branch. Это выведет список локальных веток и дополнительную информацию о том, какая из веток отслеживается, отстаёт, опережает или всё сразу относительно отслеживаемой.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] Add forgotten brackets
```

```
  master    1ae2a45 [origin/master] Deploy index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] This should do it
  testing   5ea463a Try something new
```

Итак, здесь мы видим, что наша ветка iss53 следит за origin/iss53 и «опережает» её на два изменения — это значит, что у нас есть два локальных коммита, которые не отправлены на сервер. Мы также видим, что наша ветка master отслеживает ветку origin/master и находится в актуальном состоянии. Далее мы можем видеть, что локальная ветка serverfix следит за веткой server-fix-good на сервере teamone, опережает её на три коммита и отстает на один — это значит, что на сервере есть один коммит, который мы ещё не слили, и три локальных коммита, которые ещё не отправлены на сервер. В конце мы видим, что наша ветка testing не отслеживает удаленную ветку.

Важно отметить, что эти цифры описывают состояние на момент последнего получения данных с каждого из серверов. Эта команда не обращается к серверам, а лишь говорит вам о том, какая информация с этих серверов сохранена в локальном кэше. Если вы хотите иметь актуальную информацию об этих числах, вам необходимо получить данные со всех ваших удалённых серверов перед запуском команды. Сделать это можно вот так:

```
$ git fetch --all; git branch -vv
```

Получение изменений

Команда git fetch получает с сервера все изменения, которых у вас ещё нет, но не будет изменять состояние вашей рабочей копии. Эта команда просто получает данные и позволяет вам самостоятельно сделать слияние. Тем не менее, существует команда git pull, которая в большинстве случаев является командой git fetch, за которой непосредственно следует команда git merge. Если у вас настроена ветка слежения как показано в предыдущем разделе, или она явно установлена, или она была создана автоматически командами clone или checkout, git pull определит сервер и ветку, за которыми следит ваша текущая ветка, получит данные с этого сервера и затем попытается слить удалённую ветку.

Обычно, лучше явно использовать команды fetch и merge, поскольку магия git pull может часто сбивать с толку.

Удаление веток на удалённом сервере

Скажем, вы и ваши соавторы закончили с нововведением и слили его в ветку master на удалённом сервере (или в какую-то другую ветку, где хранится стабильный код). Вы можете удалить ветку на удалённом сервере используя параметр --delete для команды git push. Для удаления ветки serverfix на сервере, выполните следующую команду:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

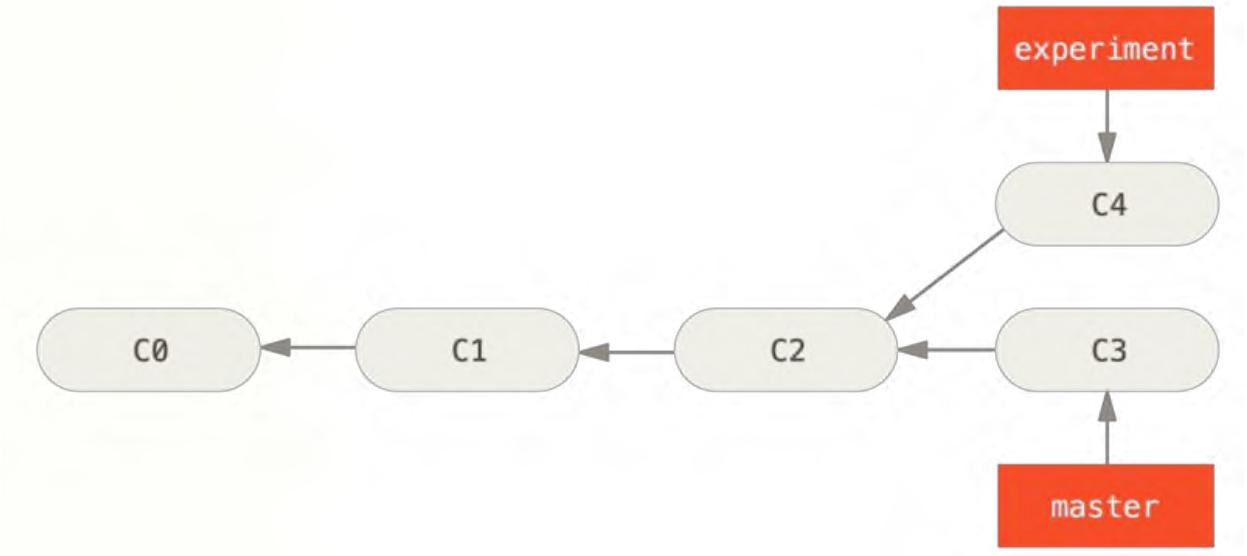
Всё, что делает эта строка — удаляет указатель на сервере. Как правило, Git сервер хранит данные пока не запустится сборщик мусора, поэтому если ветка была удалена случайно, чаще всего её легко восстановить.

Перебазирование

В Git есть два способа внести изменения из одной ветки в другую: слияние и перебазирование. В этом разделе вы узнаете, что такое перебазирование, как его осуществлять и в каких случаях этот удивительный инструмент использовать не следует.

Простейшее перебазирование

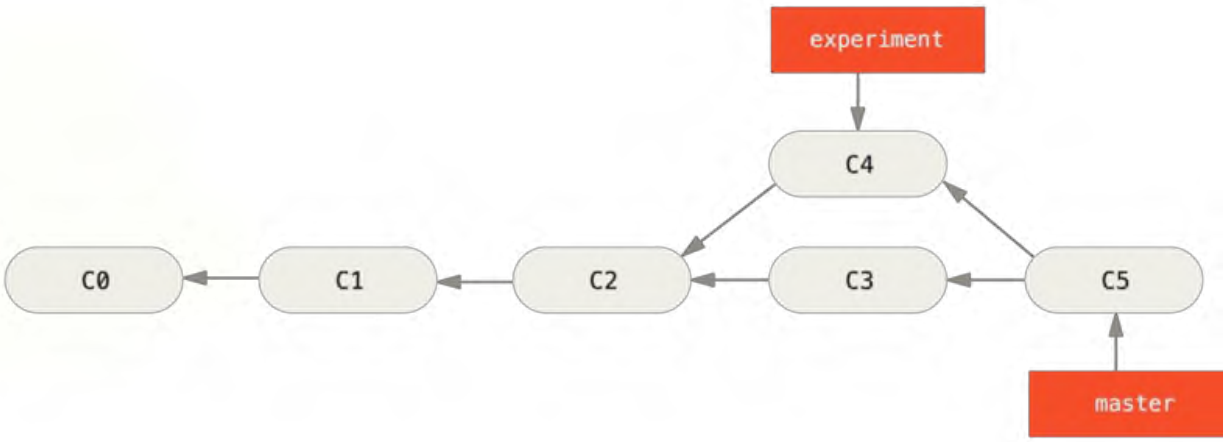
Если вы вернётесь к более раннему примеру из Основы слияния, вы увидите, что разделили свою работу и сделали коммиты в две разные ветки.



История коммитов простого разделения

Как мы выяснили ранее, простейший способ выполнить слияние двух веток — это команда merge. Она осуществляет трёхстороннее слияние между двумя последними снимками сливаемых веток (C3 и C4) и самого

недавнего общего для этих веток родительского снимка (C2), создавая новый снимок (и коммит).



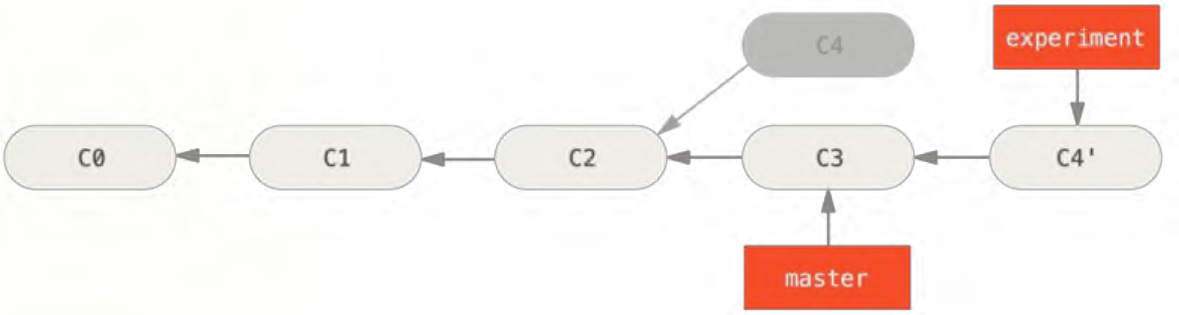
Слияние разделённой истории коммитов

Тем не менее есть и другой способ: вы можете взять те изменения, что были представлены в C4, и применить их поверх C3. В Git это называется **перебазированием**. С помощью команды rebase вы можете взять все коммиты из одной ветки и в том же порядке применить их к другой ветке.

В данном примере переключимся на ветку experiment и перебазируем её относительно ветки master следующим образом:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

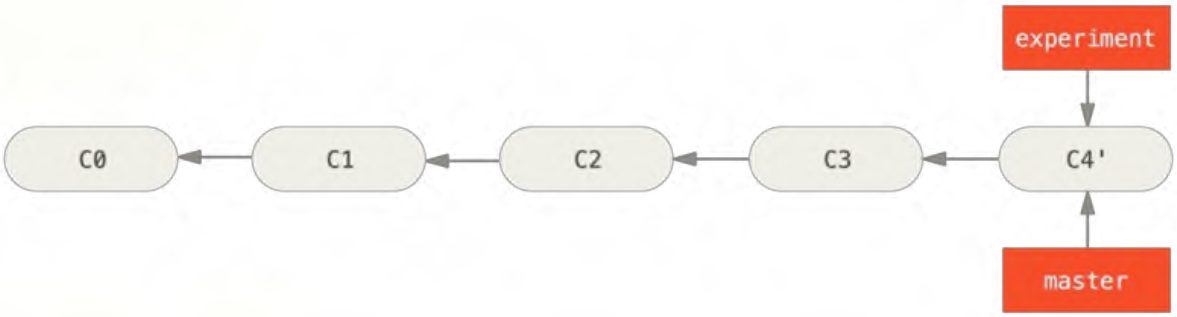
Это работает следующим образом: берётся общий родительский снимок двух веток (текущей, и той, поверх которой вы выполняете перебазирование), определяется дельта каждого коммита текущей ветки и сохраняется во временный файл, текущая ветка устанавливается на последний коммит ветки, поверх которой вы выполняете перебазирование, а затем по очереди применяются дельты из временных файлов.



Перебазирование изменений из C4 поверх C3

После этого вы можете переключиться обратно на ветку master и выполнить слияние перемоткой.

```
$ git checkout master
$ git merge experiment
```



Перемотка ветки master

Теперь снимок, на который указывает C4' абсолютно такой же, как тот, на который указывал C5 в примере с трёхсторонним слиянием. Нет абсолютно никакой разницы в конечном результате между двумя показанными примерами, но перебазирование делает историю коммитов чище. Если вы взглянете на историю перебазированной ветки, то увидите, что она выглядит абсолютно линейной: будто все операции были выполнены последовательно, даже если изначально они совершались параллельно.

Часто вы будете делать так для уверенности, что ваши коммиты могут быть бесконфликтно слиты в удалённую ветку — возможно, в проекте, куда вы пытаетесь внести вклад, но владельцем которого вы не являетесь. В этом случае вам следует работать в своей ветке и затем перебазировать вашу работу поверх origin/master, когда вы будете готовы отправить свои изменения в основной проект. Тогда владельцу проекта не придётся делать никакой лишней работы — всё решится простой перемоткой или бесконфликтным слиянием.

Учтите, что снимок, на который ссылается ваш последний коммит — является ли он последним коммитом после перебазирования или коммитом слияния после слияния — в обоих случаях это один и тот же снимок, отличаются только истории коммитов. Перебазирование повторяет изменения из одной ветки поверх другой в том порядке, в котором эти изменения были сделаны, в то время как слияние берет две конечные точки и сливает их вместе.

Опасности перемещения

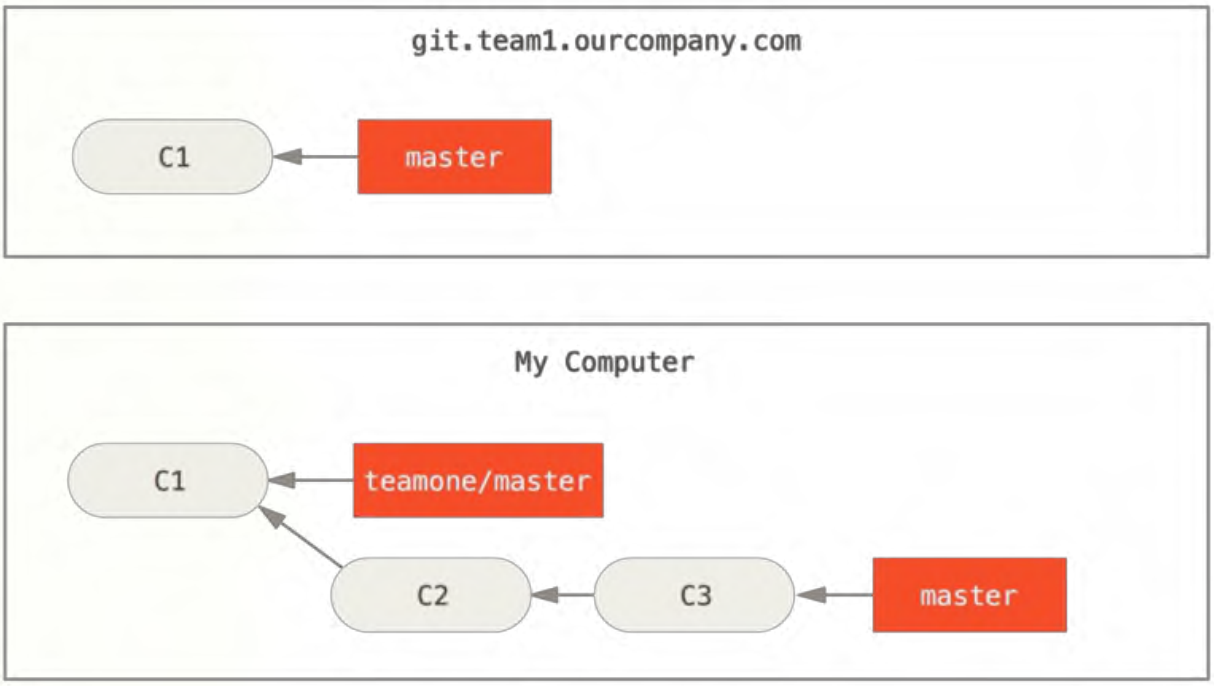
Но даже перебазирование, при всех своих достоинствах, не лишено недостатков, которые можно выразить одной строчкой:

Не перемещайте коммиты, уже отправленные в публичный репозиторий

Если вы будете придерживаться этого правила, всё будет хорошо. Если не будете, люди возненавидят вас, а ваши друзья и семья будут вас презирать.

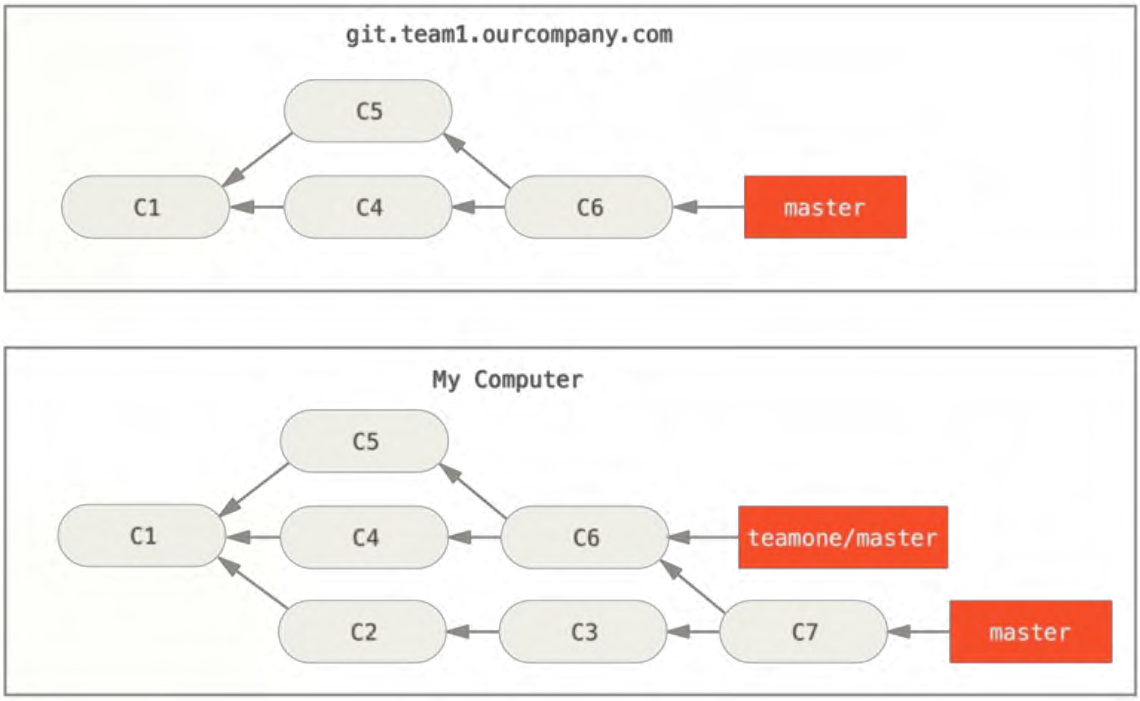
Когда вы что-то перемещаете, вы отменяете существующие коммиты и создаёте новые, **похожие** на старые, но являющиеся другими. Если вы куда-нибудь отправляете свои коммиты и другие люди забирают их себе и в дальнейшем основывают на них свою работу, а затем вы переделываете эти коммиты командой git rebase и выкладываете их снова, то ваши коллеги будут вынуждены заново выполнять слияние для своих наработок. В итоге, когда вы в очередной раз попытаетесь включить их работу в свою, вы получите путаницу.

Давайте рассмотрим пример того, как перемещение публично доступных наработок может вызвать проблемы. Предположим, вы клонировали репозиторий с сервера и сделали какую-то работу. И ваша история коммитов выглядит так:



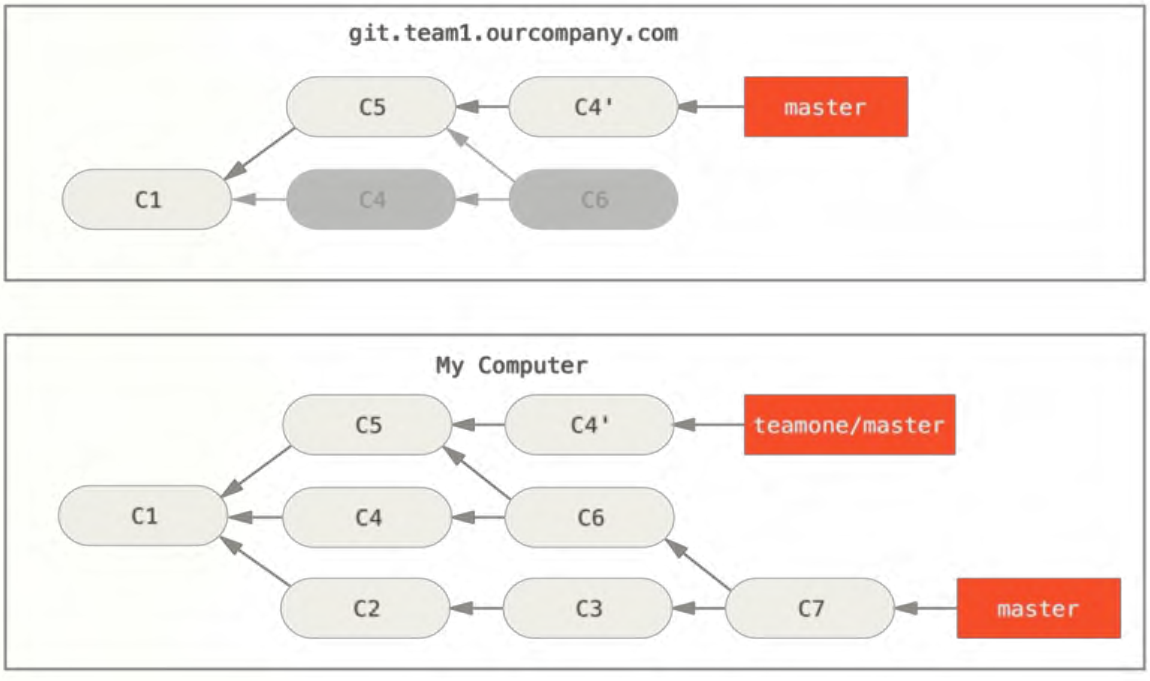
Клонирование репозитория и выполнение в нём какой-то работы/

Теперь кто-то другой внёс свои изменения, слил их и отправил на сервер. Вы стягиваете их к себе, включая новую удалённую ветку, что изменяет вашу историю следующим образом:



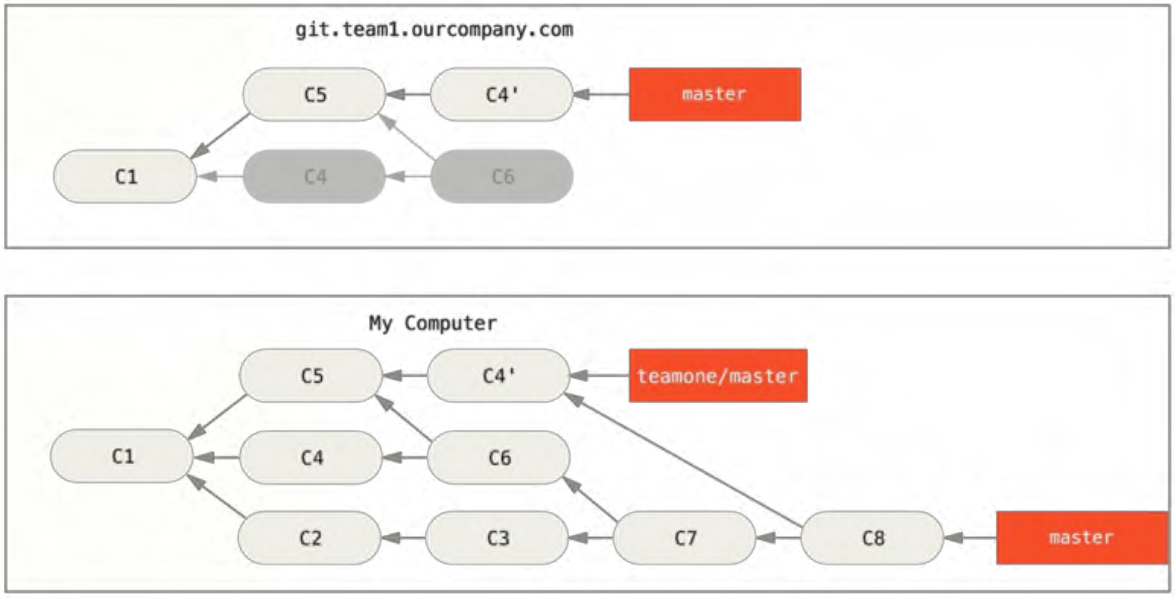
Извлекаем ещё коммиты и сливаем их со своей работой

Затем автор коммита слияния решает вернуться назад и перебазировать свою ветку; выполнив git push --force, он перезаписывает историю на сервере. При получении изменений с сервера вы получите и новые коммиты.



Кто-то выложил перебазированные коммиты, отменяя коммиты, на которых основывалась ваша работа/

Теперь вы оба в неловком положении. Если вы выполните git pull, вы создадите коммит слияния, включающий обе линии истории, и ваш репозиторий будет выглядеть следующим образом:



Вы снова выполняете слияние для той же самой работы в новый коммит слияния.

Если вы посмотрите git log в этот момент, вы увидите два коммита с одинаковыми авторами, датой и сообщением, что может сбить с толку. Помимо этого, если вы отправите свою историю на удалённый сервер в таком состоянии, вы вернёте все эти перебазированные коммиты на сервер, что ещё больше всех запутает. Логично предположить, что разработчик не хочет, чтобы C4 и C6 были в истории, и именно поэтому она перебазируется в первую очередь.

Меняя базу, меняй основание

Если вы попали в такую ситуацию, у Git есть особая магия чтобы вам помочь. Если кто-то в вашей команде форсирует отправку изменений на сервер, переписывающих работу, на которых базировалась ваша работа, то ваша задача будет состоять в определении того, что именно было ваше, а что было переписано **ими**.

Оказывается, что помимо контрольной суммы коммита SHA-1, Git также вычисляет контрольную сумму отдельно для патча, входящего в этот коммит. Это контрольная сумма называется «patch-id».

Если вы скачаете перезаписанную историю и перебазируете её поверх новых коммитов вашего коллеги, в большинстве случаев Git успешно определит, какие именно изменения были внесены вами, и применит их поверх новой ветки.

Перемещение vs. Слияние

Теперь, когда вы увидели перемещение и слияние в действии, вы можете задаться вопросом, что из них лучше. Прежде чем ответить на этот вопрос, давайте вернёмся немного назад и поговорим о том, что означает история.

Одна из точек зрения заключается в том, что история коммитов в вашем репозитории — это **запись того, что на самом деле произошло**. Это исторический документ, ценный сам по себе, и его нельзя подделывать. С этой точки зрения изменение истории коммитов практически кощунственно; вы **лжёте** о том, что на самом деле произошло. Но что, если произошла путаница в коммитах слияния? Если это случается, репозиторий должен сохранить это для потомков.

Противоположная точка зрения заключается в том, что история коммитов — это **история того, как был сделан ваш проект**. Вы не публикуете первый черновик книги или инструкции по поддержке вашего программного обеспечения, так как это нуждается в тщательном редактировании. Сторонники этого лагеря считают использование инструментов rebase и filter-branch способом рассказать историю проекта наилучшим образом для будущих читателей.

Теперь к вопросу о том, что лучше — слияние или перебазирование: надеюсь, вы видите, что это не так просто. Git — мощный инструмент, позволяющий вам делать многое с вашей историей, однако каждая команда и каждый проект индивидуален. Теперь, когда вы знаете, как работают оба эти приёма, выбор — какой из них будет лучше в вашей ситуации — зависит от вас.

При этом вы можете взять лучшее от обоих миров: использовать перебазирование для наведения порядка в истории ваших локальных изменений, но никогда не применять его для уже отправленных куда-нибудь изменений.

5. Инструменты Git

Выбор ревизии

Git позволяет различными способами указать коммиты или их диапазоны. Эти способы не всегда очевидны, но их полезно знать.

Одиночные ревизии

Конечно, вы можете ссылаться на коммит по его SHA-1 хешу, но существуют более удобные для человека способы. В данном разделе описываются различные способы обращения к одному коммиту.

Сокращённый SHA-1

Git достаточно умен, чтобы понять какой коммит имеется в виду по нескольким первым символам его хеша, если указанная часть SHA-1 имеет в длину по крайней мере четыре символа и однозначна — то есть в текущем репозитории существует только один объект с таким частичным SHA-1.

Например, предположим, чтобы найти некоторый коммит, вы выполнили команду git log и нашли коммит, в которой добавили определённую функциональность:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

```
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add some blame and merge stuff
```

Предположим, что в нашем примере это коммит 1c002dd….. Если вы хотите выполнить для него git show, то следующие команды эквиваленты (предполагается, что сокращения однозначны):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git может вычислить уникальные сокращения для ваших значений SHA-1. Если вы передадите опцию --abbrev-commit команде git log, в выводе будут использоваться сокращённые значения, сохраняющие уникальность; по умолчанию используется семь символов, но для сохранения уникальности SHA-1 могут использоваться более длинные значения.

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d Change the version number
085bb3b Remove unnecessary test code
a11bef0 Initial commit
```

Обычно от восьми до десяти символов более чем достаточно для сохранения уникальности значений в проекте.

Например, в ядре Linux, который является довольно большим проектом с более чем 450 тыс. коммитов и 3.6 млн объектов, отсутствуют объекты, чьи SHA-1 совпадают более чем в 11 первых символах.

Примечание	**Небольшое замечание о SHA-1** Большинство людей в этом месте начинают беспокоиться о том, что будет, если у них в репозитории случайно появятся два объекта с одинаковыми значениями SHA-1. Что тогда? Если вы вдруг зафиксируете объект, который имеет такое же значение SHA-1, как и предыдущий объект в вашем репозитории, Git увидит этот предыдущий

объект в своей базе и посчитает, что он уже был записан. Если вы позже попытаетесь переключиться на этот объект, то вы всегда будете получать данные первого объекта.

Однако, вы должны осознавать, насколько маловероятен такой сценарий. Длина SHA-1 составляет 20 байт или 160 бит. Количество случайно хешированных объектов, необходимых для достижения 50% вероятности возникновения коллизии, равно примерно 2^{80}. (формула для определения вероятности возникновения коллизии p = (n(n-1)/2) * (1/2^160)). 2^{80} — это 1.2×10^{24}, или 1 миллион миллиардов миллиардов, что в 1200 раз больше количества песчинок на земле.

Приведём пример, чтобы дать вам представление, чего будет стоить получение коллизии SHA-1. Если бы все 6.5 миллиардов человек на Земле были программистами, и ежесекундно каждый из них производил количество кода, эквивалентное всей истории ядра Linux (3.6 миллиона Git-объектов), и отправлял его в один огромный Git репозитории, то потребовалось бы около 2 лет, пока этот репозиторий накопил бы количество объектов, достаточное для 50% вероятности возникновения SHA-1 коллизии. Более вероятно, что каждый член вашей команды в одну и туже ночь будет атакован и убит волками в несвязанных друг с другом происшествиях.

Если выделить на это несколько тысяч долларов вычислительной мощности, можно будет синтезировать два файла с одним и тем же хешем, что было доказано проектом https://shattered.io/ в феврале 2017 года. Git движется к использованию SHA256 в качестве алгоритма хеширования по умолчанию, который намного более устойчив к атакам с коллизиями и имеет код, помогающий смягчить эту атаку (хотя он не может полностью ее устранить).

Ссылки на ветки

Для наиболее простого способа указать коммит требуется существование ветки, указывающей на этот коммит. Тогда вы можете использовать имя ветки в любой команде Git, которая ожидает коммит или значение SHA-1. Например, если вы хотите просмотреть последний коммит в ветке, то следующие команды эквивалентны (предполагается, что ветка topic1 указывает на коммит ca82a6d):

```
$ git show ca82a6dff817ec66f44342007202690a93763949

$ git show topic1
```

Если вы хотите узнать SHA-1 объекта, на который указывает ветка, или увидеть к чему сводятся все примеры в терминах SHA-1, то вы можете воспользоваться служебной командой Git, называемой rev-parse. Служебные команды подробно рассмотрены в главе Git изнутри; в основном, команда rev-parse существует для низкоуровневых операций и не предназначена для ежедневного использования. Однако она может быть полезна, когда вам нужно увидеть, что в действительности происходит. Теперь вы можете выполнить rev-parse для вашей ветки.

```
$ git rev-parse topic1

ca82a6dff817ec66f44342007202690a93763949
```

RefLog-сокращения

Одна из вещей, которую Git делает в фоновом режиме, является ведение журнала ссылок, в котором сохраняется то, куда указывали HEAD и ветки за последние несколько месяцев.

Для просмотра этого журнала используется команда git reflog:

```
$ git reflog

734713b HEAD@{0}: commit: Fix refs handling, add gc auto, update tests

d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.

1c002dd HEAD@{2}: commit: Add some blame and merge stuff

1c36188 HEAD@{3}: rebase -i (squash): updating HEAD

95df984 HEAD@{4}: commit: # This is a combination of two commits.

1c36188 HEAD@{5}: rebase -i (squash): updating HEAD

7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Каждый раз, когда по каким-то причинам изменяется вершина вашей ветки, Git сохраняет информацию об этом в эту временную историю. И вы можете указывать старые коммиты, используя эти данные. Например, чтобы посмотреть, куда ссылался указатель HEAD пять шагов назад, используйте ссылку @{5}, которую можно увидеть в выводимых данных команды reflog:

```
$ git show HEAD@{5}
```

Этот синтаксис используется и в случае, когда требуется посмотреть, в каком состоянии пребывала ветка некоторое время назад. В частности, чтобы увидеть где была ветка master вчера, следует написать:

```
$ git show master@{yesterday}
```

Вы увидите, что было на вершине ветки вчера. Такой способ работает только для данных, которые всё ещё содержатся в вашем журнале ссылок, поэтому вы не можете использовать её для коммитов, которые старше нескольких месяцев.

Для просмотра журнала ссылок в формате, похожем на вывод git log, вы можете выполнить git log -g:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: Fix refs handling, add gc auto, update tests
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

Важно отметить, что информация в журнале ссылок строго локальная — это лог того, что вы делали в вашем репозитории. Ссылки не будут такими же в других копиях репозитория; а сразу после первоначального клонирования репозитория, у вас будет пустой журнал ссылок, так как никаких действий в вашем репозитории пока не производилось. Команда git show HEAD@{2.months.ago} будет работать только если вы клонировали проект по крайней мере два месяца назад — если вы клонировали его пять минут назад, то не получите никаких результатов.

Подсказка	**Воспринимайте reflog Git как историю командной строки** Если у вас есть опыт работы с UNIX или Linux, можете думать о reflog как об истории командной строки Git, которая подчеркивает, что то, что там есть, явно актуально только для вас и вашего «сеанса» и не имеет ничего общего с кем-либо еще, кто может работать на той же машине.
Примечание	**Экранирование фигурных скобок в PowerShell** При использовании PowerShell фигурные скобки, такие как { и }, являются специальными символами и должны быть экранированы. Вы можете экранировать их с помощью апострофа ` или поместить ссылку на коммит в кавычки: $ git show HEAD@{0}  # НЕ будет работать $ git show HEAD@`{0`} # OK $ git show "HEAD@{0}" # OK

Ссылки на предков

Ещё один популярный способ указать коммит — это использовать её родословную. Если вы поместите ^ в конце ссылки, Git поймёт, что нужно использовать родителя этого коммита. Предположим, история вашего проекта выглядит следующим образом:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b Fix refs handling, add gc auto, update tests
*   d921970 Merge commit 'phedders/rdocs'
```

```
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd Add some blame and merge stuff
|/
* 1c36188 Ignore *.gem
* 9b29157 Add open3_detach to gemspec file list
```

Для просмотра предыдущего коммита достаточно написать HEAD^, что означает «родитель HEAD»:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Примечание	**Экранирование карета в Windows**
	В командной строке Windows (cmd.exe) ^ является специальным символом и требует другого обращения. Вы можете либо удвоить его, либо поместить ссылку на коммит в кавычки:
	`$ git show HEAD^ # НЕ будет работать в Windows`
	`$ git show HEAD^^ # OK`
	`$ git show "HEAD^" # OK`

Также вы можете указать число после ^ — например, d921970^2 означает «второй родитель коммита d921970». Такой синтаксис полезен только для коммитов слияния, которые имеют больше одного родителя. Первым родителем является ветка, в которую вы выполняли слияние, а вторым — коммит в ветке, которую вы сливали:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
```

```
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

Второе важное обозначение для указания предков — это символ тильда ~. Он также соответствует ссылке на первого родителя, поэтому HEAD~ и HEAD^ эквивалентны. Различия становятся заметными, когда вы указываете число. HEAD~2 означает «первый родитель первого родителя» или «дедушка» — при этом происходит переход от заданного предка вглубь указанное число раз. К примеру, для показанной ранее истории, коммитом HEAD~3 будет:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    Ignore *.gem
```

То же самое можно записать как HEAD~~~, что также является первым родителем первого родителя первого родителя:

```
$ git show HEAD~~~
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500
```

```
Ignore *.gem
```

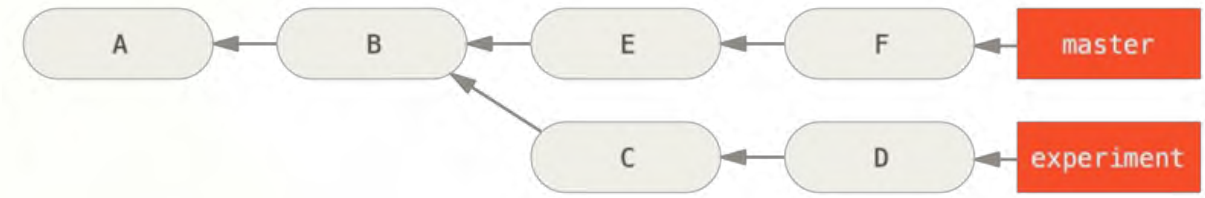
Вы также можете совмещать эти обозначения — можно получить второго родителя предыдущей ссылки (предполагается, что это коммит слияния) используя запись HEAD~3^2, и так далее.

Диапазоны коммитов

Теперь вы умеете указывать отдельные коммиты, давайте посмотрим, как указывать диапазоны коммитов. Это, в частности, полезно для управления вашими ветками — если у вас есть множество веток, вы можете использовать указание диапазонов коммитов для ответа на вопрос «Что было сделано в этой ветке, что я ещё не слил в основную ветку?»

Две точки

Наиболее часто для указания диапазона коммитов используется синтаксис с двумя точками. Таким образом, вы, по сути, просите Git включить в диапазон коммитов только те, которые достижимы из одной, но не достижимы из другой. Для примера предположим, что ваша история выглядит, как на рисунке 136.



Пример истории для выбора диапазонов коммитов

Вы хотите посмотреть, что находится в вашей экспериментальной ветке, которая ещё не была слита в основную. Вы можете попросить Git отобразить в логе только такие коммиты, используя запись master..experiment — она означает «все коммиты, которые доступны из ветки experiment, но не доступны из ветки master». Для краткости и наглядности в этих примерах вместо настоящего вывода лога мы будем использовать для коммитов их буквенные обозначения из диаграммы, располагая их в должном порядке:

```
$ git log master..experiment

D

C
```

С другой стороны, если вы хотите, наоборот, увидеть все коммиты ветки master, которых нет в ветке experiment, вы можете поменять имена веток в команде. При использовании записи experiment..master будут отображены все коммиты ветки master, недоступные из ветки experiment:

```
$ git log experiment..master
F
E
```

Это полезно если вы хотите сохранить ветку experiment в актуальном состоянии и просмотреть, какие изменения нужно в нее слить. Другое частое использование такого синтаксиса — просмотр того, что будет отправлено в удалённый репозиторий.

```
$ git log origin/master..HEAD
```

Такая команда покажет вам все коммиты вашей текущей ветки, которые отсутствуют в ветке master удалённого репозитория origin. Если вы выполните git push, находясь на ветке, отслеживающей origin/master, то коммиты, отображённые командой git log origin/master..HEAD, будут теми коммитами, которые отправятся на сервер. Вы также можете опустить одну из частей в такой записи, Git будет считать её равной HEAD. Например, вы можете получить такой же результат как в предыдущем примере, выполнив git log origin/master.. — Git подставит HEAD, если одна часть отсутствует.

Множественная выборка

Запись с двумя точками полезна как сокращение, но, возможно, вы захотите использовать более двух веток для указания нужной ревизии, например, для того, чтобы узнать какие коммиты присутствуют в любой из нескольких веток, но отсутствуют в ветке, в которой вы сейчас находитесь. Git позволяет сделать это, используя символ ^ или опцию --not, перед любой ссылкой, доступные коммиты из которой вы не хотите видеть. Таким образом, следующие три команды эквивалентны:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Этот синтаксис удобен, так как позволяет указывать в запросе более двух ссылок, чего не позволяет сделать синтаксис с двумя точками. Например, если вы хотите увидеть все коммиты, доступные из refA и refB,

но не доступные из refC, вы можете использовать одну из следующих команд:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Это делает систему запросов ревизий более мощной и должно помочь вам лучше понять, что содержится в вашей ветке.

Три точки

Последний основной способ выбора ревизий — это синтаксис с тремя точками, который обозначает все коммиты, доступные хотя бы из одной ссылки, но не из обеих сразу. Вспомните пример истории коммитов в Пример истории для выбора диапазонов коммитов. Если вы хотите узнать какие коммиты есть либо в ветке master, либо в experiment, но не в обеих сразу, вы можете выполнить:

```
$ git log master...experiment
F
E
D
C
```

Эта команда снова выводит обычный журнал коммитов, но в нем содержится информация только об этих четырёх коммитах, традиционно отсортированная по дате коммитов.

В таких случаях с командой log часто используют опцию --left-right, которая отображает сторону диапазона, с которой был сделан каждый из коммитов. Это делает данную информацию более полезной:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

С помощью этих инструментов вам будет намного проще указать Git какой коммит или коммиты вы хотите изучить.

Интерактивное индексирование

Git поставляется вместе со скриптами, которые упрощают выполнение некоторых задач из командной строки. В этом разделе мы рассмотрим несколько интерактивных команд, которые могут упростить создание коммитов, позволяя включать в них только определённый набор файлов и их частей. Эти инструменты очень полезны, если вы изменили множество файлов, а затем решили, что хотите чтобы эти изменения были в нескольких маленьких понятных коммитах, а не в одном большом и запутанном. Таким способом вы сможете гарантировать, что ваши коммиты представляют логически разделённые изменения и могут быть легко прорецензированы вашими коллегами.

Если вы выполните git add с опцией -i или --interactive, Git перейдёт в интерактивный консольный режим, отобразив что-то подобное:

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now>
```

Вы можете видеть, что эта команда показывает вашу область подготовленных изменений в уникальном представлении — вообще говоря, ту же информацию вы получите с помощью команды git status, но несколько более сжато и информативно. Эта команда показывает проиндексированные изменения слева, а непроиндексированные — справа.

Затем следует раздел со списком команд. С их помощью вы можете выполнить множество вещей — добавить или исключить файлы из индекса, добавить в индекс части файлов, добавить в индекс неотслеживаемые файлы и просмотреть проиндексированные изменения.

Добавление и удаление файлов из индекса

Если вы введете 2 или u в поле ввода What now>, скрипт спросит у вас какие файлы вы хотите добавить в индекс:

```
What now> u
           staged     unstaged path
  1:    unchanged       +0/-1 TODO
  2:    unchanged       +1/-1 index.html
  3:    unchanged       +5/-1 lib/simplegit.rb
Update>>
```

Для добавления в индекс файлов TODO и index.html, вы можете ввести их номера:

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged       +0/-1 TODO
* 2:    unchanged       +1/-1 index.html
  3:    unchanged       +5/-1 lib/simplegit.rb
Update>>
```

Символ * у каждого из этих файлов означает, что файл выбран для индексирования. Если вы нажмёте Enter, не вводя ничего в поле ввода Update>>, Git добавит в индекс всё, чтобы было выбрано ранее:

```
Update>>
updated 2 paths


*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> s
           staged     unstaged path
  1:       +0/-1      nothing TODO
  2:       +1/-1      nothing index.html
```

```
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Как вы можете заметить, сейчас файлы TODO и index.html добавлены в индекс, а файл simplegit.rb всё ещё нет. Если вы в этот момент хотите исключить файл TODO из индекса, вы можете использовать опции 3 или r (для выполнения revert):

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> r
           staged     unstaged path
  1:         +0/-1      nothing TODO
  2:         +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:         +0/-1      nothing TODO
  2:         +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Посмотрев снова на состояние вашего рабочего каталога Git, вы увидите, что файл TODO исключён из индекса:

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> s
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:         +1/-1      nothing index.html
```

```
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Для того, чтобы посмотреть изменения, которые вы добавили в индекс, вы можете использовать команду 6 и d (для выполнения diff). Она покажет вам список добавленных в индекс файлов, и вы можете выбрать один из них, для которого вы хотите увидеть добавленные в индекс изменения. Это очень похоже на вывод команды git diff --cached:

```
*** Commands ***
  1: [s]tatus     2: [u]pdate      3: [r]evert     4: [a]dd untracked
  5: [p]atch      6: [d]iff        7: [q]uit       8: [h]elp
What now> d
          staged     unstaged path
  1:       +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

  <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

  <script type="text/javascript">
```

Использование этих основных команд в интерактивном режиме команды add, может упростить вам работу с вашей областью подготовленных изменений.

Индексирование по частям

В Git существует возможность индексировать не только файлы целиком, но и некоторые их части. Например, если вы сделали в файле simplegit.rb два изменения и хотите добавить в индекс только одно из них, добиться этого в Git очень легко. В поле ввода в режиме интерактивного индексирования введите 5 или p (для выполнения patch). Git спросит у вас какие файлы вы хотите добавить в индекс частично; а затем для каждой части выбранных файлов он будет показывать изменения в ней и спрашивать хотите ли вы добавить в индекс эту часть:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
  end


  def log(treeish = 'master')
-   command("git log -n 25 #{treeish}")
+   command("git log -n 30 #{treeish}")
  end


  def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

В этой точке у вас есть множество вариантов дальнейших действий. Если вы введёте ?, Git отобразит, что именно вы можете сделать:

Добавить в индекс эту часть [y,n,a,d,/,j,J,g,e,?]? ?

y - добавить в индекс эту часть

n - не добавлять в индекс эту часть

a - добавить в индекс эту и все оставшиеся в этом файле части

d - не добавлять в индекс эту и все оставшиеся в этом файле части

g - перейти к некоторой части файла (g - показывает список частей и затем выполняет переход, g<N> - перейти к части N)

/ - найти часть, соответствующую регулярному выражению

j - отложить принятие решения по этой части, перейти к следующей части, решение по которой не принято

J - отложить принятие решения по этой части, перейти к следующей части

k - отложить принятие решения по этой части, перейти к предыдущей части, решение по которой не принято

K - отложить принятие решения по этой части, перейти к предыдущей части

s - разбить текущую часть на части меньшего размера

e - вручную отредактировать текущую часть

? - отобразить помощь

Обычно вы будете вводить y или n, если вы хотите индексировать каждую часть по отдельности, но индексация всех частей в некоторых файлах или откладывание решения по индексацию части также может быть полезным. Если вы добавили в индекс одну часть файла, но не добавили другую, состояние вашего рабочего каталога будет подобно приведённому далее:

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:      +1/-1      nothing index.html
  3:      +1/-1        +4/-0 lib/simplegit.rb
```

Обратите внимание на состояние файла simplegit.rb. Оно говорит вам, что часть строк файла добавлена в индекс, а часть нет. Таким образом, вы частично проиндексировали этот файл. В данный момент вы можете выйти из интерактивного режима команды git add и выполнить git commit, чтобы зафиксировать частично проиндексированные файлы.

Также вам не обязательно находиться в интерактивном режиме индексирования файлов для выполнения частичной индексации файлов — вы также можете запустить её, используя команды git add -p или git add --patch.

Более того, вы можете использовать работу с отдельными частями файлов для частичного восстановления файлов с помощью команды reset --patch, для переключения частей файлов с помощью команды checkout --patch и для припрятывания частей файлов с помощью stash save --patch. Мы

рассмотрим каждую из этих команд более подробно, когда будем изучать более продвинутые варианты их использования.

Припрятывание и очистка

Часто пока вы работаете над одной частью вашего проекта и всё находится в беспорядке, у вас возникает желание сменить ветку и поработать над чем-то ещё. Сложность при этом заключается в том, что вы не хотите фиксировать наполовину сделанную работу только для того, чтобы иметь возможность вернуться к ней позже. Справиться с ней помогает команда git stash.

Операция stash берет изменённое состояние вашего рабочего каталога, то есть изменённые отслеживаемые файлы и проиндексированные изменения, и сохраняет их в хранилище незавершённых изменений, которые вы можете в любое время применить обратно.

Примечание	**Переход на git stash push** В конце октября 2017 года в списке рассылки Git проходило обширное обсуждение, по итогам которого команда git stash save признана устаревшей в пользу существующей альтернативы git stash push. Основная причина этого заключается в том, что в git stash push есть возможность сохранить выбранные **спецификации пути**, что не поддерживает git stash save. Команда git stash save не исчезнет в ближайшее время, поэтому не беспокойтесь о её внезапной пропаже. Но вы можете начать переход на push для использования новой функциональности.

Припрятывание ваших наработок

Для примера, предположим, что вы перешли в свой проект, начали работать над несколькими файлами и, возможно, добавили в индекс изменения одного из них. Если вы выполните git status, то увидите ваше изменённое состояние:

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
 modified:   index.html


Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git checkout -- <file>..." to discard changes in working directory)


 modified:   lib/simplegit.rb
```

Теперь вы хотите сменить ветку, но пока не хотите фиксировать ваши текущие наработки; поэтому вы припрячете эти изменения. Для того, чтобы припрятать изменение в выделенное для этого специальное хранилище, выполните git stash или git stash push:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 Create index file"
HEAD is now at 049d078 Create index file
(To restore them type "git stash apply")
```

Теперь вы можете увидеть, что рабочая копия не содержит изменений:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

В данный момент вы можете легко переключать ветки и работать в любой; ваши изменения сохранены. Чтобы посмотреть список припрятанных изменений, вы можете использовать git stash list:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
```

В данном примере, предварительно были припрятаны два изменения, поэтому теперь вам доступны три различных отложенных

наработки. Вы можете применить только что припрятанные изменения, используя команду, указанную в выводе исходной команды: git stash apply. Если вы хотите применить одно из предыдущих припрятанных изменений, вы можете сделать это, используя его имя, вот так: git stash apply stash@{2}. Если вы не укажете имя, то Git попытается восстановить самое последнее припрятанное изменение:

```
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

 modified:   index.html
 modified:   lib/simplegit.rb

no changes added to commit (use "git add" and/or "git commit -a")
```

Как видите, Git восстановил в файлах изменения, которые вы отменили ранее, когда прятали свои наработки. В данном случае при применении отложенных наработок ваш рабочий каталог был без изменений, а вы пытались применить их в той же ветке, в которой вы их и сохранили; но отсутствие изменений в рабочем каталоге и применение их в той же ветке не являются необходимыми условиями для успешного восстановления припрятанных наработок. Вы можете припрятать изменения, находясь в одной ветке, а затем переключиться на другую и попробовать восстановить эти изменения. Также при восстановлении припрятанных наработок в вашем рабочем каталоге могут присутствовать изменённые и незафиксированные файлы — Git выдаст конфликты слияния, если не сможет восстановить какие-то наработки.

Спрятанные изменения будут применены к вашим файлам, но файлы, которые вы ранее добавляли в индекс, не будут добавлены туда снова. Для того, чтобы это было сделано, вы должны запустить git stash apply с опцией --index, при которой команда попытается восстановить изменения в индексе. Если вы выполните команду таким образом, то полностью восстановите ваше исходное состояние:

```
$ git stash apply --index
On branch master
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 modified:   index.html

Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git checkout -- <file>..." to discard changes in working directory)

 modified:   lib/simplegit.rb
```

Команда apply только пытается восстановить припрятанные наработки — при этом они останутся в хранилище. Для того, чтобы удалить их, вы можете выполнить git stash drop, указав имя удаляемых изменений:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Вы также можете выполнить git stash pop, чтобы применить припрятанные изменения и тут же удалить их из хранилища.

Необычное припрятывание

У припрятанных изменений есть несколько дополнительных вариантов использования, которые также могут быть полезны. Первый — это использование довольно популярной опции --keep-index с командой git stash. Она просит Git не только припрятать то, что вы уже добавили в индекс, но одновременно оставить это в индексе.

```
$ git status -s
```

```
M  index.html
 M lib/simplegit.rb
```

```
$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

Другой распространённый вариант, который вы, возможно, захотите использовать — это припрятать помимо отслеживаемых файлов также и неотслеживаемые. По умолчанию git stash будет сохранять только изменённые и проиндексированные **отслеживаемые** файлы. Если вы укажете опцию --include-untracked или -u, Git также припрячет все неотслеживаемые файлы, которые вы создали. Однако включение этой опции по-прежнему не будет прятать файлы с явным игнорированием; чтобы дополнительно припрятать игнорируемые файлы, используйте --all (или просто -a).

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt
```

```
$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file
```

```
$ git status -s
```

```
$
```

И наконец, если вы укажете флаг --patch, Git не будет ничего прятать, а вместо этого в интерактивном режиме спросит вас о том, какие из изменений вы хотите припрятать, а какие оставить в вашем рабочем каталоге.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
        return `#{git_cmd} 2>&1`.chomp
      end
    end
+
+   def show(treeish = 'master')
+     command("git show #{treeish}")
+   end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Создание ветки из припрятанных изменений

Если вы спрятали некоторые изменения, оставили их на время, а сами продолжили работать в той же ветке, у вас могут возникнуть проблемы с восстановлением наработок. Если восстановление будет затрагивать файл, который уже был изменён с момента сохранения

наработок, то вы получите конфликт слияния и должны будете попытаться разрешить его. Если вам нужен более простой способ снова протестировать припрятанные изменения, вы можете выполнить команду git stash branch, которая создаст для вас новую ветку, перейдёт на коммит, на котором вы были, когда прятали свои наработки, применит на нём эти наработки и затем, если они применились успешно, удалит эти припрятанные изменения:

```
$ git stash branch testchanges
M       index.html
M       lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 modified:   index.html

Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git checkout -- <file>..." to discard changes in working directory)

 modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Это удобное сокращение для того, чтобы легко восстановить припрятанные изменения и поработать над ними в новой ветке.

Очистка рабочего каталога

Наконец, у вас может возникнуть желание не прятать некоторые из изменений или файлов в вашем рабочем каталоге, а просто избавиться от них. Команда git clean сделает это для вас.

Одной из распространённых причин для этого может быть удаление мусора, который был сгенерирован при слиянии или внешними утилитами, или удаление артефактов сборки в процессе её очистки.

Вам нужно быть очень аккуратными с этой командой, так как она предназначена для удаления неотслеживаемых файлов из вашего рабочего каталога. Даже если вы передумаете, очень часто нельзя восстановить содержимое таких файлов. Более безопасным вариантом является использование команды git stash --all для удаления всего, но с сохранением этого в виде припрятанных изменений.

Предположим, вы хотите удалить мусор и очистить ваш рабочий каталог; вы можете сделать это с помощью git clean. Для удаления всех неотслеживаемых файлов в вашем рабочем каталоге, вы можете выполнить команду git clean -f -d, которая удалит все файлы и также все каталоги, которые в результате станут пустыми. Параметр -f (сокращение от слова force — заставить) означает принудительное удаление, подчеркивая, что вы действительно хотите это сделать, и требуется, если переменная конфигурации Git clean.requireForce явным образом не установлена в false.

Если вы хотите только посмотреть, что будет сделано, вы можете запустить команду с опцией -n, которая означает «имитируй работу команды и скажи мне, что ты **будешь** удалять».

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

По умолчанию команда git clean будет удалять только неотслеживаемые файлы, которые не добавлены в список игнорируемых. Любой файл, который соответствует шаблону в вашем .gitignore, или другие игнорируемые файлы не будут удалены. Если вы хотите удалить и эти файлы (например, удалить все .o-файлы, генерируемые в процессе сборки, и таким образом полностью очистить сборку), вы можете передать команде очистки опцию -x.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/
```

```
$ git clean -n -d
Would remove build.TMP
Would remove tmp/
```

```
$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

Если вы не знаете, что сделает при запуске команда git clean, всегда сначала выполняйте её с опцией -n, чтобы проверить дважды, перед заменой -n на -f и выполнением настоящей очистки. Другой способ, который позволяет вам более тщательно контролировать сам процесс — это выполнение команды с опцией -i (в «интерактивном» режиме).

Ниже выполнена команда очистки в интерактивном режиме.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask each
5: quit
    6: help
What now>
```

Таким образом, вы можете просмотреть каждый файл индивидуально или указать шаблоны для удаления в интерактивном режиме.

Примечание	Существует причудливая ситуация, когда вам, возможно, придется проявить особую настойчивость, попросив Git очистить ваш рабочий каталог. Если вы оказались в рабочем каталоге, в который вы скопировали или клонировали другие репозитории Git (возможно, в виде подмодулей), даже git clean -fd откажется удалить эти каталоги. В таких случаях вам

	нужно добавить второй параметр -f для акцентирования.

Подпись

Благодаря шифрованию система Git является безопасной, но полностью она не защищена. На случай, если вы берете у кого-то в интернете результаты его работы и хотите проверить, что коммиты действительно получены из доверенного источника, в Git есть несколько способов подписать и проверить исходники, используя GPG.

Введение в GPG

Во-первых, если вы хотите что-либо подписать, вам необходим настроенный GPG и персональный ключ.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   2048R/0A46826A 2014-06-04
uid                  Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

Если у вас нет ключа, то можете сгенерировать его командой gpg --gen-key.

```
$ gpg --gen-key
```

Если у вас есть приватный ключ для подписи, вы можете настроить Git так, чтобы этот ключ использовался для подписи, установив значение параметра user.signingkey.

```
$ git config --global user.signingkey 0A46826A
```

Теперь, если вы захотите, Git будет по умолчанию использовать этот ключ для подписи тегов и коммитов.

Подпись тегов

Если вы настроили приватный ключ GPG, то можете использовать его для подписи новых тегов. Для этого вы должны использовать опцию -s вместо -a:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'


You need a passphrase to unlock the secret key for
```

```
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Если теперь для этого тега вы выполните git show, то увидите прикреплённую к нему свою GPG подпись:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Проверка тегов

Для проверки подписанного тега вы можете воспользоваться командой git tag -v [tag-name]. Она использует GPG для проверки подписи. Чтобы всё это правильно работало нужно, чтобы публичный ключ автора присутствовал в вашем хранилище ключей:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

Если у вас отсутствует публичный ключ автора, вы увидите что-то подобное:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

Подпись коммитов

В новых версиях Git (начиная с v1.7.9), вы также можете подписывать отдельные коммиты. Если вам нужно подписывать непосредственно сами коммиты, а не теги, вы можете передать команде git commit опцию -S.

```
$ git commit -a -S -m 'Signed commit'


You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04


[master 5c3386c] Signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

Для просмотра и проверки таких подписей у команды git log есть параметр --show-signature.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700


    Signed commit
```

Также вы можете, используя формат с %G?, настроить git log так, чтобы он проверял и отображал любую обнаруженную подпись.

```
$ git log --pretty="format:%h %G? %aN  %s"
```

```
5c3386c G Scott Chacon  Signed commit
ca82a6d N Scott Chacon  Change the version number
085bb3b N Scott Chacon  Remove unnecessary test code
a11bef0 N Scott Chacon  Initial commit
```

В данном примере видно, что только последний коммит корректно подписан, а все предыдущие нет.

В Git, начиная с версии 1.8.3, команды git merge и git pull с помощью опции --verify-signatures можно заставить проверять и отклонять слияния, если коммит не содержит доверенной GPG подписи.

Если вы воспользуетесь этой опцией при слиянии с веткой, которая содержит неподписанные или некорректно подписанные коммиты, то слияние завершится ошибкой.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Если сливаемая ветка содержит только корректно подписанные коммиты, команда слияние сначала покажет все проверенные ей подписи, а затем выполнит само слияние.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Также с командой git merge вы можете использовать опцию -S, в этом случае полученный в результате слияния коммит будет подписан. В следующем примере выполняется и проверка каждого коммита сливаемой ветки, и подпись полученного в результате слияния коммита.

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com>
```

```
You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Каждый должен подписываться

Конечно, подписывать теги и коммиты это хорошая идея, но если вы решите воспользоваться ей в вашем обычном рабочем процессе, то должны удостовериться, что все в вашей команде понимают, как выполнять подпись. Если этого не сделать, то в итоге вам придётся потратить много времени, объясняя коллегами, как перезаписать их коммиты подписанными версиями. Удостоверьтесь, что вы разбираетесь в GPG и преимуществах, которые несут подписи, перед тем как использовать их как часть вашего рабочего процесса.

Поиск

Вне зависимости от размера кодовой базы, часто возникает необходимость поиска места вызова/определения функции или получения истории изменения метода. Git предоставляет несколько полезных утилит, с помощью которых легко и просто осуществлять поиск по коду и коммитам. Мы обсудим некоторые из них.

Команда git grep

Git поставляется с командой grep, которая позволяет легко искать в истории коммитов или в рабочем каталоге по строке или регулярному выражению. В следующих примерах мы обратимся к исходному коду самого Git.

По умолчанию эта команда ищет по файлам в рабочем каталоге. В качестве первого варианта вы можете использовать любой из параметров -n или` --line-number`, чтобы распечатать номера строк, в которых Git нашел совпадения:

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
```

```
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:826:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:206:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:482:             if (gmtime_r(&now, &now_tm))
date.c:545:             if (gmtime_r(&time, tm)) {
date.c:758:             /* gmtime_r() in match_digit() may have clobbered it */
git-compat-util.h:1138:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:1140:#define gmtime_r git_gmtime_r
```

В дополнение к базовому поиску, показанному выше, git grep поддерживает множество других интересных параметров.

Например, вместо того, чтобы печатать все совпадения, вы можете попросить git grep обобщить выводимые командой данные, показав только те файлы, в которых обнаружены совпадения, вместе с количеством этих совпадений в каждом файле. Для этого потребуется параметр ` -c` или --count:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:3
git-compat-util.h:2
```

Если вас интересует **контекст** строки поиска, можно показать метод или функцию, в котором присутствует совпадение с помощью параметра -p или` --show-function`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(timestamp_t num, char c, const char *date,
date.c:         if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int *tm_gmt)
date.c:         if (gmtime_r(&time, tm)) {
date.c=int parse_date_basic(const char *date, timestamp_t *timestamp, int *offset)
date.c:         /* gmtime_r() in match_digit() may have clobbered it */
```

Здесь вы можете видеть, что gmtime_r вызывается из функций match_multi_number и match_digit в файле date.c (третье отображаемое совпадение представляет собой только строку, появившуюся в комментарии).

Вы также можете искать сложные комбинации строк, используя опцию --and, которая гарантирует, что будут отображены только строки, имеющие сразу несколько совпадений. Например, давайте поищем любые строки, которые определяют константу, имя которой содержит **любую** из подстрок «LINK» или «BUF_MAX», особенно в более старой версии кодовой базы Git, представленной тегом v1.8.0 (мы добавим параметры --break и --heading, которые помогут вывести результаты в более читаемом виде):

```
$ git grep --break --heading \
    -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK  0160000
74:#define S_ISGITLINK(m)      (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
```

```
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS


v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)


v1.8.0:symlinks.c
53:#define FL_SYMLINK  (1 << 2)


v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

Команда git grep имеет несколько преимуществ перед поиском с помощью таких команд, как grep и ack. Во-первых, она действительно быстрая, во-вторых — git grep позволяет искать не только в рабочем каталоге, но и в любом другом дереве Git. Как вы видели, в прошлом примере мы искали в старой версии исходных кодов Git, а не в текущем снимке файлов.

Поиск в журнале Git

Возможно, вы ищете не **где** присутствует некоторое выражение, а **когда** оно существовало или было добавлено. Команда git log обладает некоторыми мощными инструментами для поиска определённых коммитов по содержимому их сообщений или содержимому сделанных в них изменений.

Например, если вы хотите найти, когда была добавлена константа ZLIB_BUF_MAX, то вы можете с помощью опции -S попросить Git показывать только те коммиты, в которых была добавлена или удалена эта строка.

```
$ git log -S ZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Если мы посмотрим на изменения, сделанные в этих коммитах, то увидим, что в ef49a7a константа была добавлена, а в e01503b — изменена.

Если вам нужно найти что-то более сложное, вы можете с помощью опции -G передать регулярное выражение.

Поиск по журналу изменений строки

Другой, довольно продвинутый, поиск по истории, который бывает чрезвычайно полезным — поиск по истории изменений строки. Просто запустите git log с параметром -L, и он покажет вам историю изменения функции или строки кода в вашей кодовой базе.

Например, если мы хотим увидеть все изменения, произошедшие с функцией git_deflate_bound в файле zlib.c, мы можем выполнить git log -L :git_deflate_bound:zlib.c. Эта команда постарается определить границы функции, выполнит поиск по истории и покажет все изменения, которые были сделаны с функцией, в виде набора патчей в обратном порядке до момента создания функции.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-       return deflateBound(strm, size);
+       return deflateBound(&strm->z, size);
 }
```

```
commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+       return deflateBound(strm, size);
+}
+
```

Если для вашего языка программирования Git не умеет правильно определять функции и методы, вы можете передать ему регулярное выражение. Например, следующая команда выполнит такой же поиск, как и предыдущая git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c. Также вы можете передать интервал строк или номер определённой строки и в этом случае вы получите похожий результат.

Перезапись истории

Неоднократно при работе с Git, вам может потребоваться по какой-то причине внести исправления в историю коммитов. Одно из преимуществ Git заключается в том, что он позволяет вам отложить принятие решений на самый последний момент. Область индексирования позволяет вам решить, какие файлы попадут в коммит непосредственно перед его выполнением; благодаря команде git stash вы можете решить, что не хотите продолжать работу над какими-то изменениями; также можете внести изменения в сделанные коммиты так, чтобы они выглядели как будто они произошли по-

другому. В частности, можно изменить порядок коммитов, сообщения или изменённые в коммитах файлы, объединить вместе или разбить на части, полностью удалить коммит — но только до того, как вы поделитесь своими наработками с другими.

В этом разделе вы познакомитесь со способами решения всех этих задач и научитесь перед публикацией данных приводить историю коммитов в нужный вам вид.

Примечание	**Не отправляйте свои наработки, пока вы ими не довольны** Одно из основных правил Git заключается в том, что, так как большую часть работы вы делаете в своём локальном репозитории, то вы вольны переписывать свою историю **локально**. Однако, как только вы отправите свои наработки, то это уже совсем другая история и вам следует рассматривать отправленные изменения как финальные до тех пор, пока у вас не появится весомая причина что-то изменить. Если кратко, то вы должны воздержаться от отправки своих изменений, пока не будете полностью довольны и готовы поделиться ими со всем миром.

Изменение последнего коммита

Изменение вашего последнего коммита, наверное, наиболее частое исправление истории, которое вы будете выполнять. Наиболее часто с вашим последним коммитом вам будет нужно сделать две основные операции: изменить сообщение коммита или изменить только что сделанный снимок, добавив, изменив или удалив файлы.

Если вы хотите изменить только сообщение вашего последнего коммита, это очень просто:

```
$ git commit --amend
```

Эта команда откроет в вашем текстовом редакторе сообщение вашего последнего коммита для того, чтобы вы могли его исправить. Когда вы сохраните его и закроете редактор, будет создан новый коммит,

содержащий это сообщение, который теперь и будет вашим последним коммитом.

Если вы создали коммит и затем хотите изменить зафиксированный снимок, добавив или изменив файлы (возможно, вы забыли добавить вновь созданный файл, когда совершали изначальный коммит), то процесс выглядит в основном так же. Вы добавляете в индекс необходимые изменения, редактируя файл и выполняя для него git add или git rm для отслеживаемого файла, а последующая команда git commit --amend берет вашу текущую область подготовленных изменений и делает её снимок для нового коммита.

Вы должны быть осторожными, используя этот приём, так как при этом изменяется SHA-1 коммита. Поэтому, как и с операцией rebase — не изменяйте ваш последний коммит, если вы уже отправили его в общий репозиторий.

Подсказка	**Изменённый коммит может потребовать изменения сообщения коммита** При изменении коммита существует возможность изменить как его содержимое, так и сообщение коммита. Если в коммит внесены существенные изменения, то почти наверняка следует обновить и его сообщение, чтобы оно более точно отражало содержимое коммита. С другой стороны, если изменения незначительны (исправление опечаток, добавление в коммит забытого файла), то текущее сообщение вполне можно оставить; чтобы лишний раз не вызывать редактор, просто добавьте измененные файлы в индекс и выполните команду: $ git commit --amend --no-edit

Изменение сообщений нескольких коммитов

Для изменения коммита, расположенного раньше в вашей истории, вам нужно обратиться к более сложным инструментам. В Git отсутствуют инструменты для переписывания истории, но вы можете использовать команду rebase, чтобы перебазировать группу коммитов туда же на HEAD, где они были изначально, вместо перемещения их в другое место. С помощью интерактивного режима команды rebase, вы можете останавливаться после каждого нужного вам коммита и изменять сообщения, добавлять файлы или делать что-то другое, что вам нужно. Вы

можете запустить rebase в интерактивном режиме, добавив опцию -i к git rebase. Вы должны указать, какие коммиты вы хотите изменить, передав команде коммит, на который нужно выполнить перебазирование.

Например, если вы хотите изменить сообщения последних трёх коммитов, или сообщение какого-то одного коммита этой группы, то передайте как аргумент команде git rebase -i родителя последнего коммита, который вы хотите изменить — HEAD~2^ или HEAD~3. Может быть, проще будет запомнить ~3, так как вы хотите изменить последние три коммита; но не забывайте, что вы, в действительности, указываете четвертый коммит с конца — родителя последнего коммита, который вы хотите изменить:

```
$ git rebase -i HEAD~3
```

Напомним, что это команда перебазирования — каждый коммит, входящий в диапазон HEAD~3..HEAD, будет изменён вне зависимости от того, изменили вы сообщение или нет. Не включайте в такой диапазон коммит, который уже был отправлен на центральный сервер: сделав это, вы можете запутать других разработчиков, предоставив вторую версию одних и тех же изменений.

Выполнение этой команды отобразит в вашем текстовом редакторе список коммитов, в нашем случае, например, следующее:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
```

```
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Важно отметить, что коммиты перечислены в порядке, противоположном порядку, который вы обычно видите при использовании команды log. Если вы выполните log, то увидите следующее:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d Add cat-file
310154e Update README formatting and add blame
f7f3f6d Change my name a bit
```

Обратите внимание на обратный порядок. Команда rebase в интерактивном режиме предоставит вам скрипт, который она будет выполнять. Она начнет с коммита, который вы указали в командной строке (HEAD~3) и повторит изменения, внесённые каждым из коммитов, сверху вниз. Наверху отображается самый старый коммит, а не самый новый, потому что он будет повторен первым.

Вам необходимо изменить скрипт так, чтобы он остановился на коммите, который вы хотите изменить. Для этого измените слово pick на слово edit напротив каждого из коммитов, после которых скрипт должен остановиться. Например, для изменения сообщения только третьего коммита, измените файл следующим образом:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Когда вы сохраните сообщение и выйдете из редактора, Git переместит вас к самому раннему коммиту из списка и вернёт вас в командную строку со следующим сообщением:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... Change my name a bit
You can amend the commit now, with

    git commit --amend

Once you're satisfied with your changes, run

    git rebase --continue
```

Эти инструкции говорят вам в точности то, что нужно сделать. Выполните:

```
$ git commit --amend
```

Измените сообщение коммита и выйдите из редактора. Затем выполните:

```
$ git rebase --continue
```

Эта команда автоматически применит два оставшиеся коммита и завершится. Если вы измените pick на edit в других строках, то можете повторить эти шаги для соответствующих коммитов. Каждый раз Git будет останавливаться, позволяя вам исправить коммит, и продолжит, когда вы закончите.

Упорядочивание коммитов

Вы также можете использовать интерактивное перебазирование для изменения порядка или полного удаления коммитов. Если вы хотите удалить коммит «Add cat-file» и изменить порядок, в котором были внесены два оставшихся, то вы можете изменить скрипт перебазирования с такого:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

на такой:

```
pick 310154e Update README formatting and add blame
pick f7f3f6d Change my name a bit
```

Когда вы сохраните скрипт и выйдете из редактора, Git переместит вашу ветку на родителя этих коммитов, применит 310154e, затем f7f3f6d и после этого остановится. Вы, фактически, изменили порядок этих коммитов и полностью удалили коммит «Add cat-file».

Объединение коммитов

С помощью интерактивного режима команды rebase также можно объединить несколько коммитов в один. Git добавляет полезные инструкции в сообщение скрипта перебазирования:

```
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
```

```
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .      create a merge commit using the original merge commit's
# .      message (or the oneline, if no original merge commit was
# .      specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Если вместо «pick» или «edit» вы укажете «squash», Git применит изменения из текущего и предыдущего коммитов и предложит вам объединить их сообщения. Таким образом, если вы хотите из этих трёх коммитов сделать один, вы должны изменить скрипт следующим образом:

```
pick f7f3f6d Change my name a bit
squash 310154e Update README formatting and add blame
squash a5f4a0d Add cat-file
```

Когда вы сохраните скрипт и выйдете из редактора, Git применит изменения всех трёх коммитов и затем вернёт вас обратно в редактор, чтобы вы могли объединить сообщения коммитов:

```
# This is a combination of 3 commits.
# The first commit's message is:
Change my name a bit


# This is the 2nd commit message:
```

```
Update README formatting and add blame


# This is the 3rd commit message:


Add cat-file
```

После сохранения сообщения вы получите один коммит, содержащий изменения всех трёх коммитов, существовавших ранее.

Разбиение коммита

Разбиение коммита отменяет его и позволяет затем по частям индексировать и фиксировать изменения, создавая таким образом столько коммитов, сколько вам нужно. Например, предположим, что вы хотите разбить средний коммит на два. Вместо одного коммита «Update README formatting and add blame» вы хотите получить два разных: первый — «Update README formatting», и второй — «Add blame». Вы можете добиться этого, изменив в скрипте rebase -i инструкцию для разбиваемого коммита на «edit»:

```
pick f7f3f6d Change my name a bit
edit 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Затем, когда скрипт вернёт вас в командную строку, вам нужно будет отменить индексацию изменений этого коммита, и создать несколько коммитов на основе этих изменений. Когда вы сохраните скрипт и выйдете из редактора, Git переместится на родителя первого коммита в вашем списке, применит первый коммит (f7f3f6d), применит второй (310154e), и вернёт вас в консоль. Здесь вы можете отменить коммит с помощью команды git reset HEAD^, которая, фактически, отменит этот коммит и удалит из индекса изменённые файлы. Теперь вы можете добавлять в индекс и фиксировать файлы, пока не создадите требуемые коммиты, а после этого выполнить команду git rebase --continue:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'Update README formatting'
$ git add lib/simplegit.rb
```

```
$ git commit -m 'Add blame'
$ git rebase --continue
```

Git применит последний коммит (a5f4a0d) из скрипта, и ваша история примет следующий вид:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd Add cat-file
9b29157 Add blame
35cfb2b Update README formatting
f7f3f6d Change my name a bit
```

И снова, при этом изменились SHA-1 хеши всех коммитов в вашем списке, поэтому убедитесь, что ни один коммит из этого списка ранее не был отправлен в общий репозиторий. Обратите внимание, что последний коммит в списке (f7f3f6d) не изменился. Несмотря на то, что коммит был в списке перебазирования, он был отмечен как «pick» и применён до применения перебазирования, поэтому Git оставил его нетронутым.

Удаление коммита

Если вы хотите избавиться от какого-либо коммита, то удалить его можно во время интерактивного перебазирования rebase -i. Напишите слово «drop» перед коммитом, который хотите удалить, или просто удалите его из списка:

```
pick 461cb2a This commit is OK
drop 5aecc10 This commit is broken
```

Из-за того, как Git создаёт объекты коммитов, удаление или изменение коммита влечёт за собой перезапись всех последующих коммитов. Чем дальше вы вернётесь в историю ваших коммитов, тем больше коммитов потребуется переделать. Это может вызвать множество конфликтов слияния, особенно если у вас много последующих коммитов, которые зависят от удалённого.

Если во время подобного перебазирования вы поняли, что это была не очень хорошая идея, то всегда можно остановиться. Просто выполните команду git rebase --abort и ваш репозиторий вернётся в то состояние, в котором он был до начала перебазирования.

Если вы завершили перебазирование, а затем решили, что полученный результат — это не то, что вам нужно — воспользуйтесь командой git reflog, чтобы восстановить предыдущую версию вашей ветки.

Примечание	Дрю Дево создал практическое руководство с упражнениями по использованию git rebase. Найти его можно здесь: https://git-rebase.io/

Продвинутый инструмент: filter-branch

Существует ещё один способ переписывания истории, который вы можете использовать при необходимости изменить большое количество коммитов каким-то программируемым способом — например, изменить глобально ваш адрес электронной почты или удалить файл из всех коммитов. Для этого существует команда filter-branch, и она может изменять большие периоды вашей истории, поэтому вы, возможно, не должны её использовать кроме тех случаев, когда ваш проект ещё не стал публичным и другие люди ещё не имеют наработок, основанных на коммитах, которые вы собираетесь изменить. Однако, эта команда может быть очень полезной. Далее вы ознакомитесь с несколькими обычными вариантами использованиями этой команды, таким образом, вы сможете получить представление о том, на что она способна.

Внимание	Команда git filter-branch таит в себе много подводных камней и более не является рекомендуемым способом изменения истории. Вместо этого, рассмотрите возможность использования Python скрипта git-filter-repo, который лучше подходит для большинства ситуаций, в которых вы обычно используете filter-branch.

Удаление файла из каждого коммита

Такое случается довольно часто. Кто-нибудь случайно зафиксировал огромный бинарный файл, неосмотрительно выполнив git add ., и вы хотите отовсюду его удалить. Возможно, вы случайно зафиксировали файл, содержащий пароль, а теперь хотите сделать ваш проект общедоступным. В общем, утилиту filter-branch вы, вероятно, захотите использовать, чтобы привести к нужному виду всю вашу историю. Для удаления файла passwords.txt из всей вашей истории вы можете использовать опцию --tree-filter команды filter-branch:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD

Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
```

```
Ref 'refs/heads/master' was rewritten
```

Опция --tree-filter выполняет указанную команду после переключения на каждый коммит и затем повторно фиксирует результаты. В данном примере вы удаляете файл passwords.txt из каждого снимка вне зависимости от того, существует он или нет. Если вы хотите удалить все случайно зафиксированные резервные копии файлов, созданные текстовым редактором, то вы можете выполнить нечто подобное git filter-branch --tree-filter 'rm -f *~' HEAD.

Вы можете посмотреть, как Git изменит деревья и коммиты, а затем уже переместить указатель ветки. Как правило, хорошим подходом будет выполнение всех этих действий в тестовой ветке и, после проверки полученных результатов, установка на неё указателя основной ветки. Для выполнения filter-branch на всех ваших ветках, вы можете передать команде опцию --all.

Установка подкаталога как корневого каталога проекта

Предположим, вы выполнили импорт из другой системы контроля версий и получили в результате подкаталоги, которые не имеют никакого смысла (trunk, tags и так далее). Если вы хотите сделать подкаталог trunk корневым для каждого коммита, команда filter-branch может помочь вам в этом:

```
$ git filter-branch --subdirectory-filter trunk HEAD

Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)

Ref 'refs/heads/master' was rewritten
```

Теперь вашим новым корневым каталогом проекта будет являться подкаталог trunk. Git также автоматически удалит коммиты, которые не затрагивали этот подкаталог.

Глобальное изменение адреса электронной почты

Ещё один типичный случай возникает, когда вы забыли выполнить git config для настройки своего имени и адреса электронной почты перед началом работы, или, возможно, хотите открыть исходные коды вашего рабочего проекта и изменить везде адрес вашей рабочей электронной почты на персональный. В любом случае вы можете изменить адрес электронный почты сразу в нескольких коммитах с помощью команды filter-branch. Вы должны быть осторожны, чтобы изменить только свои адреса электронной почты, для этого используйте опцию --commit-filter:

```
$ git filter-branch --commit-filter '
```

```
if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
then
    GIT_AUTHOR_NAME="Scott Chacon";
    GIT_AUTHOR_EMAIL="schacon@example.com";
    git commit-tree "$@";
else
    git commit-tree "$@";
fi' HEAD
```

Эта команда пройдёт по всем коммитам и установит в них ваш новый адрес. Так как коммиты содержат значения SHA-1-хешей их родителей, эта команда изменяет хеш SHA-1 каждого коммита в вашей истории, а не только тех, которые соответствовали адресам электронной почты.

Раскрытие тайн reset

Перед тем, как перейти к более специализированными утилитам, давайте поговорим о reset и checkout. Эти команды кажутся самыми непонятными из всех, которые есть в Git, когда вы в первый раз сталкиваетесь с ними. Они делают так много, что попытки по-настоящему их понять и правильно использовать кажутся безнадёжными. Для того, чтобы всё же достичь этого, мы советуем воспользоваться простой аналогией.

Три дерева

Разобраться с командами reset и checkout будет проще, если считать, что Git управляет содержимым трёх различных деревьев. Здесь под «деревом» мы понимаем «набор файлов», а не специальную структуру данных. (В некоторых случаях индекс ведет себя не совсем так, как дерево, но для наших текущих целей его проще представлять именно таким.)

В своих обычных операциях Git управляет тремя деревьями:

Дерево	Назначение
HEAD	Снимок последнего коммита, родитель следующего
Индекс	Снимок следующего намеченного коммита
Рабочий Каталог	Песочница

Указатель HEAD

HEAD — это указатель на текущую ветку, которая, в свою очередь, является указателем на последний коммит, сделанный в этой ветке. Это значит, что HEAD будет родителем следующего созданного коммита. Как правило, самое простое считать HEAD снимком **вашего последнего коммита**.

На самом деле, довольно легко увидеть, что представляет из себя этот снимок. Ниже приведён пример получения содержимого каталога и контрольных сумм для каждого файла в HEAD:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
committer Scott Chacon  1301511835 -0700

initial commit
```

```
$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

Команды cat-file и ls-tree являются «служебными» (**plumbing**) командами, которые используются внутри системы и не требуются при ежедневной работе, но они помогают нам разобраться, что же происходит на самом деле.

Индекс

Индекс — это ваш **следующий намеченный коммит**. Мы также упоминали это понятие как «область подготовленных изменений» Git — то, что Git просматривает, когда вы выполняете git commit.

Git заполняет индекс списком изначального содержимого всех файлов, выгруженных в последний раз в ваш рабочий каталог. Затем вы заменяете некоторые из таких файлов их новыми версиями, и команда git commit преобразует изменения в дерево для нового коммита.

```
$ git ls-files -s
```

```
100644 a906cb2a4a904a152e80877d4088654daad0c859 0
README

100644 8f94139338f9404f26296befa88755fc2598c289 0Rakefile

100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0
lib/simplegit.rb
```

Повторим, здесь мы используем служебную команду ls-files, которая показывает вам, как выглядит сейчас ваш индекс.

Технически, индекс не является древовидной структурой, на самом деле, он реализован как сжатый список (**flattened manifest**) — но для наших целей такого представления будет достаточно.

Рабочий Каталог

Наконец, у вас есть рабочий каталог. Два других дерева сохраняют свое содержимое эффективным, но неудобным способом внутри каталога .git. Рабочий Каталог распаковывает их в настоящие файлы, что упрощает для вас их редактирование. Считайте Рабочий Каталог **песочницей**, где вы можете опробовать изменения перед их коммитом в индекс (область подготовленных изменений) и затем в историю.

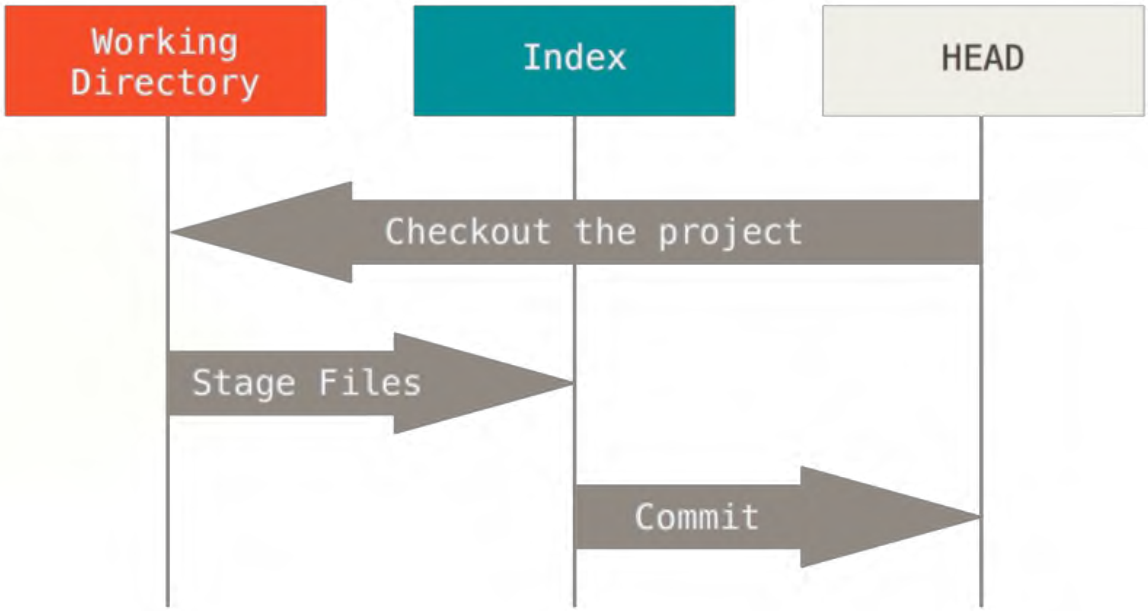
```
$ tree

.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

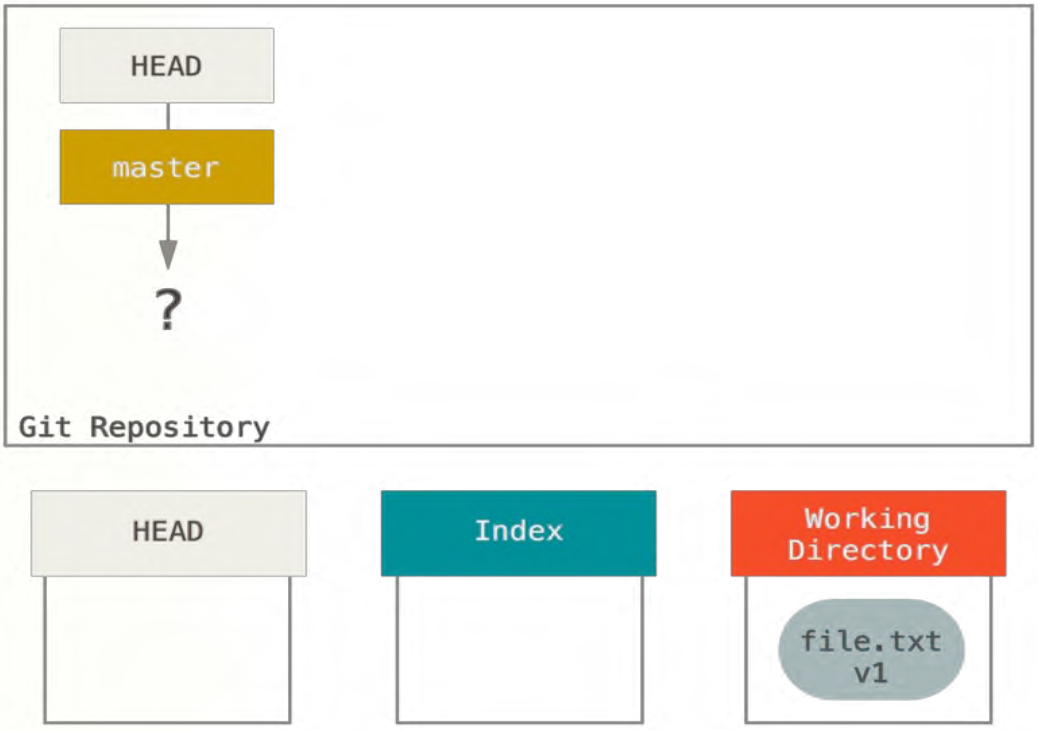

1 directory, 3 files
```

Технологический процесс

Основное предназначение Git — это сохранение снимков последовательно улучшающихся состояний вашего проекта, путём управления этими тремя деревьями.

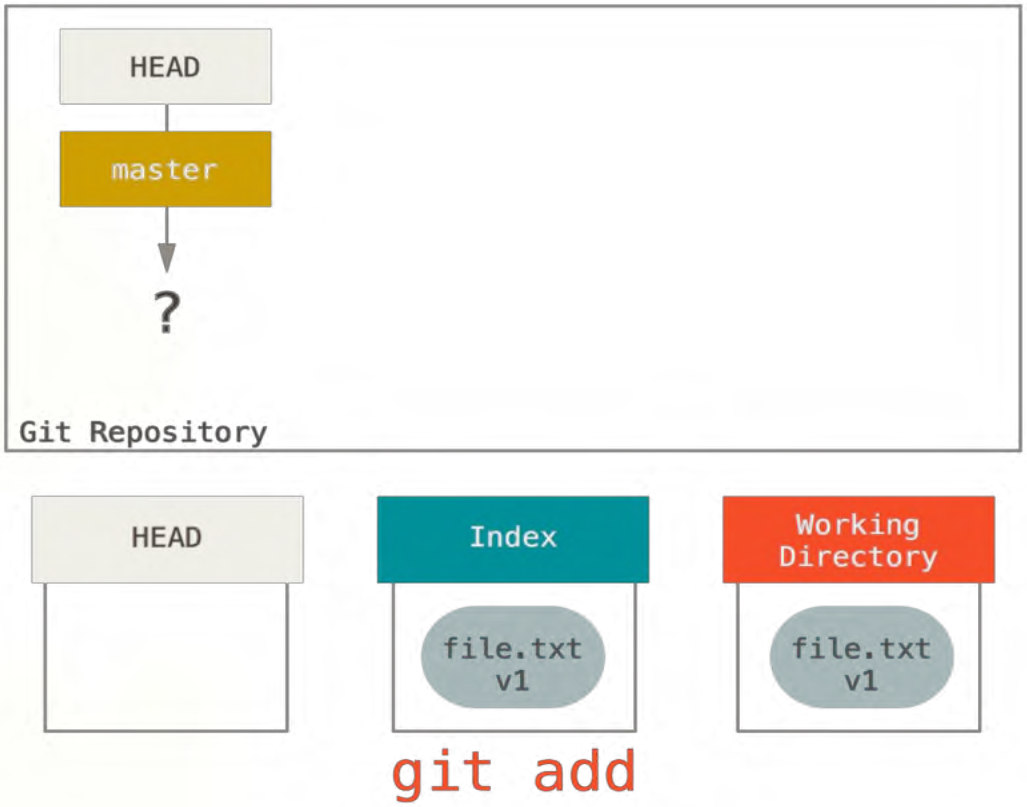


Давайте рассмотрим этот процесс: пусть вы перешли в новый каталог, содержащий один файл. Данную версию этого файла будем называть **v1** и изображать голубым цветом. Выполним команду git init, которая создаст Git-репозиторий, у которого ссылка HEAD будет указывать на ещё несуществующую ветку (master пока не существует).

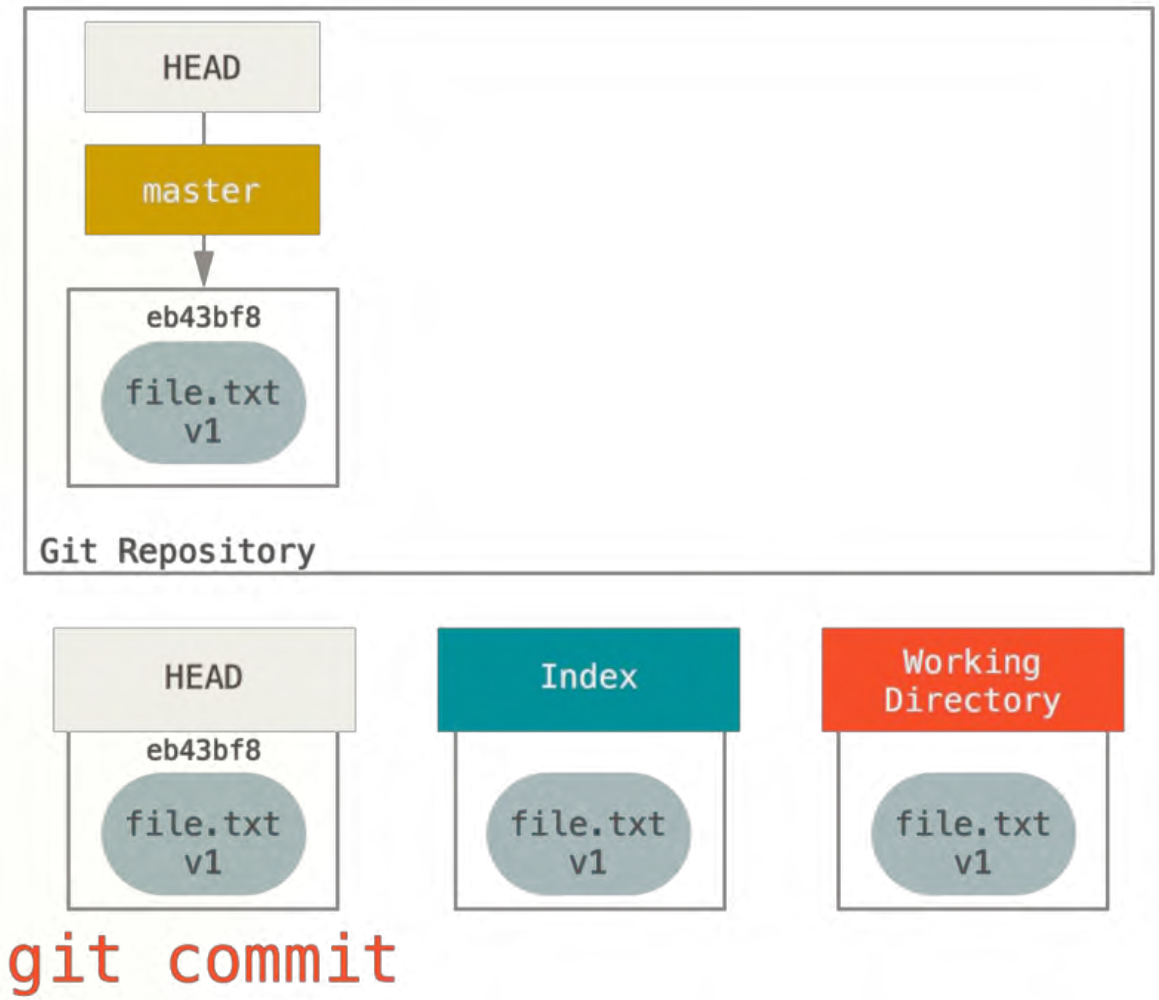


На данном этапе только дерево Рабочего Каталога содержит данные.

Теперь мы хотим закоммитить этот файл, поэтому мы используем git add для копирования содержимого Рабочего Каталога в Индекс.

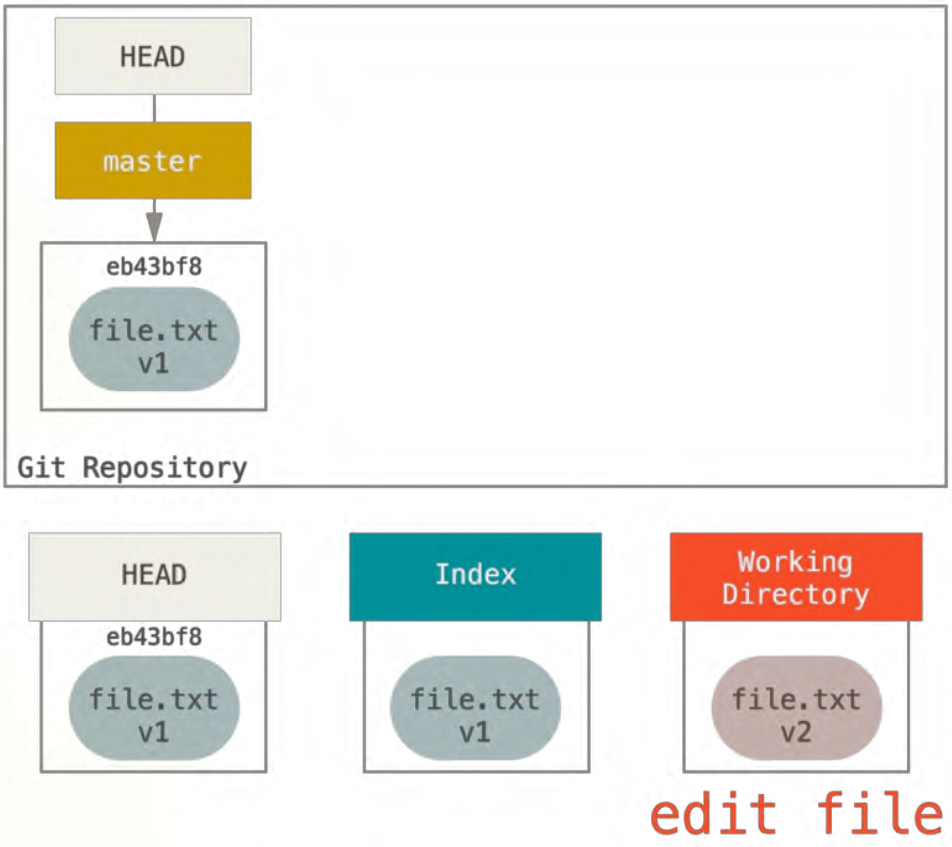


Затем, мы выполняем команду git commit, которая сохраняет содержимое Индекса как неизменяемый снимок, создает объект коммита, который указывает на этот снимок, и обновляет master так, чтобы он тоже указывал на этот коммит.

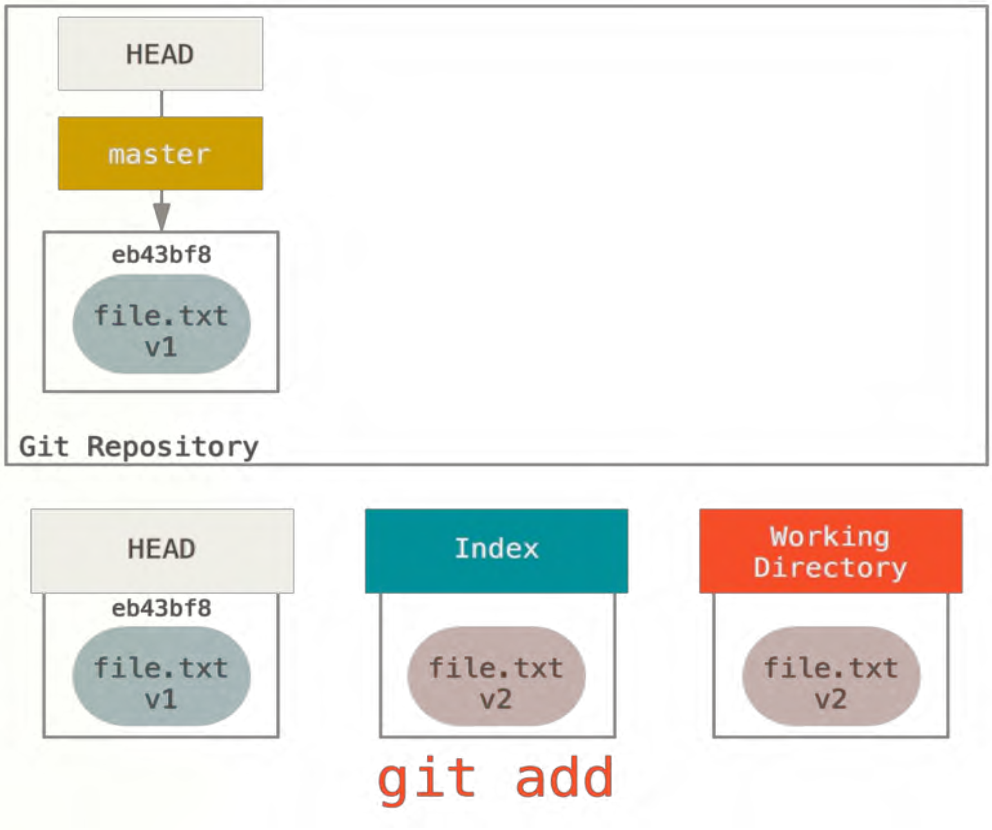


Если сейчас выполнить git status, то мы не увидим никаких изменений, так как все три дерева одинаковые.

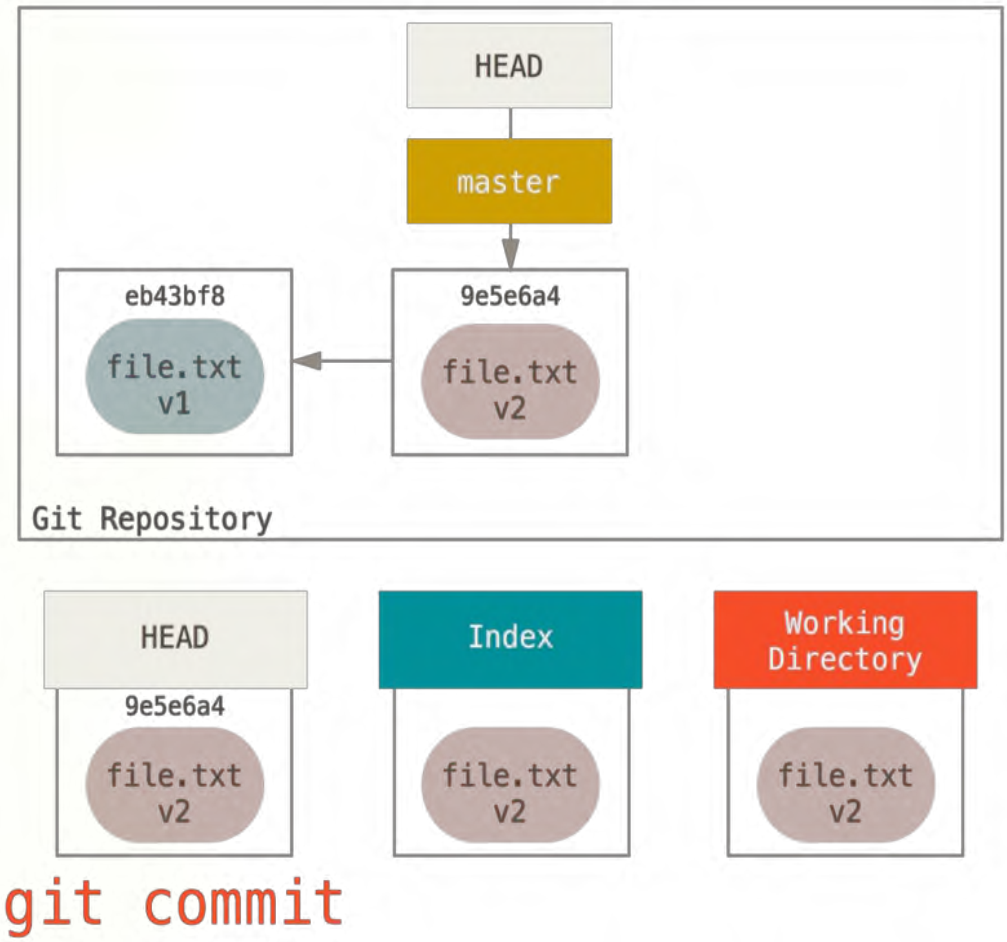
Теперь мы хотим внести изменения в файл и закоммитить его. Мы пройдём через всё ту же процедуру; сначала мы отредактируем файл в нашем рабочем каталоге. Давайте называть эту версию файла **v2** и обозначать красным цветом.



Если сейчас мы выполним git status, то увидим, что файл выделен красным в разделе «Изменения, не подготовленные к коммиту», так как его представления в Индексе и Рабочем Каталоге различны. Затем мы выполним git add для этого файла, чтобы поместить его в Индекс.



Если сейчас мы выполним git status, то увидим, что этот файл выделен зелёным цветом в разделе «Изменения, которые будут закоммичены», так как Индекс и HEAD различны — то есть, наш следующий намеченный коммит сейчас отличается от нашего последнего коммита. Наконец, мы выполним git commit, чтобы окончательно совершить коммит.



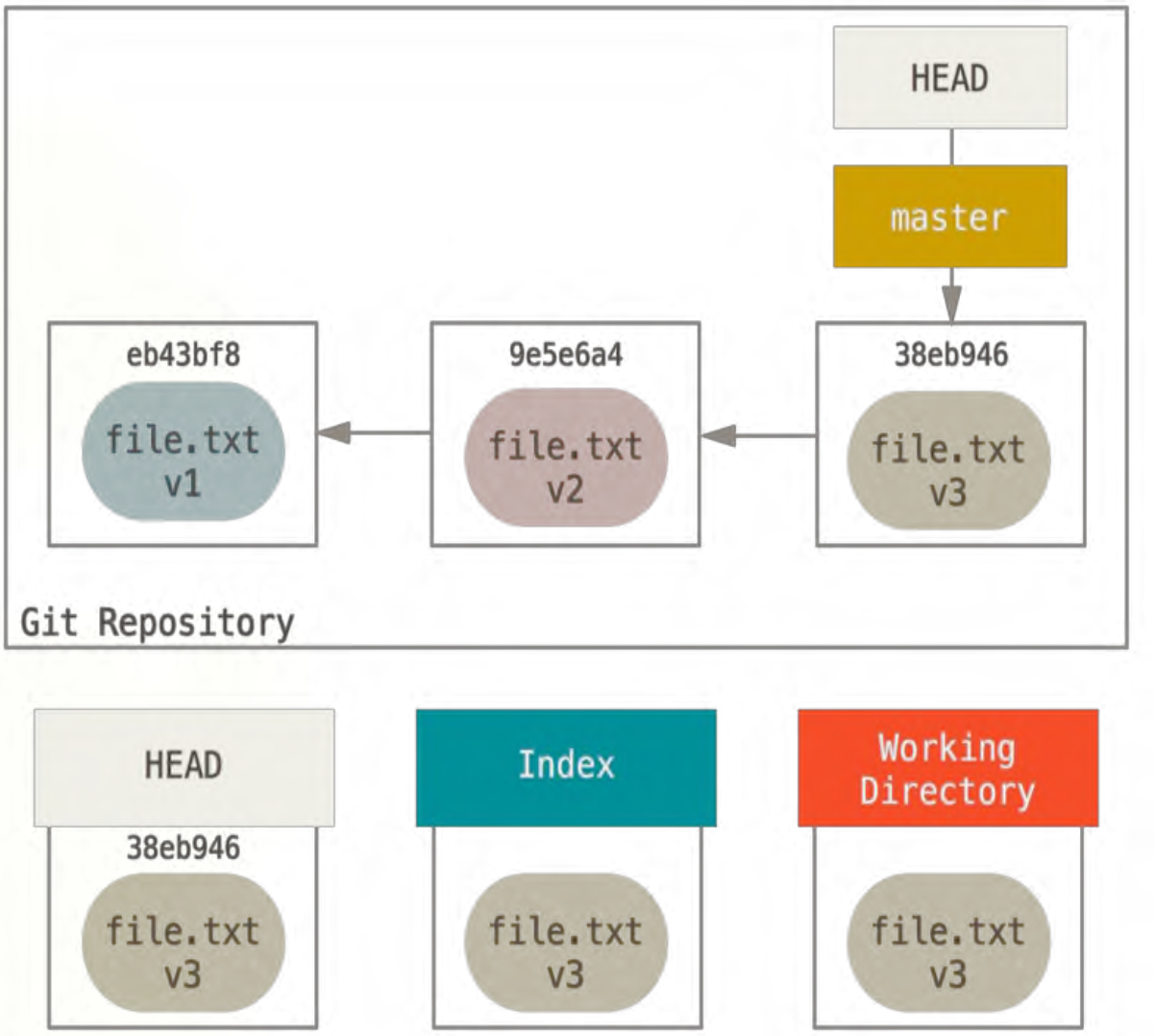
Сейчас команда git status не показывает ничего, так как снова все три дерева одинаковые.

Переключение веток и клонирование проходят через похожий процесс. Когда вы переключаетесь (checkout) на ветку, **HEAD** начинает также указывать на новую ветку, ваш **Индекс** замещается снимком коммита этой ветки, и затем содержимое **Индекса** копируется в ваш **Рабочий Каталог**.

Назначение команды reset

Команда reset становится более понятной, если рассмотреть её с учётом вышеизложенного.

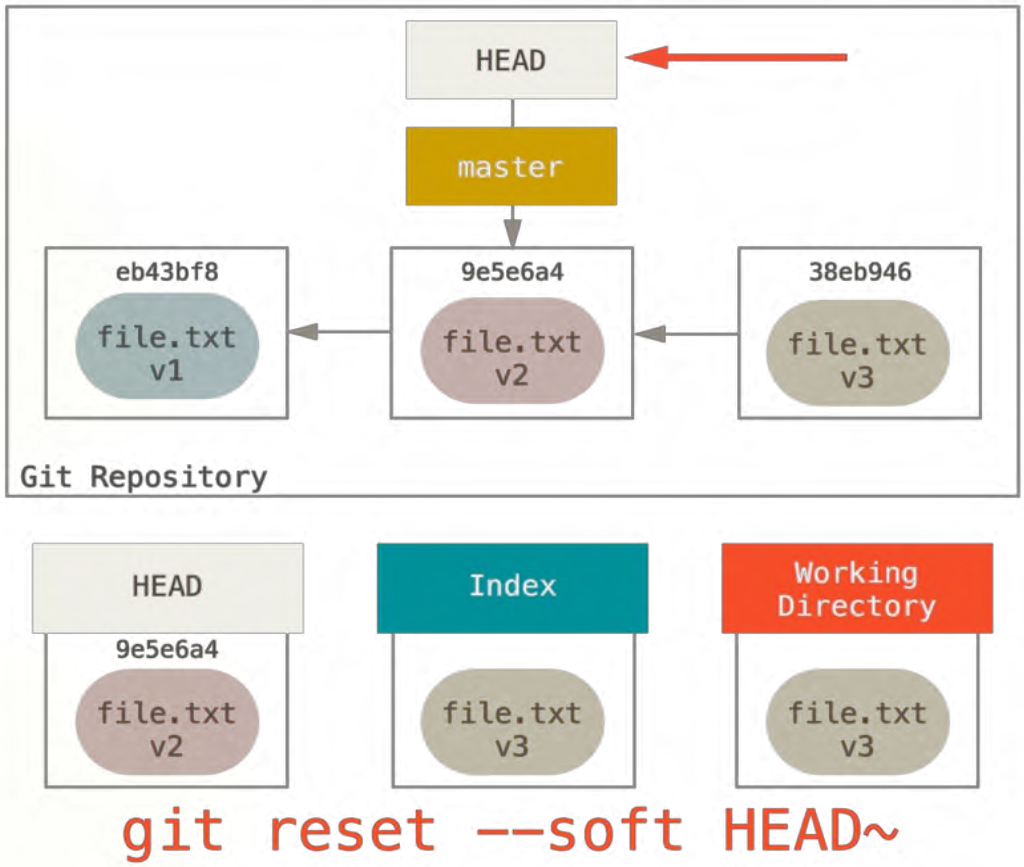
В следующих примерах предположим, что мы снова изменили файл file.txt и закоммитили его в третий раз. Так что наша история теперь выглядит так:



Давайте теперь внимательно проследим, что именно происходит при вызове reset. Эта команда простым и предсказуемым способом управляет тремя деревьями, существующими в Git. Она выполняет три основных операции.

Шаг 1: Перемещение указателя HEAD

Первое, что сделает reset — переместит то, на что указывает HEAD. Обратите внимание, изменяется не сам HEAD (что происходит при выполнении команды checkout); reset перемещает ветку, на которую указывает HEAD. Таким образом, если HEAD указывает на ветку master (то есть вы сейчас работаете с веткой master), выполнение команды git reset 9e5e6a4 сделает так, что master будет указывать на 9e5e6a4.



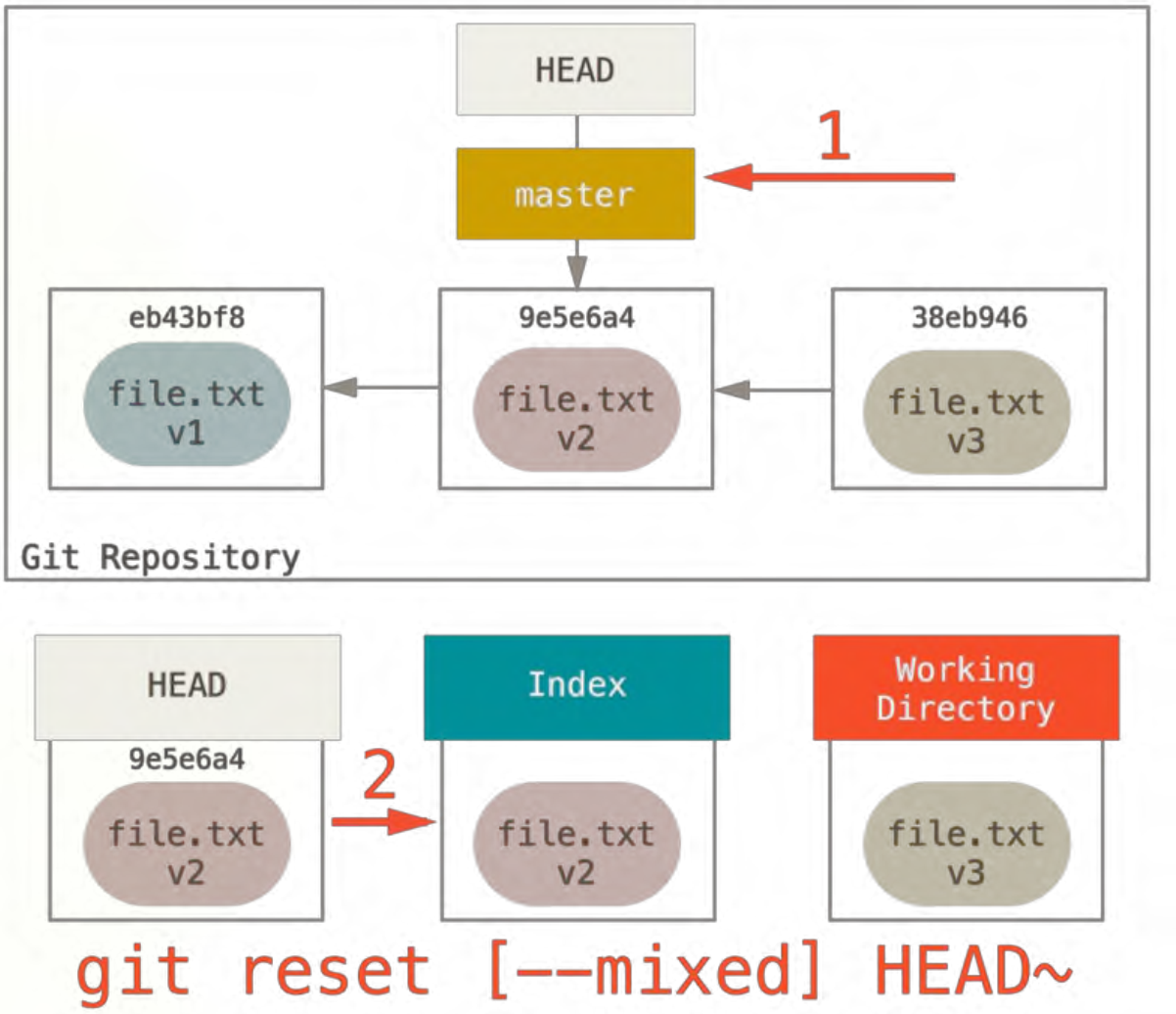
Не важно с какими опциями вы вызвали команду reset **с указанием коммита** (reset также можно вызывать с указанием пути), она всегда будет пытаться сперва сделать данный шаг. При вызове reset --soft на этом выполнение команды и остановится.

Теперь взгляните на диаграмму и постарайтесь разобраться, что случилось: фактически была отменена последняя команда git commit. Когда вы выполняете git commit, Git создает новый коммит и перемещает на него ветку, на которую указывает HEAD. Если вы выполняете reset на HEAD~ (родителя HEAD), то вы перемещаете ветку туда, где она была раньше, не изменяя при этом ни Индекс, ни Рабочий Каталог. Вы можете обновить Индекс и снова выполнить git commit, таким образом добиваясь того же, что делает команда git commit --amend (смотрите Изменение последнего коммита).

Шаг 2: Обновление Индекса (--mixed)

Заметьте, если сейчас вы выполните git status, то увидите отмеченные зелёным цветом изменения между Индексом и новым HEAD.

Следующим, что сделает reset, будет обновление Индекса содержимым того снимка, на который указывает HEAD.

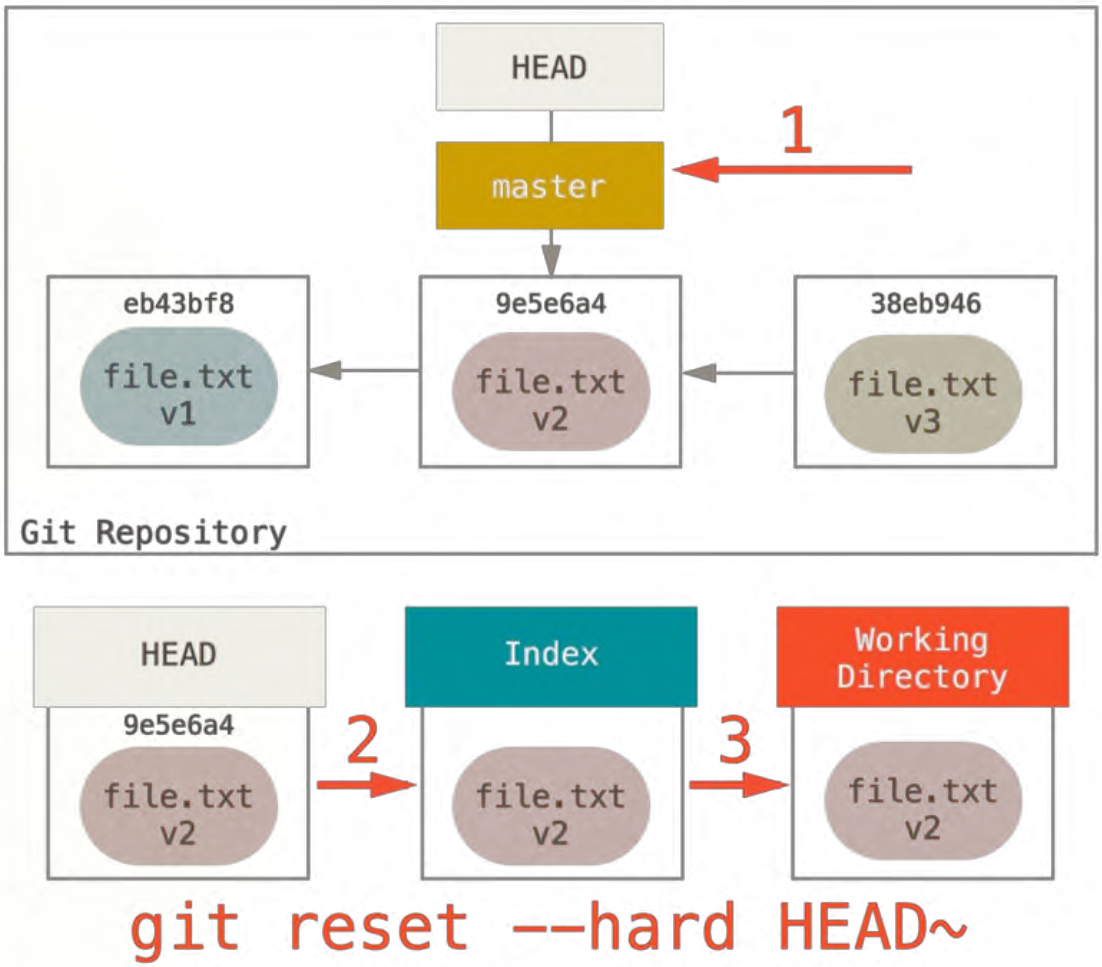


Если вы указали опцию --mixed, выполнение reset остановится на этом шаге. Такое поведение также используется по умолчанию, поэтому если вы не указали совсем никаких опций (в нашем случае git reset HEAD~), выполнение команды также остановится на этом шаге.

Снова взгляните на диаграмму и постарайтесь разобраться, что произошло: отменен не только ваш последний commit, но также и **добавление в индекс** всех файлов. Вы откатились назад до момента выполнения команд git add и git commit.

Шаг 3: Обновление Рабочего Каталога (--hard)

Третье, что сделает reset — это приведение вашего Рабочего Каталога к тому же виду, что и Индекс. Если вы используете опцию --hard, то выполнение команды будет продолжено до этого шага.



Давайте разберемся, что сейчас случилось. Вы отменили ваш последний коммит, результаты выполнения команд git add и git commit, а также **все** изменения, которые вы сделали в рабочем каталоге.

Важно отметить, что только указание этого флага (--hard) делает команду reset опасной, это один из немногих случаев, когда Git действительно удаляет данные. Все остальные вызовы reset легко отменить, но при указании опции --hard команда принудительно перезаписывает файлы в Рабочем Каталоге. В данном конкретном случае версия **v3** нашего файла всё ещё остаётся в коммите внутри базы данных Git и мы можем вернуть её, просматривая наш reflog, но если вы не коммитили эту версию, Git перезапишет файл и её уже нельзя будет восстановить.

Резюме

Команда reset в заранее определённом порядке перезаписывает три дерева Git, останавливаясь тогда, когда вы ей скажете:

1. Перемещает ветку, на которую указывает HEAD **(останавливается на этом, если указана опция --soft)**
2. Делает Индекс таким же, как и HEAD **(останавливается на этом, если не указана опция --hard)**

3. Делает Рабочий Каталог таким же, как и Индекс.

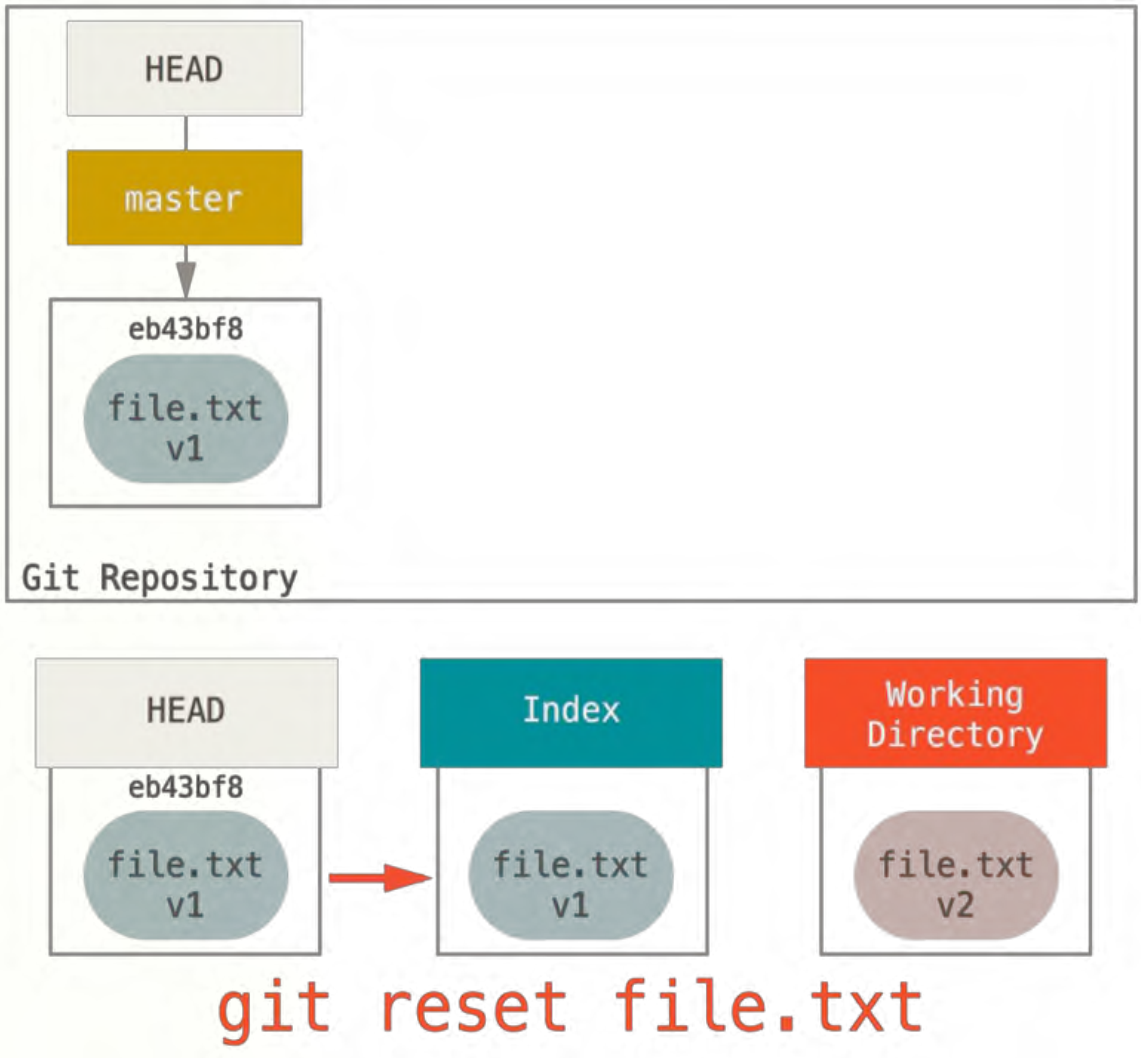
Reset с указанием пути

Основной форме команды reset (без опций --soft и --hard) вы также можете передавать путь, с которым она будет оперировать. В этом случае reset пропустит первый шаг, а на остальных будет работать только с указанным файлом или набором файлов. Первый шаг пропускается, так как HEAD является указателем и не может ссылаться частично на один коммит, а частично на другой. Но Индекс и Рабочий Каталог **могут** быть изменены частично, поэтому reset выполняет шаги 2 и 3.

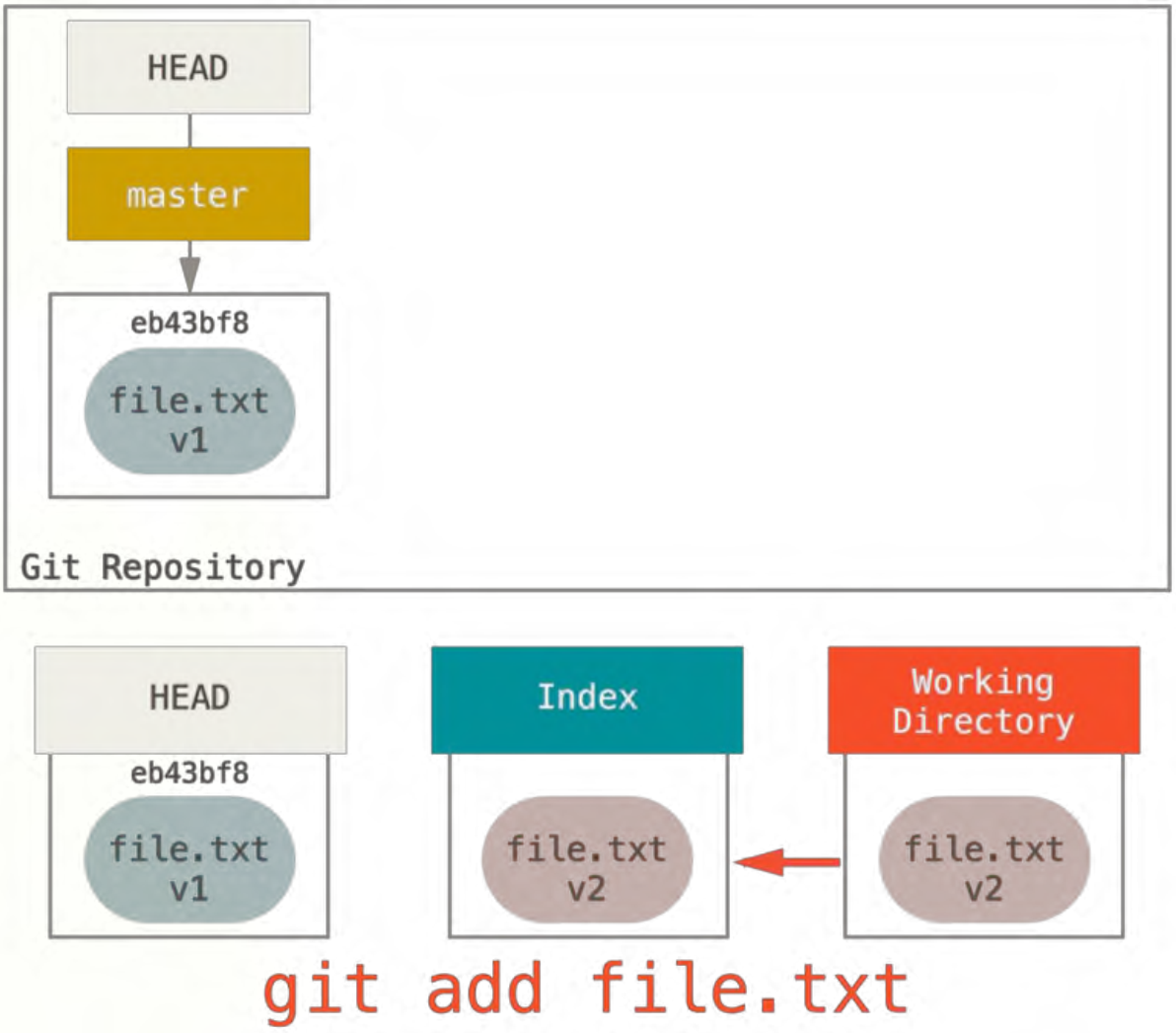
Итак, предположим вы выполнили команду git reset file.txt. Эта форма записи (так как вы не указали ни SHA-1 коммита, ни ветку, ни опций --soft или --hard) является сокращением для git reset --mixed HEAD file.txt, которая:

1. Перемещает ветку, на которую указывает HEAD **(будет пропущено)**
2. Делает Индекс таким же, как и HEAD **(остановится здесь)**

То есть, фактически, она копирует файл file.txt из HEAD в Индекс.

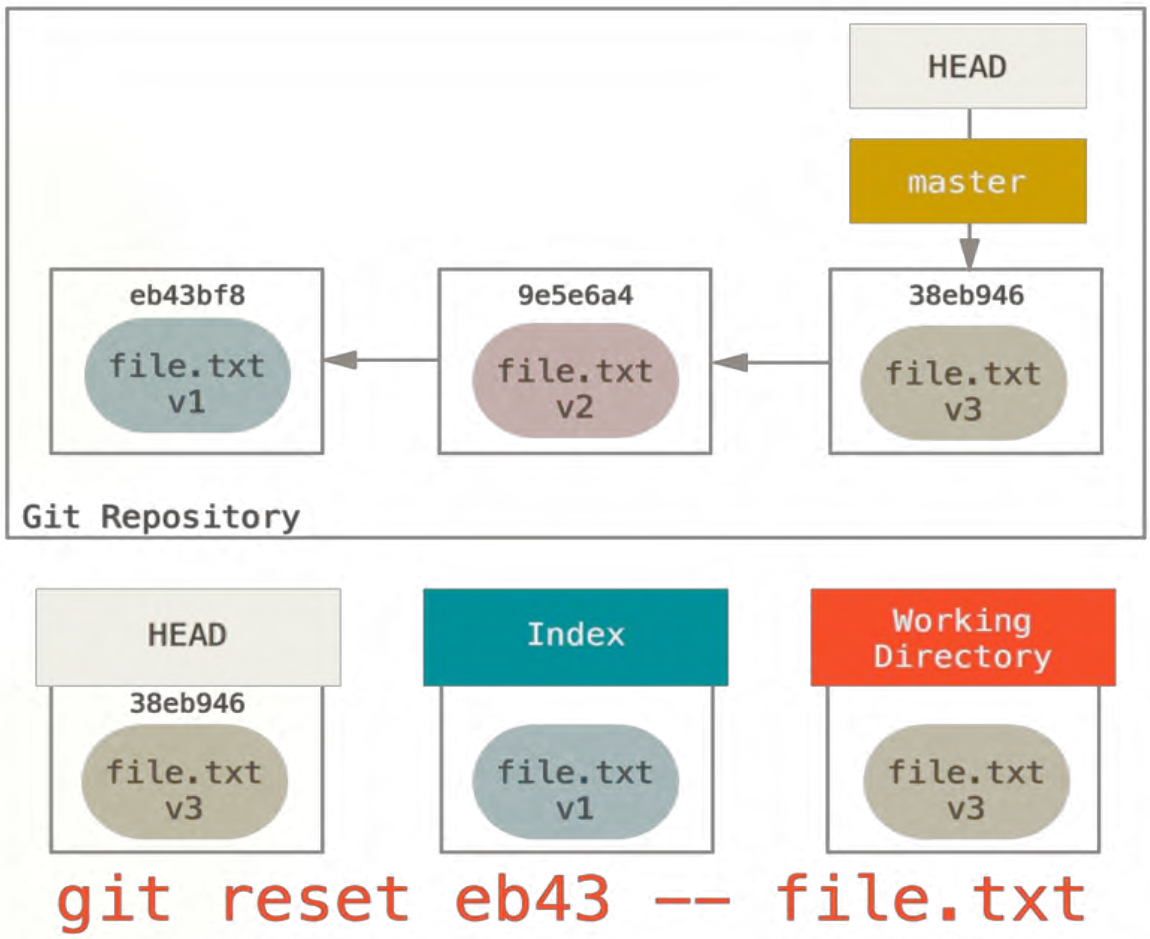


Это создает эффект **отмены индексации** файла. Если вы посмотрите на диаграммы этой команды и команды git add, то увидите, что их действия прямо противоположные.



Именно поэтому в выводе git status предлагается использовать такую команду для отмены индексации файла. (Смотрите подробности в Отмена индексации файла.)

Мы легко можем заставить Git «брать данные не из HEAD», указав коммит, из которого нужно взять версию этого файла. Для этого мы должны выполнить следующее git reset eb43bf file.txt.



Можно считать, что, фактически, мы в Рабочем Каталоге вернули содержимое файла к версии **v1**, выполнили для него git add, а затем вернули содержимое обратно к версии **v3** (в действительности все эти шаги не выполняются). Если сейчас мы выполним git commit, то будут сохранены изменения, которые возвращают файл к версии **v1**, но при этом файл в Рабочем Каталоге никогда не возвращался к такой версии.

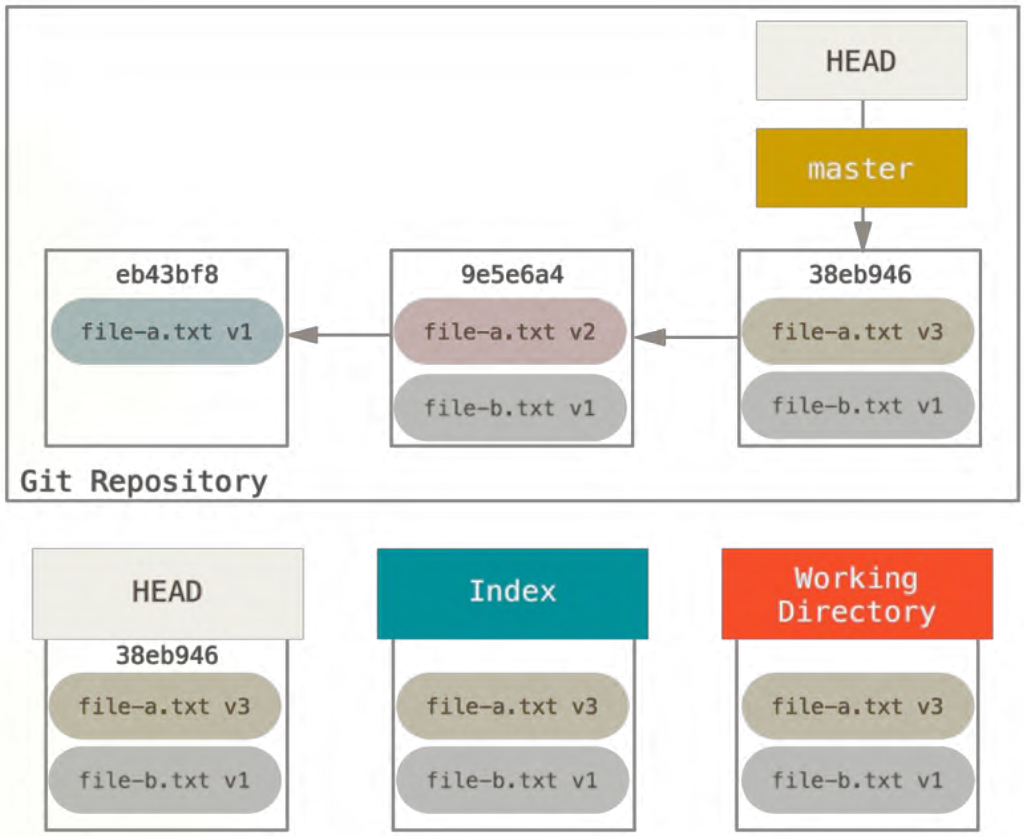
Заметим, что, как и команде git add, reset можно указывать опцию --patch для отмены индексации части содержимого. Таким способом вы можете избирательно отменять индексацию или откатывать изменения.

Слияние коммитов

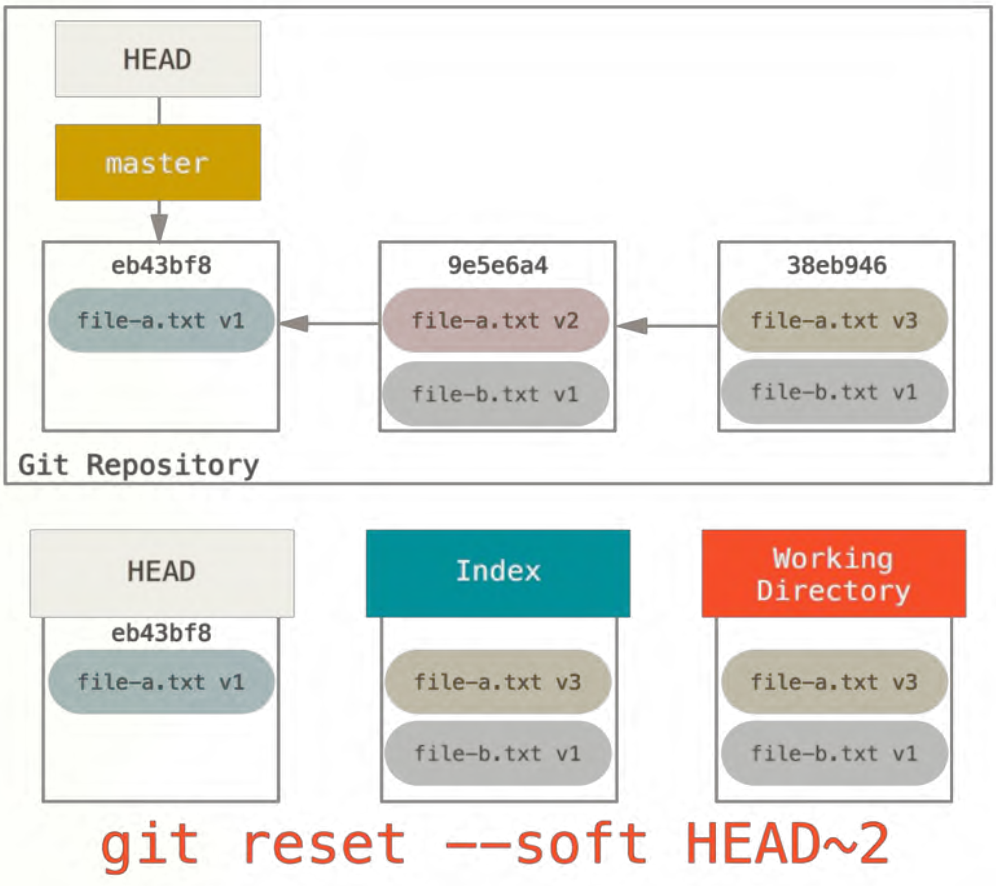
Давайте посмотрим, как, используя вышеизложенное, сделать кое-что интересное — слияние коммитов.

Допустим, у вас есть последовательность коммитов с сообщениями вида «упс.», «В работе» и «позабыл этот файл». Вы можете использовать reset для того, чтобы просто и быстро слить их в один.

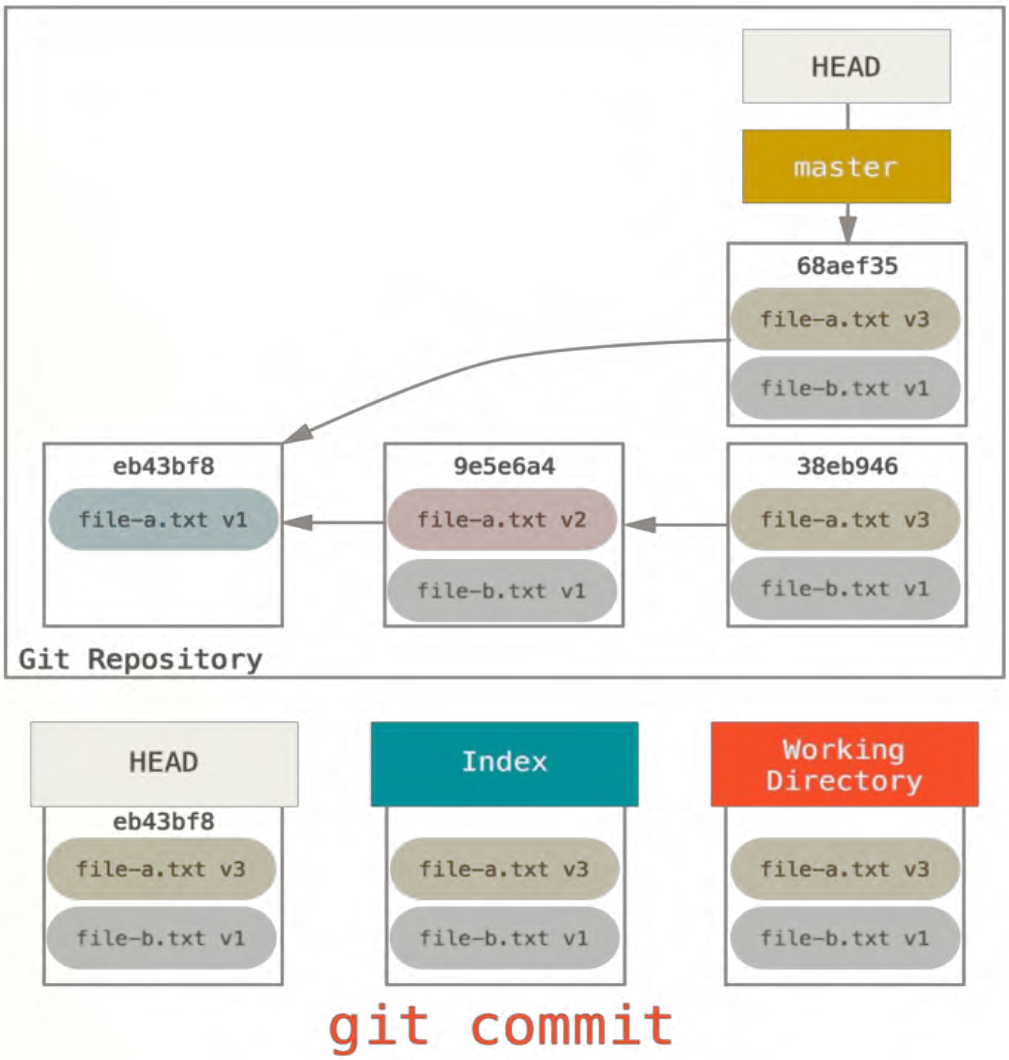
Предположим, у вас есть проект, в котором первый коммит содержит один файл, второй коммит добавляет новый файл и изменяет первый, а третий коммит снова изменяет первый файл. Второй коммит был сделан в процессе работы, и вы хотите слить его со следующим.



Вы можете выполнить git reset --soft HEAD~2, чтобы вернуть ветку HEAD на какой-то из предыдущих коммитов (на первый коммит, который вы хотите оставить):



Затем просто снова выполните git commit:



Теперь вы можете видеть, что ваша «достижимая» история (история, которую вы впоследствии отправите на сервер), сейчас выглядит так — у вас есть первый коммит с файлом file-a.txt версии **v1**, и второй, который изменяет файл file-a.txt до версии **v3** и добавляет file-b.txt. Коммита, который содержал файл версии **v2** не осталось в истории.

Сравнение с checkout

Наконец, вы можете задаться вопросом, в чем же состоит отличие между checkout и reset. Как и reset, команда checkout управляет тремя деревьями Git, и также её поведение зависит от того указали ли вы путь до файла или нет.

Без указания пути

Команда git checkout [branch] очень похожа на git reset --hard [branch], в процессе их выполнения все три дерева изменяются так, чтобы выглядеть как [branch]. Но между этими командами есть два важных отличия.

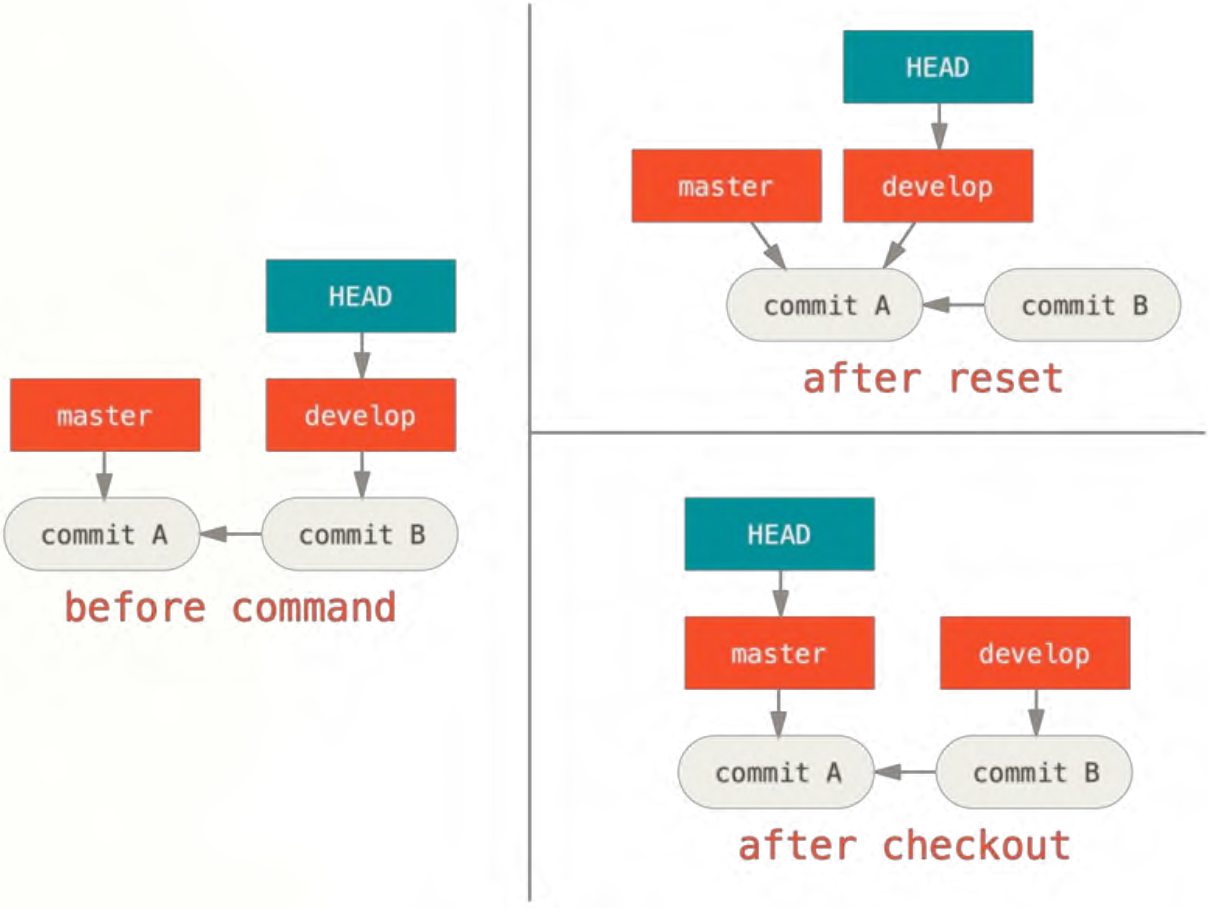
Во-первых, в отличие от reset --hard, команда checkout бережно относится к рабочему каталогу, и проверяет, что она не трогает файлы, в

которых есть изменения. В действительности эта команда поступает немного умнее — она пытается выполнить в Рабочем Каталоге простые слияния так, чтобы все файлы, которые вы **не** изменяли, были обновлены. С другой стороны, команда reset --hard просто заменяет всё целиком, не выполняя проверок.

Второе важное отличие заключается в том, как эти команды обновляют HEAD. В то время как reset перемещает ветку, на которую указывает HEAD, команда checkout перемещает сам HEAD так, чтобы он указывал на другую ветку.

Например, пусть у нас есть ветки master и develop, которые указывают на разные коммиты, и мы сейчас находимся на ветке develop (то есть HEAD указывает на неё). Если мы выполним git reset master, сама ветка develop станет ссылаться на тот же коммит, что и master. Если мы выполним git checkout master, то develop не изменится, но изменится HEAD. Он станет указывать на master.

Итак, в обоих случаях мы перемещаем HEAD на коммит A, но важное отличие состоит в том, **как** мы это делаем. Команда reset переместит также и ветку, на которую указывает HEAD, а checkout перемещает только сам HEAD.



С указанием пути

Другой способ выполнить checkout состоит в том, чтобы указать путь до файла. В этом случае, как и для команды reset, HEAD не перемещается. Эта команда, как и git reset [branch] file обновляет файл в индексе версией из коммита, но дополнительно она обновляет и файл в рабочем каталоге. То же самое сделала бы команда git reset --hard [branch] file (если бы reset можно было бы так запускать) — это небезопасно для рабочего каталога и не перемещает HEAD.

Так же как git reset и git add, команда checkout принимает опцию --patch для того, чтобы позволить вам избирательно откатить изменения содержимого файла по частям.

Заключение

Надеюсь, вы разобрались с командой reset и можете её спокойно использовать. Но, возможно, вы всё ещё немного путаетесь, чем именно она отличается от checkout, и не запомнили всех правил, используемых в различных вариантах вызова.

Ниже приведена памятка того, как эти команды воздействуют на каждое из деревьев. В столбце «HEAD» указывается «REF» если эта команда перемещает ссылку (ветку), на которую HEAD указывает, и «HEAD» если перемещается только сам HEAD. Обратите особое внимание на столбец «Сохранность РК?» — если в нем указано **NO**, то хорошенько подумайте прежде, чем выполнить эту команду.

	HEAD	**Индекс**	**Рабочий Каталог**	**Сохранность рабочего каталога**
На уровне коммитов				
reset --soft [коммит]	REF	НЕТ	НЕТ	ДА
reset [коммит]	REF	ДА	НЕТ	ДА
reset --hard [коммит]	REF	ДА	ДА	**НЕТ**
checkout [коммит]	HEAD	ДА	ДА	ДА

	HEAD	Индекс	Рабочий Каталог	Сохранность рабочего каталога
На уровне файлов				
reset (коммит) [путь]	НЕТ	ДА	НЕТ	ДА
checkout (коммит) [путь]	НЕТ	ДА	ДА	**НЕТ**

Продвинутое слияние

Обычно выполнять слияния в Git довольно легко. Git упрощает повторные слияния с одной и той же веткой, таким образом, позволяя вам иметь очень долго живущую ветку, и вы можете сохранять ее всё это время в актуальном состоянии, часто разрешая маленькие конфликты, а не доводить дело до одного большого конфликта по завершению всех изменений.

Однако, иногда всё же будут возникать сложные конфликты. В отличие от других систем управления версиями, Git не пытается быть слишком умным при разрешении конфликтов слияния. Философия Git заключается в том, чтобы быть умным, когда слияние разрешается однозначно, но, если возникает конфликт, он не пытается сумничать и разрешить его автоматически. Поэтому, если вы слишком долго откладываете слияние двух быстрорастущих веток, вы можете столкнуться с некоторыми проблемами.

В этом разделе мы рассмотрим некоторые из возможных проблем и инструменты, которые предоставляет Git, чтобы помочь вам справиться с этими более сложными ситуациями. Мы также рассмотрим некоторые другие нестандартные типы слияний, которые вы можете выполнять, и вы узнаете, как можно откатить уже выполненные слияния.

Конфликты слияния

Для работы с более сложными конфликтами Git предоставляет несколько инструментов, которые помогут вам понять, что произошло и как лучше обойтись с конфликтом.

Во-первых, если есть возможность, перед слиянием, в котором может возникнуть конфликт, позаботьтесь о том, чтобы ваша рабочая копия была без локальных изменений. Если у вас есть несохраненные наработки, либо припрячьте их, либо сохраните их во временной ветке. Таким образом,

вы сможете легко отменить **любые** изменения, которые сделаете в рабочем каталоге. Если при выполнении слияния вы не сохраните сделанные изменения, то некоторые из описанных ниже приёмов могут привести к утрате этих наработок.

Давайте рассмотрим очень простой пример. Допустим, у нас есть файл с исходниками на Ruby, выводящими на экран строку 'hello world'.

```
#! /usr/bin/env ruby


def hello
  puts 'hello world'
end


hello()
```

В нашем репозитории мы создадим новую ветку по имени whitespace и выполним замену всех окончаний строк в стиле Unix на окончания строк в стиле DOS. Фактически, изменения будут внесены в каждую строку, но изменятся только пробельные символы. Затем мы заменим строку «hello world» на «hello mundo».

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'
```

```
$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'Convert hello.rb to DOS'
[whitespace 3270f76] Convert hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)
```

```
$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
```

```
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()
```

```
$ git commit -am 'Use Spanish instead of English'
[whitespace 6d338d2] Use Spanish instead of English
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Теперь мы переключимся обратно на ветку master и добавим к функции некоторую документацию.

```
$ git checkout master
Switched to branch 'master'
```

```
$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
```

```
#! /usr/bin/env ruby


+# prints out a greeting
def hello
  puts 'hello world'
end
```

```
$ git commit -am 'Add comment documenting the function'
[master bec6336] Add comment documenting the function
 1 file changed, 1 insertion(+)
```

Теперь мы попытаемся слить в текущую ветку whitespace и в результате получим конфликты, так как изменились пробельные символы.

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Прерывание слияния

В данный момент у нас есть несколько вариантов дальнейших действий. Во-первых, давайте рассмотрим, как выйти из этой ситуации. Если вы, возможно, не были готовы к конфликтам и на самом деле не хотите связываться с ними, вы можете просто отменить попытку слияния, используя команду git merge --abort.

```
$ git status -sb
## master
UU hello.rb
```

```
$ git merge --abort
```

```
$ git status -sb
```

```
## master
```

Эта команда пытается откатить ваше состояние до того, что было до запуска слияния. Завершиться неудачно она может только в случаях, если перед запуском слияния у вас были не припрятанные или не зафиксированные изменения в рабочем каталоге, во всех остальных случаях всё будет хорошо.

Если по каким-то причинам вы обнаружили себя в ужасном состоянии и хотите просто начать всё сначала, вы можете также выполнить `git reset --hard HEAD` (либо вместо HEAD указав то, куда вы хотите откатиться). Но помните, что это откатит все изменения в рабочем каталоге, поэтому удостоверьтесь, что никакие из них вам не нужны.

Игнорирование пробельных символов

В данном конкретном случае конфликты связаны с пробельными символами. Мы знаем это, так как это простой пример, но в реальных ситуациях это также легко определить при изучении конфликта, так как каждая строка в нем будет удалена и добавлена снова. По умолчанию Git считает все эти строки изменёнными и поэтому не может слить файлы.

Стратегии слияния, используемой по умолчанию, можно передать аргументы, и некоторые из них предназначены для соответствующей настройки игнорирования изменений пробельных символов. Если вы видите, что множество конфликтов слияния вызваны пробельными символами, то вы можете прервать слияние и запустить его снова, но на этот раз с опцией `-Xignore-all-space` или `-Xignore-space-change`. Первая опция игнорирует изменения в любом **количестве** существующих пробельных символов, вторая игнорирует вообще все изменения пробельных символов.

```
$ git merge -Xignore-space-change whitespace

Auto-merging hello.rb

Merge made by the 'recursive' strategy.

 hello.rb | 2 +-

 1 file changed, 1 insertion(+), 1 deletion(-)
```

Поскольку в этом примере реальные изменения файлов не конфликтуют, то при игнорировании изменений пробельных символов всё сольётся хорошо.

Это значительно облегчает жизнь, если кто-то в вашей команде любит временами заменять все пробелы на табуляции или наоборот.

Ручное слияние файлов

Хотя Git довольно хорошо обрабатывает пробельные символы, с другими типами изменений он не может справиться автоматически, но существуют другие варианты исправления. Например, представим, что Git не умеет обрабатывать изменения пробельных символов и нам нужно сделать это вручную.

То, что нам действительно нужно — это перед выполнением самого слияния прогнать сливаемый файл через программу dos2unix. Как мы будем делать это?

Во-первых, мы перейдём в состояние конфликта слияния. Затем нам необходимо получить копии **нашей** версии файла, **их** версии файла (из ветки, которую мы сливаем) и **общей** версии (от которой ответвились первые две). Затем мы исправим либо их версию, либо нашу и повторим слияние только для этого файла.

Получить эти три версии файла, на самом деле, довольно легко. Git хранит все эти версии в индексе в разных «состояниях», каждое из которых имеет ассоциированный с ним номер. Состояние 1 — это общий предок, состояние 2 — ваша версия и состояния 3 взято из MERGE_HEAD — версия, которую вы сливаете («их» версия).

Вы можете извлечь копию каждой из этих версий конфликтующего файла с помощью команды git show и специального синтаксиса.

```
$ git show :1:hello.rb > hello.common.rb

$ git show :2:hello.rb > hello.ours.rb

$ git show :3:hello.rb > hello.theirs.rb
```

Если вы хотите что-то более суровое, то можете также воспользоваться служебной командой ls-files -u для получения SHA-1 хешей для каждого из этих файлов.

```
$ git ls-files -u

100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1
hello.rb

100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2hello.rb

100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3
hello.rb
```

Выражение :1:hello.rb является просто сокращением для поиска такого SHA-1 хеша.

Теперь, когда в нашем рабочем каталоге присутствует содержимое всех трёх состояний, мы можем вручную исправить их, чтобы устранить проблемы с пробельными символами и повторно выполнить слияние с помощью малоизвестной команды git merge-file, которая делает именно это.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...
```

```
$ git merge-file -p \
   hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb
```

```
$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

Теперь у нас есть корректно слитый файл. На самом деле, данный способ лучше, чем использование опции ignore-all-space, так как в его

рамках вместо игнорирования изменений пробельных символов перед слиянием выполняется корректное исправление таких изменений. При слиянии с ignore-all-space мы в результате получим несколько строк с окончаниями в стиле DOS, то есть в одном файле смешаются разные стили окончания строк.

Если перед коммитом изменений вы хотите посмотреть какие в действительности были различия между состояниями, то можете воспользоваться командой git diff, сравнивающей содержимое вашего рабочего каталога, которое будет зафиксировано как результат слияния, с любым из трёх состояний. Давайте посмотрим на все эти сравнения.

Чтобы сравнить результат слияния с тем, что было в вашей ветке до слияния, или другими словами увидеть, что привнесло данное слияние, вы можете выполнить git diff --ours

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

Итак, здесь мы можем легко увидеть, что же произошло с нашей веткой, какие изменения в действительности внесло слияние в данный файл — изменение только одной строки.

Если вы хотите узнать, чем результат слияния отличается от сливаемой ветки, то можете выполнить команду `git diff --theirs`. В этом и следующем примере мы используем опцию `-w` для того, чтобы не учитывать изменения в пробельных символах, так как мы сравниваем результат с тем, что есть в Git, а не с нашим исправленным файлом `hello.theirs.rb`.

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello mundo'
 end
```

И, наконец, вы можете узнать, как изменился файл по сравнению сразу с обеими ветками с помощью команды `git diff --base`.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby
```

```
+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end


 hello()
```

В данный момент мы можем использовать команду git clean для того, чтобы удалить не нужные более дополнительные файлы, созданные нами для выполнения слияния.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Использование команды checkout в конфликтах

Возможно, нас по каким-то причинам не устраивает необходимость выполнения слияния в текущий момент, или мы не можем хорошо исправить конфликт и нам необходимо больше информации.

Давайте немного изменим пример. Предположим, что у нас есть две долгоживущих ветки, каждая из которых имеет несколько коммитов, что при слиянии приводит к справедливому конфликту.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) Update README
* 9af9d3b Create README
* 694971d Update phrase to 'hola world'
| * e3eb223 (mundo) Add more tests
| * 7cff591 Create initial testing script
| * c3ffff1 Change text to 'hello mundo'
|/
* b7dcc89 Initial hello world code
```

У нас есть три уникальных коммита, которые присутствуют только в ветке master и три других, которые присутствуют в ветке mundo. Если мы попытаемся слить ветку mundo, то получим конфликт.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Мы хотели бы увидеть в чем состоит данный конфликт. Если мы откроем конфликтующий файл, то увидим нечто подобное:

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

В обеих сливаемых ветках в этот файл было добавлено содержимое, но в некоторых коммитах изменялись одни и те же строки, что и привело к конфликту.

Давайте рассмотрим несколько находящихся в вашем распоряжении инструментов, которые позволяют определить, как возник этот конфликт. Возможно, не понятно, как именно вы должны исправить конфликт и вам требуется больше информации.

Полезным в данном случае инструментом является команда `git checkout` с опцией `--conflict`. Она заново выкачает файл и заменит маркеры конфликта. Это может быть полезно, если вы хотите восстановить маркеры конфликта и попробовать разрешить его снова.

В качестве значения опции --conflict вы можете указывать diff3 или merge (последнее значение используется по умолчанию). Если вы укажете diff3, Git будет использовать немного другую версию маркеров конфликта — помимо «нашей» и «их» версий файлов будет также отображена «базовая» версия, и таким образом вы получите больше информации.

```
$ git checkout --conflict=diff3 hello.rb
```

После того, как вы выполните эту команду, файл будет выглядеть так:

```
#! /usr/bin/env ruby


def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end


hello()
```

Если вам нравится такой формат вывода, то вы можете использовать его по умолчанию для будущих конфликтов слияния, установив параметру merge.conflictstyle значение diff3.

```
$ git config --global merge.conflictstyle diff3
```

Команде git checkout также можно передать опции --ours и --theirs, которые позволяют действительно быстро выбрать одну из версий файлов, не выполняя слияния совсем.

Это может быть действительно полезным при возникновении конфликтов в бинарных файлах (в этом случае вы можете просто выбрать одну из версий), или при необходимости слить из другой ветки только

некоторые файлы (в этом случае вы можете выполнить слияние, а затем перед коммитом переключить нужные файлы на требуемые версии).

История при слиянии

Другой полезный инструмент при разрешении конфликтов слияния — это команда `git log`. Она поможет вам получить информацию о том, что могло привести к возникновению конфликтов. Временами может быть очень полезным просмотреть историю, чтобы понять почему в двух ветках разработки изменялась одна и та же область кода.

Для получения полного списка всех уникальных коммитов, которые были сделаны в любой из сливаемых веток, мы можем использовать синтаксис «трёх точек.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 Update README
< 9af9d3b Create README
< 694971d Update phrase to 'hola world'
> e3eb223 Add more tests
> 7cff591 Create initial testing script
> c3ffff1 Change text to 'hello mundo'
```

Это список всех шести коммитов, включённых в слияние, с указанием также ветки разработки, в которой находится каждый из коммитов.

Мы также можем сократить его, попросив предоставить нам более специализированную информацию. Если мы добавим опцию `--merge` к команде `git log`, то она покажет нам только те коммиты, в которых изменялся конфликтующий в данный момент файл.

```
$ git log --oneline --left-right --merge
< 694971d Update phrase to 'hola world'
> c3ffff1 Change text to 'hello mundo'
```

Если вы выполните эту команду с опцией -p, то получите только список изменений файла, на котором возник конфликт. Это может быть **действительно** полезным для быстрого получения информации, которая необходима, чтобы понять почему что-либо конфликтует и как наиболее правильно это разрешить.

Комбинированный формат изменений

Так как Git добавляет в индекс все успешные результаты слияния, то при вызове `git diff` в состоянии конфликта слияния будет отображено только то, что сейчас конфликтует. Это может быть полезно, так как вы сможете увидеть какие ещё конфликты нужно разрешить.

Если вы выполните `git diff` сразу после конфликта слияния, то получите информацию в довольно своеобразном формате.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
 #! /usr/bin/env ruby

 def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
++>>>>>>> mundo
 end

 hello()
```

Такой формат называется «комбинированным» («Combined Diff»), для каждого различия в нем содержится два раздела с информацией. В первом разделе отображены различия строки (добавлена она или удалена) между «вашей» веткой и содержимым вашего рабочего каталога, а во втором разделе содержится то же самое, но между «их» веткой и рабочим каталогом.

Таким образом, в данном примере вы можете увидеть строки `<<<<<<<` и `>>>>>>>` в файле в вашем рабочем каталоге, хотя они

отсутствовали в сливаемых ветках. Это вполне оправдано, потому что, добавляя их, инструмент слияния предоставляет вам дополнительную информацию, но предполагается, что мы удалим их.

Если мы разрешим конфликт и снова выполним команду git diff, то получим ту же информацию, но в немного более полезном представлении.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
 #! /usr/bin/env ruby

 def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
 end

 hello()
```

В этом выводе указано, что строка «hola world» при слиянии присутствовала в «нашей» ветке, но отсутствовала в рабочей копии, строка «hello mundo» была в «их» ветке, но не в рабочей копии, и, наконец, «hola mundo» не была ни в одной из сливаемых веток, но сейчас присутствует в рабочей копии. Это бывает полезно просмотреть перед коммитом разрешения конфликта.

Такую же информацию вы можете получить и после выполнения слияния с помощью команды git log, узнав таким образом как был разрешён конфликт. Git выводит информацию в таком формате, если вы выполните

git show для коммита слияния или вызовете команду git log -p с опцией --cc (без неё данная команда не показывает изменения для коммитов слияния).

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

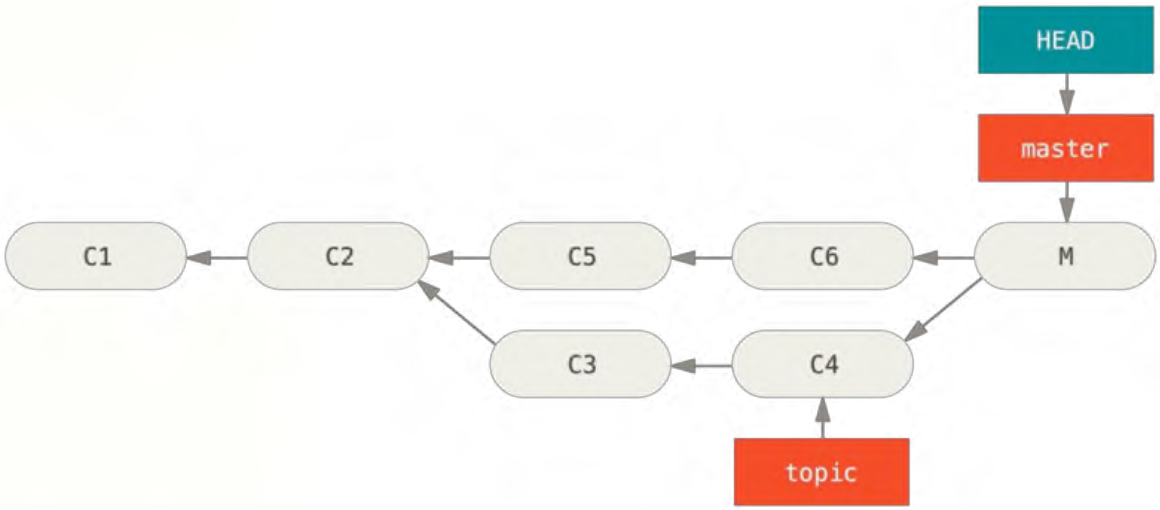
  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Отмена слияний

Теперь, когда вы знаете, как создать коммит слияния, вы можете сделать его по ошибке. Одна из замечательных вещей в работе с Git — это то, что ошибки совершать не страшно, так как есть возможность исправить их (и в большинстве случаев сделать это просто).

Коммит слияния не исключение. Допустим, вы начали работать в тематической ветке, случайно слили ее в `master`, и теперь ваша история коммитов выглядит следующим образом:

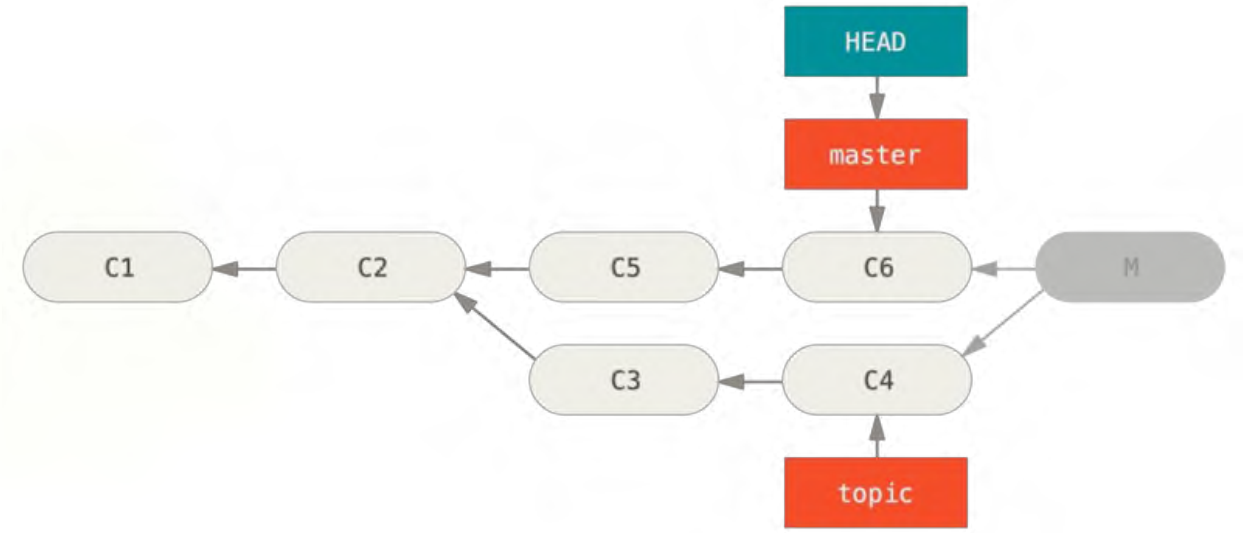


Случайный коммит слияния

Есть два подхода к решению этой проблемы, в зависимости от того, какой результат вы хотите получить.

Исправление ссылок

Если нежелаемый коммит слияния существует только в вашем локальном репозитории, то простейшее и лучшее решение состоит в перемещении веток так, чтобы они указывали туда куда вам нужно. В большинстве случаев, если вы после случайного `git merge` выполните команду `git reset --hard HEAD~`, то указатели веток восстановятся так, что будут выглядеть следующим образом:



История после `git reset --hard HEAD~`

Небольшое напоминание: `reset --hard` обычно выполняет три шага:

1. Перемещает ветку, на которую указывает HEAD. В данном случае мы хотим переместить `master` туда, где она была до коммита слияния (C6).
2. Приводит индекс к такому же виду что и HEAD.
3. Приводит рабочий каталог к такому же виду, что и индекс.

Недостаток этого подхода состоит в изменении истории, что может привести к проблемам в случае совместно используемого репозитория. Кратко говоря, если у других людей уже есть какие-то из изменяемых вами коммитов, вы должны отказаться от использования `reset`. Этот подход также не будет работать, если после слияния уже был сделан хотя бы один коммит; перемещение ссылки фактически приведёт к потере этих изменений.

Отмена коммита

Если перемещение указателей ветки вам не подходит, Git предоставляет возможность сделать новый коммит, который откатывает все изменения, сделанные в другом. Git называет эту операцию «восстановлением» («revert»), в данном примере вы можете вызвать её следующим образом:

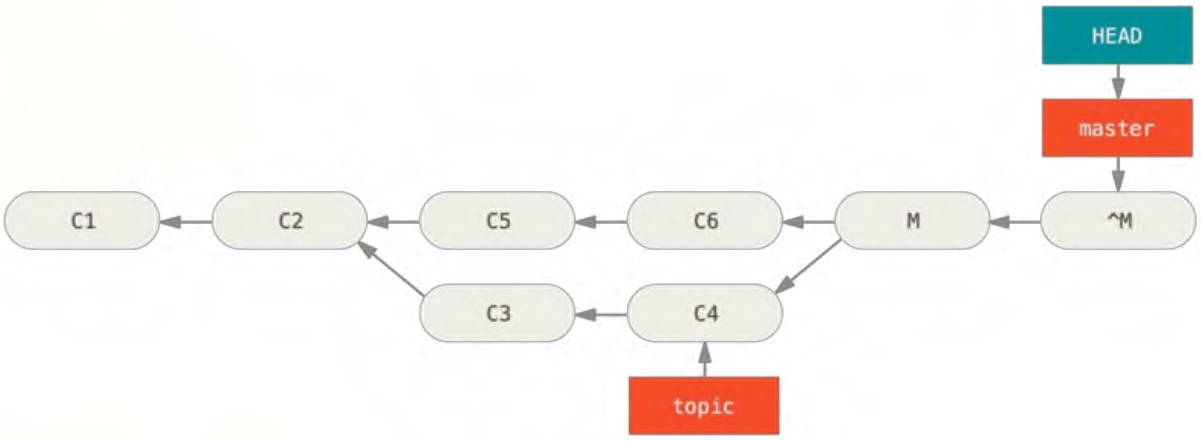
```
$ git revert -m 1 HEAD

[master b1d8379] Revert "Merge branch 'topic'"
```

Опция `-m 1` указывает какой родитель является «основной веткой» и должен быть сохранен. Когда вы выполняете слияние в `HEAD` (git merge topic), новый коммит будет иметь двух родителей: первый из них HEAD (C6),

а второй — вершина ветки, которую сливают с текущей (C4). В данном случае мы хотим отменить все изменения, внесённые слиянием родителя #2 (C4), и сохранить при этом всё содержимое из родителя #1 (C6).

История с коммитом восстановления (отменой коммита слияния) выглядит следующим образом:



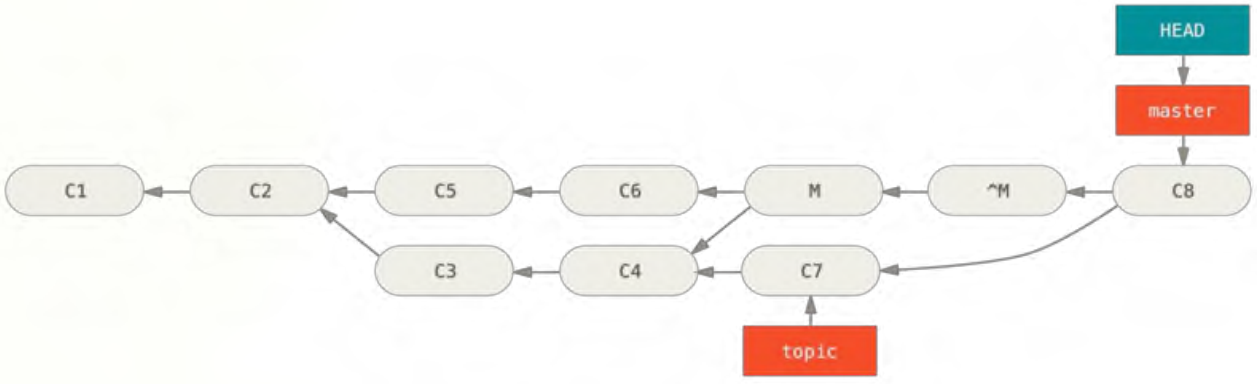
История после git revert -m 1

Новый коммит ^M имеет точно такое же содержимое как C6, таким образом, начиная с нее всё выглядит так, как будто слияние никогда не выполнялось, за тем лишь исключением, что «теперь уже не слитые» коммиты всё также присутствуют в истории HEAD. Git придет в замешательство, если вы вновь попытаетесь слить topic в ветку master:

```
$ git merge topic

Already up-to-date.
```

В ветке topic нет ничего, что ещё недоступно из ветки master. Плохо, что в случае добавления новых наработок в topic, при повторении слияния Git добавит **только** те изменения, которые были сделаны после отмены слияния:



История с плохим слиянием

Лучшим решением данной проблемы является откат коммита отмены слияния, так как теперь вы хотите внести изменения, которые были отменены, а **затем** создание нового коммита слияния:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

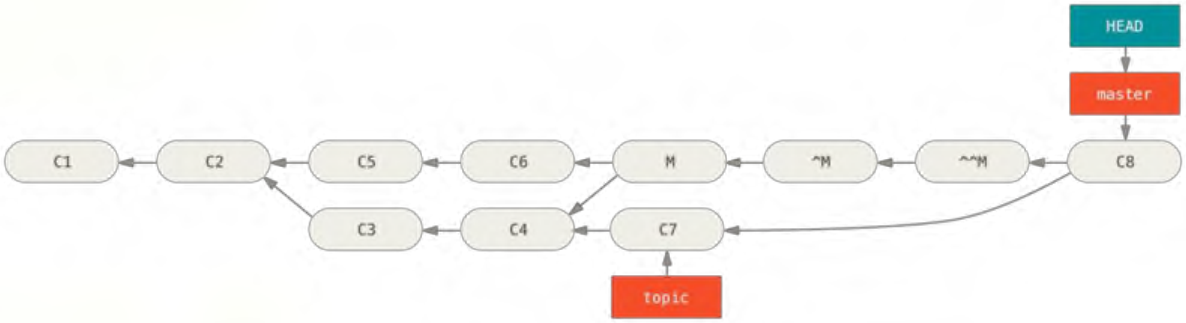


Рисунок 141. История после повторения отменённого слияния

В этом примере, M и ^M отменены. В коммите ^^M, фактически, сливаются изменения из C3 и C4, а в C8 — изменения из C7, таким образом, ветка topic полностью слита.

Другие типы слияний

До этого момента мы рассматривали типичные слияния двух веток, которые обычно выполняются с использованием стратегии слияния, называемой «рекурсивной». Но существуют и другие типы слияния веток. Давайте кратко рассмотрим некоторые из них.

Выбор «нашей» или «их» версий

Во-первых, существует ещё один полезный приём, который мы можем использовать в обычном «рекурсивном» режиме слияния. Мы уже видели опции ignore-all-space и ignore-space-change, которые передаются с префиксом -X, но мы можем также попросить Git при возникновении конфликта использовать ту или иную версию файлов.

По умолчанию, когда Git при слиянии веток замечает конфликт, он добавляет в код маркеры конфликта, отмечает файл как конфликтующий и позволяет вам разрешить его. Если же вместо ручного разрешения конфликта вы хотите, чтобы Git просто использовал какую-то определённую версию файла, а другую игнорировал, то вы можете передать команде merge одну из двух опций -Xours или -Xtheirs.

В этом случае Git не будет добавлять маркеры конфликта. Все неконфликтующие изменения он сольёт, а для конфликтующих он целиком

возьмёт ту версию, которую вы указали (это относится и к бинарным файлам).

Если мы вернёмся к примеру «hello world», который использовали раньше, то увидим, что попытка слияния в нашу ветку приведёт к конфликту.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

Однако, если мы выполним слияние с опцией `-Xours` или `-Xtheirs`, конфликта не будет.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

В этом случае, вместо добавления в файл маркеров конфликта с «hello mundo» в качестве одной версии и с «hola world» в качестве другой, Git просто выберет «hola world». Однако, все другие неконфликтующие изменения будут слиты успешно.

Такая же опция может быть передана команде `git merge-file`, которую мы обсуждали ранее, то есть для слияния отдельных файлов можно использовать команду `git merge-file --ours`.

На случай если вам нужно нечто подобное, но вы хотите, чтобы Git даже не пытался сливать изменения из другой версии, существует более суровый вариант — **стратегия** слияния «ours». Важно отметить, что это не то же самое что **опция** «ours» рекурсивной стратегии слияния.

Фактически, эта стратегия выполнит ненастоящее слияние. Она создаст новый коммит слияния, у которой родителями будут обе ветки, но при этом данная стратегия даже не взглянет на ветку, которую вы сливаете.

В качестве результата слияния она просто оставляет тот код, который находится в вашей текущей ветке.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Вы можете видеть, что между веткой, в которой мы были, и результатом слияния нет никаких отличий.

Это часто бывает полезно, когда нужно заставить Git считать, что ветка уже слита, а реальное слияние отложить на потом. Для примера предположим, что вы создали ветку `release` и проделали в ней некоторую работу, которую когда-то впоследствии захотите слить обратно в master. Тем временем в `master` были сделаны некоторые исправления, которые необходимо перенести также в вашу ветку `release`. Вы можете слить ветку с исправлениями в `release`, а затем выполнить `merge -s ours` этой ветки в `master` (хотя исправления в ней уже присутствуют), так что позже, когда вы будете снова сливать ветку `release`, не возникнет конфликтов, связанных с этими исправлениями.

Слияние поддеревьев

Идея слияния поддеревьев состоит в том, что у вас есть два проекта и один из проектов отображается в подкаталог другого. Когда вы выполняете слияние поддеревьев, Git в большинстве случаев способен понять, что одно из них является поддеревом другого и выполнить слияние подходящим способом.

Далее мы рассмотрим пример добавления в существующий проект другого проекта и последующее слияние кода второго проекта в подкаталог первого.

Первым делом мы добавим в наш проект приложение Rack. Мы добавим Rack в наш собственный проект, как удалённый репозиторий, а затем выгрузим его в отдельную ветку.

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
```

```
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Таким образом, теперь у нас в ветке `rack_branch` находится основная ветка проекта Rack, а в ветке `master` — наш собственный проект. Если вы переключитесь сначала на одну ветку, а затем на другую, то увидите, что они имеют абсолютно разное содержимое:

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile  contrib  lib
COPYING      README        bin       example  test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Может показаться странным, но, на самом деле, ветки в вашем репозитории не обязаны быть ветками одного проекта. Это мало распространено, так как редко бывает полезным, но иметь ветки, имеющие абсолютно разные истории, довольно легко.

В данном примере мы хотим выгрузить проект Rack в подкаталог нашего основного проекта. В Git мы можем выполнить это с помощью

команды git read-tree. Сейчас же вам достаточно знать, что она считывает содержимое некоторой ветки в ваш текущий индекс и рабочий каталог. Мы просто переключимся обратно на ветку master и выгрузим ветку rack_branch в подкаталог rack ветки master нашего основного проекта:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Когда мы будем выполнять коммит, он будет выглядеть так, как будто все файлы проекта Rack были добавлены в этот подкаталог — например, мы скопировали их из архива. Важно отметить, что слить изменения одной из веток в другую довольно легко. Таким образом, если проект Rack обновился, мы можем получить изменения из его репозитория просто переключившись на соответствующую ветку и выполнив операцию git pull:

```
$ git checkout rack_branch

$ git pull
```

Затем мы можем слить эти изменения обратно в нашу ветку master.

Для того, чтобы получить изменения и заполнить сообщение коммита используйте параметр --squash, вместе с опцией -Xsubtree рекурсивной стратегии слияния. Вообще-то, по умолчанию используется именно рекурсивная стратегия слияния, но мы указали и её тоже для пущей ясности.

```
$ git checkout master

$ git merge --squash -s recursive -Xsubtree=rack rack_branch

Squash commit -- not updating HEAD

Automatic merge went well; stopped before committing as requested
```

Все изменения из проекта Rack слиты и подготовлены для локального коммита. Вы также можете поступить наоборот — сделать изменения в подкаталоге rack вашей основной ветки и затем слить их в вашу ветку rack_branch, чтобы позже передать их ответственным за проекты или отправить их в вышестоящий репозиторий проекта Rack.

Таким образом, слияние поддеревьев даёт нам возможность использовать рабочий процесс в некоторой степени похожий на рабочий процесс с подмодулями, но при этом без использования подмодулей (которые мы рассмотрим в Подмодули). Мы можем держать ветки с другими связанными проектами в нашем репозитории и периодически

сливать их как поддеревья в наш проект. С одной стороны, это удобно, например, тем, что весь код хранится в одном месте. Однако, при этом есть и некоторые недостатки — поддеревья немного сложнее, проще допустить ошибки при повторной интеграции изменений или случайно отправить ветку не в тот репозиторий.

Другая небольшая странность состоит в том, что для получения различий между содержимым подкаталога rack и содержимого ветки rack_branch — для того, чтобы увидеть необходимо ли выполнять слияния между ними — вы не можете использовать обычную команду diff. Вместо этого следует выполнить команду git diff-tree, указав ветку, с которой производится сравнение:

```
$ git diff-tree -p rack_branch
```

Вот как выглядит процедура сравнения содержимого подкаталога rack с содержимым ветки master на сервере после последнего скачивания изменений:

```
$ git diff-tree -p rack_remote/master
```

Rerere

Функциональность git rerere — частично скрытый компонент Git. Её имя является сокращением для «reuse recorded resolution» («повторно использовать сохранённое решение»). Как следует из названия, эта функциональность позволяет попросить Git запомнить то, как вы разрешили некоторую часть конфликта, так что в случае возникновения такого же конфликта, Git сможет его разрешить автоматически.

Существует несколько ситуаций, в которых данная функциональность может быть действительно удобна. Один из примеров, упомянутый в документации, состоит в том, чтобы обеспечить в будущем простоту слияния некоторой долгоживущей ветки, не создавая при этом набор промежуточных коммитов слияния. При использовании rerere вы можете время от времени выполнять слияния, разрешать конфликты, а затем откатывать слияния. Если делать это постоянно, то итоговое слияние должно пройти легко, так как rerere сможет разрешить все конфликты автоматически.

Такая же тактика может быть использована, если вы хотите сохранить ветку легко перебазируемой, то есть вы не хотите сталкиваться с одними и теми же конфликтами каждый раз при перебазировании. Или, например, вы решили ветку, которую уже сливали и разрешали при этом

некоторые конфликты, вместо слияния перебазировать — навряд ли вы захотите разрешать те же конфликты ещё раз.

Другая ситуация возникает, когда вы изредка сливаете несколько веток, относящихся к ещё разрабатываемым задачам, в одну тестовую ветку, как это часто делается в проекте самого Git. Если тесты завершатся неудачей, вы можете откатить все слияния и повторить их, исключив из них ветку, которая поломала тесты, при этом не разрешая конфликты снова.

Для того, чтобы включить функциональность rerere, достаточно изменить настройки следующим образом:

```
$ git config --global rerere.enabled true
```

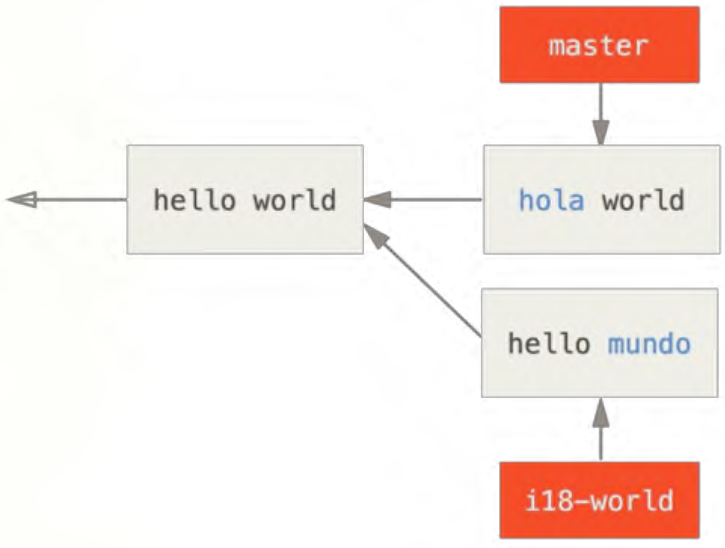
Также вы можете включить её, создав каталог .git/rr-cache в нужном репозитории, но включение через настройки понятнее и может быть сделано глобально.

Давайте рассмотрим простой пример, подобный используемому ранее. Предположим, у нас есть файл вида:

```
#! /usr/bin/env ruby


def hello
  puts 'hello world'
end
```

Как и ранее, в одной ветке мы изменили слово «hello» на «hola», а в другой — слово «world» на «mundo».



Когда мы будем сливать эти две ветки в одну, мы получим конфликт:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

Вы должно быть заметили в выводе новую строку `Recorded preimage for FILE`. Во всём остальном вывод такой же, как при обычном конфликте слияния. В этот момент `rerere` может сообщить нам несколько вещей. Обычно в такой ситуации вы можете выполнить `git status`, чтобы увидеть в чем заключается конфликт:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#         both modified:      hello.rb
#
```

Однако, с помощью команды `git rerere status` вы также можете узнать, для каких файлов `git rerere` сохранил снимки состояния, в котором они были до начала слияния:

```
$ git rerere status
hello.rb
```

А команда `git rerere diff` показывает текущее состояние разрешения конфликта — то, с чего вы начали разрешать конфликт, и то, как вы его разрешили (фактически, патч, который в дальнейшем можно использовать для разрешения такого же конфликта).

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
```

```
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby


 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
+<<<<<<< HEAD
   puts 'hola world'
->>>>>>>
+=======
+  puts 'hello mundo'
+>>>>>>> i18n-world
 end
```

Также (и это уже не относится к rerere), вы можете использовать команду `ls-files -u`, чтобы увидеть конфликтующие файлы, их общую родительскую версию и обе сливаемых версии:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1 hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2 hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3 hello.rb
```

Теперь вы можете разрешить конфликт, используя `puts 'hola mundo'`, и снова выполнить команду `rerere diff`, чтобы увидеть, что именно `rerere` запомнит:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
```

```
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

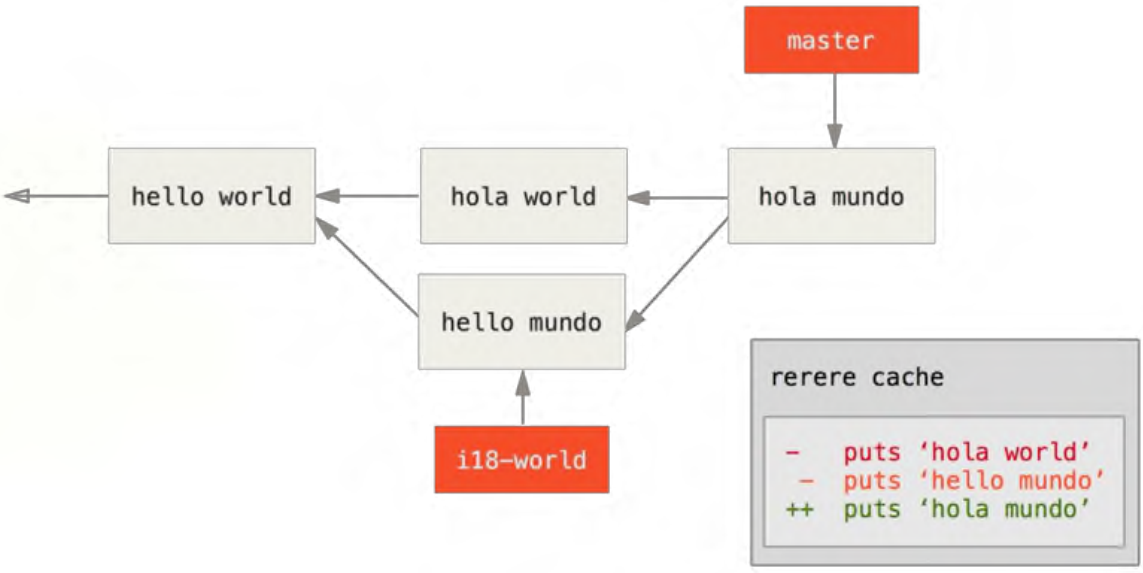
 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

То есть, когда Git увидит в файле hello.rb конфликт, в котором с одной строны стоит «hello mundo» и «hola world» с другой, он разрешит его как «hola mundo».

Теперь мы можем отметить конфликт как разрешённый и закоммитить его:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Как вы видите, при этом было «сохранено разрешение конфликта для ФАЙЛА» («Recorded resolution for FILE»).



Теперь давайте отменим это слияние и перебазируем ветку i18n-world поверх master. Как мы видели в Раскрытие тайн reset, мы можем переместить нашу ветку назад, используя команду reset.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Наше слияние отменено. Теперь давайте перебазируем ветку i18n-world.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'
```

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
```

```
Patch failed at 0001 i18n one word
```

При этом мы получили ожидаемый конфликт слияния, но обратите внимание на строку `Resolved FILE using previous resolution`. Если мы посмотрим на содержимое файла, то увидим, что конфликт уже был разрешён, и в файле отсутствуют маркеры конфликта слияния.

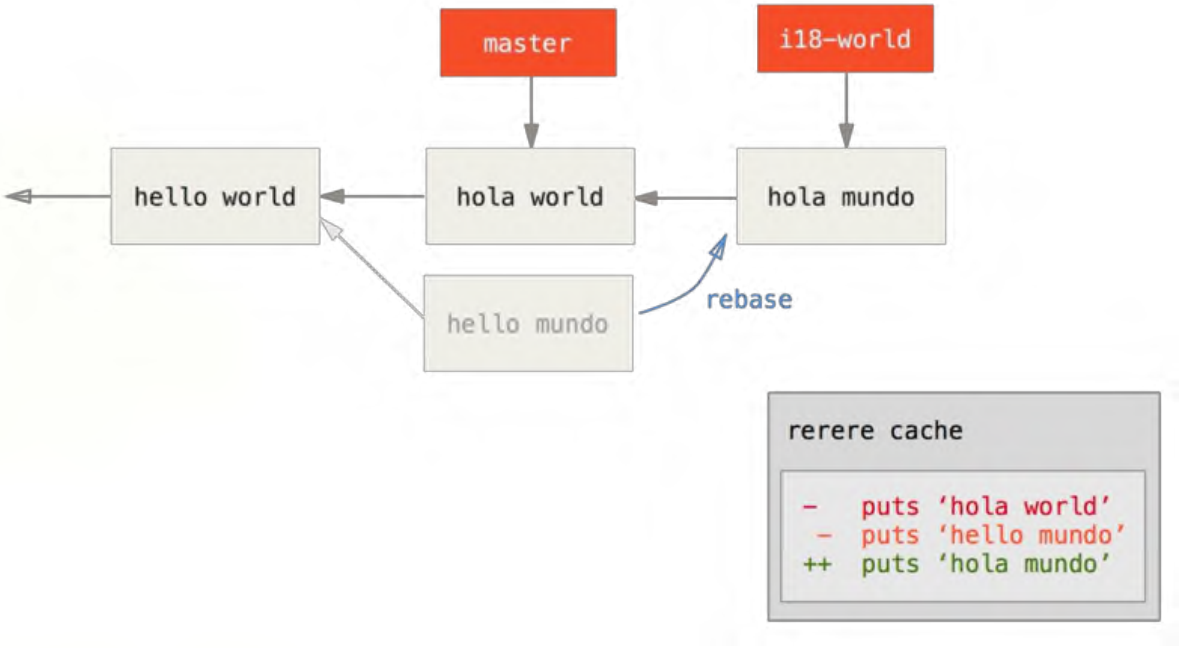
```
#! /usr/bin/env ruby


def hello
  puts 'hola mundo'
end
```

При этом команда `git diff` покажет вам **как именно** этот конфликт был автоматически повторно разрешён:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby


  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end
```



С помощью команды checkout вы можете вернуть этот файл назад в конфликтующее состояние:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

Мы видели пример этого в Продвинутое слияние. Теперь давайте повторно разрешим конфликт используя rerere:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby
```

```
def hello
  puts 'hola mundo'
end
```

Мы автоматически повторно разрешили конфликт, используя сохранённый rerere вариант разрешения. Теперь вы можете добавить файл в индекс и продолжить перебазирование ветки.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

Итак, если вы выполняете много повторных слияний или хотите сохранять тематическую ветку в состоянии, актуальном вашей основной ветке, без множества слияний в истории, или часто перебазируете ветки, то вы можете включить rerere. Это, в какой-то мере, упростит вам жизнь.

Обнаружение ошибок с помощью Git

Git предоставляет несколько инструментов, которые помогут вам найти и устранить проблемы в ваших проектах. Так как Git рассчитан на работу с проектом почти любого типа, эти инструменты имеют довольно обобщённые возможности, но часто они могут помочь вам отловить ошибку или её виновника.

Аннотация файла

Если вы обнаружили ошибку в вашем коде и хотите знать, когда она была добавлена и почему, то в большинстве случаев аннотация файла будет лучшим инструментом для этого. С помощью неё для любого файла можно увидеть, каким коммитом последний раз изменяли каждую из строк. Поэтому если вы видите, что некоторый метод в вашем коде работает неправильно, вы можете с помощью команды git blame снабдить файл аннотацией, и таким образом увидеть, когда каждая строка метода была изменена последний раз и кем.

В следующем примере используется git blame, чтобы определить, какой коммит и коммиттер отвечал за строки в Makefile ядра Linux верхнего уровня и, кроме того, использует параметр -L для ограничения вывода аннотации строками с 69 по 82 из этого файла:

```
$ git blame -L 69,82 Makefile
```

```
b8b0618cf6fab (Cheng Renquan  2009-05-26 16:03:07 +0800 69) ifeq ("$(origin V)", "command line")
b8b0618cf6fab (Cheng Renquan  2009-05-26 16:03:07 +0800 70) KBUILD_VERBOSE = $(V)
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 71) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 72) ifndef KBUILD_VERBOSE
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 73) KBUILD_VERBOSE = 0
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 74) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 75)
066b7ed955808 (Michal Marek  2014-07-04 14:29:30 +0200 76) ifeq ($(KBUILD_VERBOSE),1)
066b7ed955808 (Michal Marek  2014-07-04 14:29:30 +0200 77)  quiet =
066b7ed955808 (Michal Marek  2014-07-04 14:29:30 +0200 78)  Q =
066b7ed955808 (Michal Marek  2014-07-04 14:29:30 +0200 79) else
066b7ed955808 (Michal Marek  2014-07-04 14:29:30 +0200 80) quiet=quiet_
066b7ed955808 (Michal Marek  2014-07-04 14:29:30 +0200 81)  Q = @
066b7ed955808 (Michal Marek  2014-07-04 14:29:30 +0200 82) endif
```

Обратите внимание, что первое поле — это неполная SHA-1 сумма последнего коммита, который изменял соответствующую строку. Следующими двумя полями являются значения, извлечённые из этого коммита — имя автора и время создания коммита — таким образом, вы можете легко увидеть кто изменял строку и когда. После этого следуют номер строки и содержимое файла. Обратите внимание на строки со значением `^4832fe2` в поле коммита, так обозначаются те строки, которые были в первом коммите этого файла. Этот коммит был сделан, когда данный файл был впервые добавлен в проект и с тех пор эти строки не были изменены. Немного сбивает с толку то, что вы уже видели в Git, по крайней мере, три различных варианта использования символа `^` для изменения SHA-1 коммита, в данном случае этот символ имеет такое значение.

Другая отличная вещь в Git — это то, что он не отслеживает явно переименования файлов (пользователю не нужно **явно** указывать какой файл в какой был переименован). Он сохраняет снимки и уже после выполнения самого переименования неявно попытается выяснить, что было переименовано. Одна из интересных возможностей, вытекающих из этого — это то, что вы также можете попросить Git выявить перемещения кода всех других видов. Если передать опцию `-C` команде `git blame`, Git проанализирует аннотируемый файл и пытается выяснить откуда изначально появились фрагменты кода, если они, конечно же, были откуда-то скопированы. Например, предположим при реорганизации кода в файле `GITServerHandler.m` вы разнесли его по нескольким файлам, один из которых `GITPackUpload.m`. Вызывая `git blame` с опцией `-C` для файла `GITPackUpload.m`, вы можете увидеть откуда изначально появились разные фрагменты этого файла.

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString *parentSha;
ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit *commit = [g
ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151) if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152) [refDict setOb
```

```
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Это, действительно, полезно. Обычно вы получаете в качестве изначального коммит, в котором вы скопировали код, так как это первый коммит, в котором вы обращаетесь к этим строкам в **этом** файле. Но в данном случае Git сообщает вам первый коммит, в котором эти строки были написаны, даже если это было сделано в другом файле.

Бинарный поиск

Аннотирование файла помогает, если вы знаете, где находится проблема и можете начать исследование с этого места. Если вы не знаете, что сломано, а с тех пор, как код работал, были сделаны десятки или сотни коммитов, вы, вероятно, воспользуетесь командой `git bisect`. Эта команда выполняет бинарный поиск по истории коммитов для того, чтобы помочь вам как можно быстрее определить коммит, который создал проблему.

Допустим, вы только что развернули некоторую версию вашего кода в боевом окружении и теперь получаете отчёты о некоторой ошибке, которая не возникала в вашем разработческом окружении, и вы не можете представить, почему код ведет себя так. Вы возвращаетесь к вашему коду и выясняете, что можете воспроизвести проблему, но всё ещё не понимаете, что работает неверно. Вы можете воспользоваться бинарным поиском, чтобы выяснить это. Во-первых, выполните команду `git bisect start` для запуска процесса поиска, а затем используйте `git bisect bad`, чтобы сообщить Git, что текущий коммит сломан. Затем, используя `git bisect good [good_commit]`, вы должны указать, когда было последнее известное рабочее состояние:

```
$ git bisect start

$ git bisect bad

$ git bisect good v1.0

Bisecting: 6 revisions left to test after this

[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] Error handling on repo
```

Git выяснил, что произошло около 12 коммитов между коммитом, который вы отметили как последний хороший коммит (v1.0), и текущим плохим коммитом, и выгрузил вам один из середины. В этот момент вы можете запустить ваши тесты, чтобы проверить присутствует ли проблема в этом коммите. Если это так, значит она была внесена до выгруженного промежуточного коммита, если нет, значит проблема была внесена после

этого коммита. Пусть в данном коммите проблема не проявляется, вы сообщаете об этом Git с помощью `git bisect good` и продолжаете ваше путешествие:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] Secure this thing
```

Теперь вы оказались на другом коммите, расположенном посредине между только что протестированным и плохим коммитами. Вы снова выполняете ваши тесты, обнаруживаете, что текущий коммит сломан, и сообщаете об этом Git с помощью команды `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] Drop exceptions table
```

Этот коммит хороший и теперь Git имеет всю необходимую информацию для определения того, где была внесена ошибка. Он сообщает вам SHA-1 первого плохого коммита и отображает некоторую информацию о коммите и файлах, которые были изменены в этом коммите, так, чтобы вы смогли разобраться что же, случилось, что могло привнести эту ошибку:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    Secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

Когда вы закончили бинарный поиск, нужно выполнить git bisect reset для того, чтобы вернуть HEAD туда, где он был до начала поиска, иначе вы останетесь в, довольно, причудливом состоянии:

```
$ git bisect reset
```

Это мощный инструмент, который помогает вам за считанные минуты проверить сотни коммитов на возможность внесения ошибки. В действительности, если у вас есть скрипт, который будет возвращать 0 если проект находится в рабочем состоянии и любое другое число в обратном случае, то вы можете полностью автоматизировать git bisect. Сперва, вы снова сообщаете границы бинарного поиска, указывая известные плохие и хорошие коммиты. Вы можете сделать это, передавая их команде bisect start — первым аргументом известный плохой коммит, а вторым известный хороший коммит:

```
$ git bisect start HEAD v1.0

$ git bisect run test-error.sh
```

Это приведёт к автоматическому выполнению test-error.sh на каждый выгруженный коммит до тех пор, пока Git не найдёт первый сломанный коммит. Вы также можете использовать что-то вроде make или make tests, или что-то ещё, что у вас есть для запуска автоматизированных тестов.

Подмодули

Часто при работе над одним проектом, возникает необходимость использовать в нем другой проект. Возможно, это библиотека, разрабатываемая сторонними разработчиками или вами, но в рамках отдельного проекта, и используемая в нескольких других проектах. Типичная проблема, возникающая при этом — вы хотите продолжать работать с двумя проектами по отдельности, но при этом использовать один из них в другом.

Приведём пример. Предположим, вы разрабатываете веб-сайт и создаёте ленту в формате Atom. Вместо написания собственного генератора Atom, вы решили использовать библиотеку. Скорее всего, вы либо подключите библиотеку как сторонний модуль, такой как модуль CPAN или Ruby Gem, либо скопируете исходный код библиотеки в свой проект. Проблема использования библиотеки состоит в трудности её адаптации под собственные нужды и часто более сложным является её распространение, так как нужно быть уверенным, что каждому клиенту доступна изменённая версия. При включении кода библиотеки в свой проект проблемой будет объединение ваших изменений в ней с новыми изменениями в оригинальном репозитории.

Git решает эту проблему с помощью подмодулей. Подмодули позволяют вам сохранить один Git-репозиторий, как подкаталог другого Git-репозитория. Это даёт вам возможность клонировать в ваш проект другой репозиторий, но коммиты при этом хранить отдельно.

Начало работы с подмодулями

Далее мы рассмотрим процесс разработки простого проекта, разбитого на один главный проект и несколько подпроектов.

Давайте начнём с добавления, существующего Git-репозитория как подмодуля репозитория, в котором мы работаем. Для добавления нового подмодуля используйте команду git submodule add, указав относительный или абсолютный URL проекта, который вы хотите начать отслеживать. В данном примере мы добавим библиотеку «DbConnector».

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

По умолчанию подмодуль сохраняется в каталог с именем репозитория, в нашем примере — «DbConnector». Изменить каталог сохранения подмодуля можно указав путь к нему в конце команды.

Если в данный момент выполнить git status, то можно заметить следующее.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

 new file:   .gitmodules
```

```
new file:   DbConnector
```

Во-первых, вы увидите новый файл .gitmodules. Это конфигурационный файл, в котором хранится соответствие между URL проекта и локальным подкаталогом, в который вы его выкачали:

```
[submodule "DbConnector"]
 path = DbConnector
 url = https://github.com/chaconinc/DbConnector
```

Если у вас несколько подмодулей, то в этом файле у вас будет несколько записей. Важно заметить, что этот файл добавлен под версионный контроль Git так же, как и другие ваши файлы, например, ваш файл .gitignore. Этот файл можно получить или отправить на сервер вместе с остальными файлами проекта. Благодаря этому другие люди, которые клонируют ваш проект, узнают откуда взять подмодули проекта.

Примечание	Поскольку другие люди первым делом будут пытаться выполнить команды clone/fetch по URL, указанным в файле .gitmodules, старайтесь проверять, что URL будут им доступны. Например, если вы выполняете отправку по URL, отличному от того, по которому другие люди получают данные, то используйте тот URL, к которому у других участников будет доступ. В таком случае для себя вы можете задать локальное значение URL с помощью команды git config submodule.DbConnector.url PRIVATE_URL. Если применимо, относительные URL могут помочь.

Следующим пунктом в списке вывода команды git status будет запись о каталоге проекта. Команда git diff покажет для него кое-что интересное:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
```

```
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Хоть DbConnector и является подкаталогом вашего рабочего каталога, Git распознает его как подмодуль и не отслеживает его содержимое пока вы не перейдёте в него. Вместо этого, Git видит его как конкретный коммит этого репозитория.

Если вам нужен немного более понятный вывод, передайте команде git diff параметр --submodule.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

При создании коммита вы увидите следующее:

```
$ git commit -am 'Add DbConnector module'
[master fb9093c] Add DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Обратите внимание на права доступа 160000 у DbConnector. Это специальные права доступа в Git, которые означают, что вы сохраняете коммит как элемент каталога, а не как подкаталог или файл.

Наконец, отправьте эти изменения:

```
$ git push origin master
```

Клонирование проекта с подмодулями

Далее мы рассмотрим клонирование проекта, содержащего подмодули. Когда вы клонируете такой проект, по умолчанию вы получите каталоги, содержащие подмодули, но ни одного файла в них не будет:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x   9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x   7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x  13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--   1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x   2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--   1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x   3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

Каталог `DbConnector` присутствует, но он пустой. Вы должны выполнить две команды: `git submodule init` — для инициализации локального конфигурационного файла, и `git submodule update` — для

получения всех данных этого проекта и извлечения соответствующего коммита, указанного в основном проекте.

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Теперь ваш каталог DbConnector находится в точно таком же состоянии, как и ранее при выполнении коммита.

Однако, существует немного более простой вариант сделать то же самое. Если вы передадите опцию `--recurse-submodules` команде `git clone`, то она автоматически инициализирует и обновит каждый подмодуль в этом репозитории, включая вложенные подмодули, при условии, что подмодули репозитория имеют свои подмодули.

```
$ git clone --recurse-submodules https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
```

```
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Если вы уже клонировали проект, но забыли указать `--recurse-submodules`, вы можете объединить шаги `git submodule init` и `git submodule update`, запустив `git submodule update --init`. Также для инициализации, получения и извлечения всех существующих вложенных подмодулей можно использовать безопасную команду `git submodule update --init --recursive`.

Работа над проектом с подмодулями

Теперь, когда у нас есть копия проекта с подмодулями, мы будем работать совместно с нашими коллегами над обоими проектами: основным проектом и проектом подмодуля.

Получение изменений подмодуля из удалённого репозитория

Простейший вариант использования подмодулей в проекте состоит в том, что вы просто получаете сам подпроект и хотите периодически получать обновления, при этом в своей копии проекта ничего не изменяете. Давайте рассмотрим этот простой пример.

Если вы хотите проверить наличие изменений в подмодуле, перейдите в его каталог и выполните `git fetch`, а затем `git merge` для обновления локальной версии отслеживаемой ветки.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
  c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
```

```
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Теперь если вы вернётесь в основной проект и выполните `git diff --submodule`, то увидите, что подмодуль обновился, а также список новых коммитов. Если вы не хотите каждый раз при вызове команды `git diff` указывать опцию --submodule, то задайте конфигурационному параметру `diff.submodule` значение «log» для использования такого формата вывода по умолчанию.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Если сейчас вы создадите коммит, то состояние подмодуля будет зафиксировано с учётом последних изменений, что позволит другим людям их получить.

Если вы не хотите вручную извлекать и сливать изменения в подкаталог, то для вас существует более простой способ сделать то же самое. Если вы выполните `git submodule update --remote`, то Git сам перейдёт в ваши подмодули, получит изменения и обновит их для вас.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master     -> origin/master
Submodule path 'DbConnector': checked out
'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Эта команда по умолчанию предполагает, что вы хотите обновить локальную копию до состояния ветки master из репозитория подмодуля. Однако, при желании вы можете задать другую ветку. Например, если вы хотите, чтобы подмодуль DbConnector отслеживал ветку stable, то можете указать это либо в файле .gitmodules (тогда и другие люди также будут отслеживать эту ветку), либо в вашем локальном файле .git/config. Давайте настроим это в файле .gitmodules:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable
```

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
  27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

Если опустить -f .gitmodules, то команда сделает изменения локальном файле .git/config, но, вероятно, имеет смысл всё же отслеживать эту информацию в репозитории, так чтобы и все остальные участники имели к ней доступ.

Если сейчас выполнить команду git status, то Git покажет нам наличие «новых коммитов» в подмодуле.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
  modified:   .gitmodules
  modified:   DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

Если установить в настройках параметр status.submodulesummary, то Git будет также отображать краткое резюме об изменениях в ваших подмодулях:

```
$ git config status.submodulesummary 1
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

Если сейчас выполнить git diff, то увидите изменённый файл .gitmodules, а также наличие нескольких недавно полученных коммитов, которые готовы к включению в проект нашего подмодуля.

```
$ git diff
```

```
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
 Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

Здорово, что мы можем увидеть список подготовленных коммитов в нашем подмодуле. После создания коммита вы тоже можете получить эту информацию, выполнив команду git log -p.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
```

```
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
       path = DbConnector
       url = https://github.com/chaconinc/DbConnector
+      branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

При выполнении команды `git submodule update --remote` Git по умолчанию будет пытаться обновить **все** подмодули. Поэтому, при большом количестве подмодулей вы возможно захотите указать название только того подмодуля, который необходимо обновить.

Получение изменений основного проекта из удалённого репозитория

Теперь давайте поставим себя на место нашего коллеги, у которого есть собственная копия репозитория MainProject. Простого запуска команды `git pull` для получения последних изменений уже недостаточно:

```
$ git pull
From https://github.com/chaconinc/MainProject
   fb9093c..0a24cfc  master     -> origin/master
Fetching submodule DbConnector
From https://github.com/chaconinc/DbConnector
   c3f01dc..c87d55d  stable     -> origin/stable
Updating fb9093c..0a24cfc
Fast-forward
 .gitmodules        | 2 +-
 DbConnector        | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)
```

```
$ git status
 On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

 modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c87d55d...c3f01dc (4):
  < catch non-null terminated lines
  < more robust error handling
  < more efficient db routine
  < better connection routine

no changes added to commit (use "git add" and/or "git commit -a")
```

По умолчанию, команда git pull рекурсивно получает изменения для подмодулей, что можно увидеть выше в выводе первой команды. При этом она не **обновляет** подмодули. Это отображается в выводе команды git status, которая показывает, что подмодуль изменён (modified) и содержит новые коммиты (new commits). Более того, новые коммиты обозначены символом «<», означающим, что эти коммиты добавлены в проект MainProject, но не извлечены в текущем состоянии подмодуля DbConnector. Для завершения обновления следует выполнить команду git submodule update:

```
$ git submodule update --init --recursive
Submodule path 'vendor/plugins/demo': checked out '48679c6302815f6c76f1fe30625d795d9e55fc56'
```

```
$ git status
 On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

На всякий случай, команду git submodule update лучше запускать с параметром --init, чтобы проинициализировать новые подмодули, которые, возможно, были добавлены в новых коммитах MainProject репозитория, а также с параметром --recursive, чтобы обновить вложенные подмодули при их наличии.

Если вы хотите автоматизировать этот процесс, то добавьте параметр --recurse-submodules при запуске команды git pull (начиная с версии 2.14). Это приведёт к выполнению команды git submodule update сразу после получения и объединения изменений, приводя подмодули в корректное состояние. Более того, если вы хотите, чтобы Git всегда добавлял параметр --recurse-submodules к команде git pull, можно установить конфигурационный параметр submodule.recurse в значение true (работает для команды git pull начиная с версии 2.15). Если этот параметр конфигурации установлен, Git будет всегда добавлять --recurse-submodules ко всем командам, которые его поддерживают (кроме clone).

Существует специфическая ситуация, которая может возникнуть при получении изменений основного проекта: в одном из коммитов изменён URL подмодуля в файле .gitmodules. Такое может произойти, например, когда проект подмодуля переехал на другой хостинг. В таком случае команды git pull --recurse-submodules или git submodule update могут завершиться с ошибкой, потому как основной проект ссылается на коммит подмодуля, который не найден в удалённом репозитории этого подмодуля, на который указывает локальная конфигурация в основном репозитории. Чтобы исправить эту ситуацию требуется запуск команды git submodule sync:

```
# copy the new URL to your local config
$ git submodule sync --recursive
# update the submodule from the new URL
$ git submodule update --init --recursive
```

Работа с подмодулем

Весьма вероятно, что вы используете подмодули, потому что хотите работать над кодом подмодуля (или нескольких подмодулей) во время работы над кодом основного проекта. Иначе бы вы, скорее всего, предпочли использовать более простую систему управления зависимостями (такую как Maven или Rubygems).

Давайте теперь рассмотрим пример, в котором мы одновременно с изменениями в основном проекте внесём изменения в подмодуль, зафиксировав и опубликовав все эти изменения в одно и то же время.

Ранее, когда мы выполняли команду `git submodule update` для извлечения изменений из репозитория подмодуля, Git получал изменения и обновлял файлы в поддиректории, но оставлял вложенный репозиторий в состоянии, называемом «отсоединённый HEAD» («detached HEAD»). Это значит, что локальная рабочая ветка (такая, например, как master), отслеживающая изменения, отсутствует. Отсутствие отслеживающей ветки означает, что даже если вы зафиксируете изменения в подмодуле, эти изменения, скорее всего, будут потеряны при следующем запуске `git submodule update`. Если хотите, чтобы изменения в подмодуле отслеживались, потребуется выполнить несколько дополнительных шагов.

Для подготовки репозитория подмодуля к такой работе с ним, нужно сделать две вещи. Нужно перейти в каждый подмодуль и извлечь ветку, в которой будете далее работать. Затем необходимо сообщить Git что ему делать, если вы внесли свои изменения, а затем получаете новые изменения из удалённого репозитория командой `git submodule update --remote`. Возможны два варианта — вы можете слить их в вашу локальную версию или попробовать перебазировать ваши локальные наработки поверх новых изменений.

Первым делом, давайте перейдём в каталог нашего подмодуля и переключимся на нужную ветку.

```
$ cd DbConnector/

$ git checkout stable

Switched to branch 'stable'
```

Давайте попробуем обновить подмодуль путём слияния изменений. Чтобы задать её вручную, просто добавьте опцию `--merge` при вызове команды `update`. Ниже мы видим, что с удалённого сервера получен коммит для подмодуля и слит с текущим состоянием:

```
$ cd ..
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Перейдя в каталог DbConnector, можно увидеть, что новые изменения уже слиты в нашу локальную ветку stable. Теперь давайте посмотрим что происходит, когда мы вносим локальные изменения в библиотеку, а кто-то другой в то же время отправляет другие изменения в удалённый репозиторий.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'Unicode support'
[stable f906e16] Unicode support
 1 file changed, 1 insertion(+)
```

Теперь мы можем увидеть, что происходит при обновлении подмодуля, когда локальные изменения должны включать новые изменения, полученные с удалённого сервера.

```
$ cd ..
$ git submodule update --remote --rebase
```

```
First, rewinding head to replay your work on top of it...
Applying: Unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Если вы забудете указать параметры `--rebase` или `--merge`, то Git просто обновит подмодуль до состояния репозитория на сервере и установит его в состояние отсоединённого HEAD.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Если такое происходит — не беспокойтесь, вы всегда можете перейти в каталог подмодуля, переключиться на вашу ветку (которая всё ещё будет содержать ваши наработки) и вручную слить ветку `origin/stable` или перебазировать свои изменения относительно неё (или любую другую удалённую ветку).

Если вы не зафиксировали ваши изменения в подмодуле и выполнили его обновление, то это приведёт к проблемам — Git получит изменения из удалённого репозитория, но не перезапишет несохранённые изменения в каталоге вашего подмодуля.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
 scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
```

```
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

Если ваши локальные изменения в подмодуле конфликтуют с какими-либо изменениями в удалённом репозитории, то Git вам сообщит об этом при обновлении подмодуля.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

Для разрешения конфликта вы можете перейти в каталог подмодуля и сделать это как вы бы делали в обычной ситуации.

Публикация изменений подмодуля

Теперь у нас есть некоторые изменения в каталоге нашего подмодуля. Какие-то из них мы получили при обновлении из удалённого репозитория, а другие были сделаны локально и пока никому не доступны, так как мы их ещё никуда не отправили.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > Update setup script
  > Unicode support
  > Remove unnecessary method
  > Add new option for conn pooling
```

Если мы создадим коммит в основном проекте и отправим его на сервер, не отправив при этом изменения в подмодуле, то другие люди, которые попытаются использовать наши изменения, столкнутся с проблемой, так как у них не будет возможности получить требуемые изменения подмодуля. Эти изменения будут присутствовать только в нашей локальной копии.

Чтобы гарантированно избежать такой проблемы, вы можете попросить Git проверять отправлены ли изменения всех подмодулей до

отправки изменений основного репозитория на сервер. Команда git push принимает аргумент --recurse-submodules, который может принимать значения либо «check», либо «on-demand». Использование значения «check» приведёт к тому, что push просто завершится ошибкой, если какой-то из подмодулей не был отправлен на сервер.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

 git push --recurse-submodules=on-demand

or cd to the path and use

 git push

to push them to a remote.
```

Как видите, эта команда также даёт нам некоторые полезные советы о том, что мы могли бы делать дальше. Самый простой вариант — это пройти по всем подмодулям и вручную отправить изменения на удалённые серверы, чтобы гарантировать доступность изменений другим людям, а затем попытать отправить изменения снова. Если вы хотите использовать поведение “check” при каждом выполнении git push, то его можно установить поведением по умолчанию, выполнив команду git config push.recurseSubmodules check.

Другой вариант — это использовать значение «on-demand», тогда Git попытается сделать всё вышеописанное за вас.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
```

```
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

Как видите, перед отправкой на сервер основного проекта Git перешел в каталог модуля DbConnector и отправил его изменения на сервер. Отправка изменений основного репозитория также завершится неудачей если во время отправки изменений подмодуля возникла ошибка. Установить такое поведение по умолчанию можно с помощью команды `git config push.recurseSubmodules` on-demand.

Слияние изменений подмодуля

Если вы измените ссылку на подмодуль одновременно с кем-то ещё, то вы можете столкнуться с некоторыми проблемами. Такое случается если истории подмодуля разошлись, и они зафиксированы в разошедшихся ветках основного проекта, что может потребовать некоторых усилий для исправления.

Если один из коммитов является прямым предком другого (слияние может быть выполнено перемоткой вперёд), то Git просто выберет последний для выполнения слияния, то есть все отработает хорошо.

Однако, Git не будет пытаться выполнить даже простейшего слияния. Если коммиты подмодуля разошлись и слияние необходимо, вы получите нечто подобное:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
  9a377d1..eb974f8  master     -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Здесь говорится о том, что Git понял, что в этих двух ветках содержатся указатели на разошедшиеся записи в истории подмодуля и их необходимо слить. Git поясняет это как «слияние последующих коммитов не найдено» («merge following commits not found»), что несколько обескураживает, но вскоре мы объясним почему так происходит.

Для решения этой проблемы мы должны разобраться в каком состоянии должен находиться подмодуль. Странно, но Git не предоставляет вам для этого никакой вспомогательной информации, даже SHA-1 хешей коммитов разошедшихся веток истории. К счастью, получить эту информации довольно просто. Если выполнить команду `git diff`, то вы получите SHA-1 хеши коммитов обоих веток, которые вы пытаетесь слить.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

В данном примере `eb41d76` является **нашим** коммитом в подмодуле, а `c771610` — коммитом из вышестоящего репозитория. Если мы

перейдём в каталог нашего подмодуля, то он должен быть на коммите `eb41d76`, так как операция слияния его не изменяла. Если по каким-то причинам это не так, то вы можете просто создать новую ветку из этого коммита и извлечь её.

Куда более важным является SHA-1 хеш коммита второй ветки истории. Именно его мы должны будем слить и разрешить конфликты. Вы можете попытаться либо просто выполнить слияние, указав непосредственно этот SHA-1 хеш, либо создать из него отдельную ветку и слить её. Мы предлагаем использовать последний вариант хотя бы только потому, что сообщение коммита слияния получается более читаемым.

Итак, перейдите в каталог нашего подмодуля, создайте ветку «try-merge» на основе второго SHA-1 хеша из `git diff` и выполните слияние вручную.

```
$ cd DbConnector
```

```
$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135
```

```
$ git branch try-merge c771610
```

```
$ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

Мы получили конфликт слияния, поэтому если мы разрешим его и создадим коммит, то можно просто включить результирующий коммит в основной проект.

```
$ vim src/main.c (1)
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
```

```
[master 9fd905e] merged our changes
```

```
$ cd .. (2)
$ git diff (3)
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
 -Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector (4)
```

```
$ git commit -m "Merge Tom's Changes" (5)
[master 10d2c60] Merge Tom's Changes
```

1. Во-первых, мы разрешили конфликт.
2. Затем мы вернулись в каталог основного проекта.
3. Мы снова проверили SHA-1 хеши.
4. Обновили указатель на подмодуль с учётом разрешенного конфликта.
5. Зафиксировали наше слияние, создав новый коммит.

Это может немного запутать, но на самом деле здесь нет ничего сложного.

Интересно, что существует ещё один случай, с которым Git умеет работать. Если подмодуль содержит некий коммит слияния, который в своей истории содержит **оба** первоначальных коммита, то Git предложит его как возможное решение. Git видит, что в проекте подмодуля ветки, содержащие эти два коммита, уже когда-то кем-то сливались и этот коммит слияния, вероятно, именно то, что вам нужно.

Именно поэтому сообщение об ошибке выше было «merge following commits not found» — Git не мог найти такой коммит. Кто бы мог подумать, что Git **пытается** такое сделать?

Если Git нашёл только один приемлемый коммит слияния, то вы увидите нечто подобное:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Предлагаемая команда обновляет индекс таким же образом, как если бы вы выполнили git add (что очищает список конфликтов), а затем создали коммит. Хотя, наверное, вам не стоит так делать. Будет лучше просто перейти в каталог подмодуля, просмотреть изменения, сместить вашу ветку на этот коммит, должным образом всё протестировать и только потом создать коммит в основном репозитории.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward
```

```
$ cd ..

$ git add DbConnector

$ git commit -am 'Fast forward to a common submodule child'
```

Это приводит к аналогичному результату, но таким способом вы получаете рабочий проверенный код в каталоге подмодуля.

Полезные советы

Существует несколько хитростей, которые могут немного упростить вашу работу с подмодулями.

Команда Foreach

Существует команда foreach, которая позволяет выполнить произвольную команду в каждом подмодуле. Это может быть, действительно, полезным если у вас в одном проекте присутствует большое количество подмодулей.

Для примера допустим, что мы хотим начать работу над какой-то новой функциональностью или исправить какую-то ошибку и наша работа затронет сразу несколько подмодулей. Мы можем легко припрятать все наработки во всех наших подмодулях.

```
$ git submodule foreach 'git stash'

Entering 'CryptoLibrary'

No local changes to save

Entering 'DbConnector'

Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable

HEAD is now at 82d2ad3 Merge from origin/stable
```

Затем мы можем создать новую ветку и переключиться на неё во всех наших подмодулях.

```
$ git submodule foreach 'git checkout -b featureA'

Entering 'CryptoLibrary'

Switched to a new branch 'featureA'

Entering 'DbConnector'

Switched to a new branch 'featureA'
```

Вы уловили идею. Ещё одна полезная вещь, которую можно сделать с помощью этой команды — создать комплексную дельту изменений основного проекта и всех подмодулей.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

     commit_pager_choice();

+    url = url_decode(url_orig);
+
     /* build alias_argv */
     alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
     alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
      return url_decode_internal(&url, len, NULL, &out, 0);
 }
```

```
+char *url_decode(const char *url)
+{
+      return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
       struct strbuf out = STRBUF_INIT;
```

Здесь видно, что мы определили в подмодуле функцию и вызываем её в основном проекте. Это, конечно, упрощённый пример, но надеемся, что мы смогли донести до вас всю полезность этой функции.

Полезные псевдонимы

Возможно, вы захотите настроить псевдонимы для некоторых из этих команд, так как они могут быть довольно длинными и для большинства из них вы не можете задать значения по умолчанию. Мы рассмотрели настройку псевдонимов Git в разделе Псевдонимы в Git главы 2, однако ниже приведен пример, который вы наверняка захотите использовать, если планируете часто работать с подмодулями Git.

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

Таким образом, для обновления подмодулей достаточно выполнить команду `git supdate`, а для отправки всех изменений с учётом подмодулей — команду `git spush`.

Проблемы с подмодулями

Однако, использование подмодулей не обходится без проблем.

Переключение веток

Например, переключение веток с подмодулями в них может оказаться довольно запутанным, особенно для версий Git старше 2.13. Если вы создадите новую ветку и добавите в ней подмодуль, а затем переключитесь обратно на ветку без подмодуля, то каталог подмодуля всё равно останется и будет неотслеживаемым:

```
$ git --version
```

```
git version 2.12.2


$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'
```

```
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...
```

```
$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary
```

```
$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

 CryptoLibrary/
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Удалить каталог не сложно, но может показаться странным, что он вообще присутствует. Если удалить каталог и снова переключиться на ветку с подмодулем, то потребуется выполнить `submodule update --init`, чтобы в нём появились файлы.

```
$ git clean -ffdx
Removing CryptoLibrary/
$ git checkout add-crypto
Switched to branch 'add-crypto'
```

```
$ ls CryptoLibrary/
```

```
$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'
```

```
$ ls CryptoLibrary/
Makefile includes scripts src
```

Опять же, это не сильно сложно, но может немного сбивать с толку.

В более новых версиях (Git >= 2.13) всё несколько упрощено за счёт поддержки командой `git checkout` параметра `--recurse-submodules`, который отвечает за приведение подмодулей в состояние, соответствующее извлекаемой ветке.

```
$ git --version
git version 2.13.3
```

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'
```

```
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...
```

```
$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary
```

```
$ git checkout --recurse-submodules master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.


nothing to commit, working tree clean
```

Использование параметра --recurse-submodules команды git checkout помогает при работе с несколькими ветками основного проекта, когда в каждой из них используется разное состояние подмодуля. Действительно, если вы переключитесь между ветками, в которых зафиксированы разные состояния подмодуля, то команда git status отметит подмодуль как изменённый («modified») и покажет наличие новых коммитов в нём. Это происходит из-за того, что по умолчанию состояние подмодуля не изменяется при переключении веток.

Такое поведение может сильно сбивать с толку, поэтому рекомендуется всегда использовать git checkout --recurse-submodules если в вашем проекте есть подмодули. В более старых версиях Git, в которых команда git checkout не поддерживает параметр --recurse-submodules, для

приведения подмодулей в ожидаемое состояние используйте команду git submodule update --init --recursive сразу после извлечения ветки.

К счастью, вы можете указать Git (начиная с версии 2.14) всегда использовать параметр --recurse-submodules с помощью изменения конфигурации: git config submodule.recurse true. Как упоминалось выше, это заставит Git автоматически добавлять параметр --recurse-submodules ко всем командам (кроме git clone), которые его поддерживают, и рекурсивно обрабатывать подмодули.

Переход от подкаталогов к подмодулям

Другая большая проблема возникает при переходе от использования подкаталогов к использованию подмодулей. Если у вас были отслеживаемые файлы в вашем проекте, и вы хотите переместить их в подмодуль, то вы должны быть осторожны, иначе Git будет ругаться на вас. Предположим, у вас есть файлы в каком-то каталоге вашего проекта, и вы хотите переместить их в подмодуль. Если вы удалите подкаталог, а затем выполните submodule add, то Git ругнётся на вас:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Первым делом вы должны удалить каталог CryptoLibrary из индекса. Затем можно добавить подмодуль:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Предположим, что вы сделали это в какой-то ветке. Если вы попробуете переключиться обратно на ветку, в которой эти файлы всё ещё являются частью основного проекта, то вы получите ошибку:

```
$ git checkout master
```

```
error: The following untracked working tree files would be overwritten by checkout:
 CryptoLibrary/Makefile
 CryptoLibrary/includes/crypto.h
 ...
Please move or remove them before you can switch branches.
Aborting
```

Вы всё же можете извлечь ветку принудительно, используя команду checkout -f, только убедитесь перед этим в отсутствии несохранённых изменений, так как они могут быть перезаписаны.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Если вернуться обратно на предыдущую ветку, то по какой-то причине каталог CryptoLibrary будет пустым, а команда git submodule update не сможет это исправить. Чтобы вернуть свои файлы, вам понадобится перейти в каталог подмодуля и выполнить команду git checkout .. Можно использовать эту команду вместе с submodule foreach для извлечения сразу всех подмодулей.

Важно отметить, что в современных версиях Git подмодули сохраняют все служебные данные в каталоге .git основного проекта, поэтому, в отличие от более старых версий Git, удаление каталога подмодуля не приведёт к потере каких-либо коммитов или веток.

Все эти инструменты делают подмодули довольно простым и эффективным методом работы одновременно над несколькими тесно связанными, но разделёнными проектами.

Создание пакетов

Помимо рассмотренных ранее основных способов передачи данных Git по сети (HTTP, SSH и т. п.), существует ещё один способ, который обычно не используется, но в некоторых случаях может быть весьма полезным.

Git умеет «упаковывать» свои данные в один файл. Это может быть полезным в разных ситуациях. Может быть, ваша сеть не работает, а вы хотите отправить изменения своим коллегам. Возможно, вы работаете откуда-то извне офиса и не имеете доступа к локальной сети по

соображениям безопасности. Может быть, ваша карта беспроводной/проводной связи просто сломалась. Возможно, у вас в данный момент нет доступа к общему серверу, а вы хотите отправить кому-нибудь по электронной почте обновления, но передавать 40 коммитов с помощью format-patch не хотите.

В этих случаях вам может помочь команда git bundle. Она упакует всё, что в обычной ситуации было бы отправлено по сети командой git push, в бинарный файл, который вы можете передать кому-нибудь по электронной почте или поместить на флешку и затем распаковать в другом репозитории.

Рассмотрим простой пример. Допустим, у вас есть репозиторий с двумя коммитами:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    Second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    First commit
```

Если вы хотите отправить кому-нибудь этот репозиторий, но не имеете доступа на запись к общей копии репозитория или просто не хотите его настраивать, то вы можете упаковать его командой git bundle create.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
```

```
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

В результате вы получили файл repo.bundle, в котором содержатся все данные, необходимые для воссоздания ветки master репозитория. Команде bundle необходимо передать список или диапазон коммитов, которые вы хотите добавить в пакет. Если вы намереваетесь использовать пакет для того, чтобы клонировать репозиторий где-нибудь ещё, вы должны добавить в этот список HEAD, как это сделали мы.

Вы можете отправить файл repo.bundle кому-нибудь по электронной почте или скопировать его на USB-диск, тем самым легко решив исходную проблему.

С другой стороны, допустим, вы получили файл repo.bundle и хотите поработать над этим проектом. Вы можете клонировать репозиторий из бинарного файла в каталог, почти также как вы делаете это при использовании URL.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 Second commit
b1ec324 First commit
```

Если при создании пакета вы не указали в списке ссылок HEAD, то при распаковке вам потребуется указать -b master или какую-либо другую ветку, включённую в пакет, иначе Git не будет знать, на какую ветку ему следует переключиться.

Теперь предположим, что вы сделали три коммита и хотите отправить их обратно в виде пакета на USB-флешке или по электронной почте.

```
$ git log --oneline
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
```

```
9a466c5 Second commit

b1ec324 First commit
```

Во-первых, нам нужно определить диапазон коммитов, которые мы хотим включить в пакет. В отличие от сетевых протоколов, которые сами выясняют минимальный набор данных, который нужно передать по сети, в данном случае мы должны сделать это сами вручную. В данном примере вы можете сделать, как раньше и упаковать полностью весь репозиторий, но будет лучше упаковать только изменения — три коммита, сделанные локально.

Для того, чтобы сделать это, вы должны вычислить различия. Как мы рассказывали в Диапазоны коммитов, вы можете указать диапазон коммитов несколькими способами. Для того, чтобы получить три коммита из нашей основной ветки, которые отсутствовали в изначально клонированной ветке, мы можем использовать запись вида origin/master..master или master ^origin/master. Вы можете проверить ее с помощью команды log.

```
$ git log --oneline master ^origin/master

71b84da Last commit - second repo

c99cf5b Fourth commit - second repo

7011d3d Third commit - second repo
```

Так что теперь у нас есть список коммитов, которые мы хотим включить в пакет, давайте упакуем их. Сделаем мы это с помощью команды git bundle create, указав имя выходного пакета и диапазон коммитов, которые мы хотим включить в него.

```
$ git bundle create commits.bundle master ^9a466c5

Counting objects: 11, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (9/9), 775 bytes, done.

Total 9 (delta 0), reused 0 (delta 0)
```

В результате в нашем каталоге появился файл commits.bundle. Если мы отправим его нашему коллеге, то он сможет импортировать пакет в

исходный репозиторий, даже если в репозитории была проделана некоторая работа параллельно с нашей.

При получении пакета коллега перед импортом его в свой репозиторий может проверить пакет, просмотрев его содержимое. Лучшей командой для этого является bundle verify, которая может проверить, что файл действительно является корректным Git-пакетом и что у вас есть все необходимые предки коммитов для правильного его восстановления.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

Если автор создал пакет только с последними двумя коммитами, которые он сделал, а не со всеми тремя, то исходный репозиторий не сможет импортировать этот пакет, так как у него отсутствует необходимая история. В таком случае команда verify вернёт нечто подобное:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 Third commit - second repo
```

Однако, наш первый пакет корректен, поэтому мы можем извлечь коммиты из него. На случай если вы захотите увидеть ветки пакета, которые могут быть импортированы, существует команда для отображения только списка веток:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Подкоманда verify также выводит список веток. Если цель состоит в том, чтобы увидеть, что может быть извлечено из пакета, то вы можете использовать команды fetch или pull для импорта коммитов. Ниже мы ветку 'master' из пакета извлекаем в ветку 'other-master' нашего репозитория:

```
$ git fetch ../commits.bundle master:other-master
```

```
From ../commits.bundle
 * [new branch]      master     -> other-master
```

Теперь мы можем увидеть, какие коммиты мы импортировали в ветку 'other-master' так же, как и любые коммиты, которые мы сделали в то же время в нашей собственной ветке 'master'.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) Third commit - first repo
| * 71b84da (other-master) Last commit - second repo
| * c99cf5b Fourth commit - second repo
| * 7011d3d Third commit - second repo
|/
* 9a466c5 Second commit
* b1ec324 First commit
```

Таким образом, команда git bundle может быть, действительно, полезной для организации совместной работы или для выполнения сетевых операций, когда у вас нет доступа к соответствующей сети или общему репозиторию.

Замена

Объекты в Git неизменяемы, но он предоставляет интересный способ эмулировать замену объектов в своей базе другими объектами.

Команда replace позволяет вам указать объект Git и сказать «каждый раз, когда встречается этот объект, заменяй его другим». В основном, это бывает полезно для замены одного коммита в вашей истории другим.

Например, допустим в вашем проекте огромная история изменений и вы хотите разбить ваш репозиторий на два — один с короткой историей для новых разработчиков, а другой с более длинной историей для людей, интересующихся анализом истории. Вы можете пересадить одну историю на другую, «заменяя» самый первый коммит в короткой истории последним коммитом в длинной истории. Это удобно, так как вам не придётся по-настоящему изменять каждый коммит в новой истории, как это вам бы потребовалось делать в случае обычного объединения историй (так как родословная коммитов влияет на SHA-1).

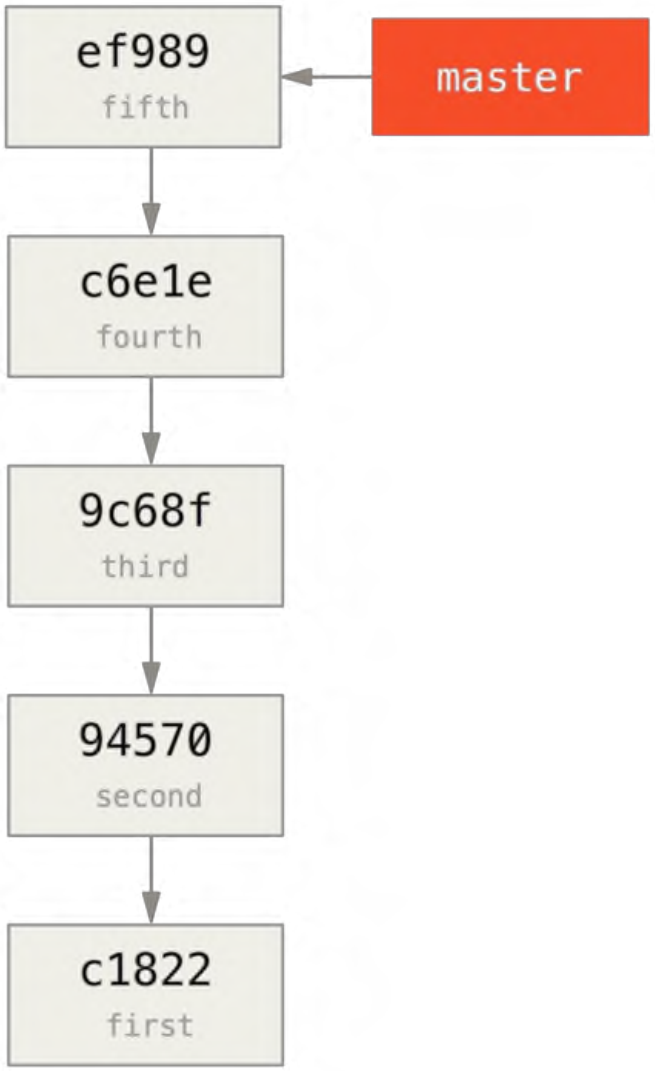
Давайте испробуем как это работает, возьмём существующий репозиторий и разобьём его на два — один со свежими правками, а другой

с историческими, и затем посмотрим, как мы можем воссоединить их с помощью операции replace, не изменяя при этом значений SHA-1 в свежем репозитории.

Мы будем использовать простой репозиторий с пятью коммитами:

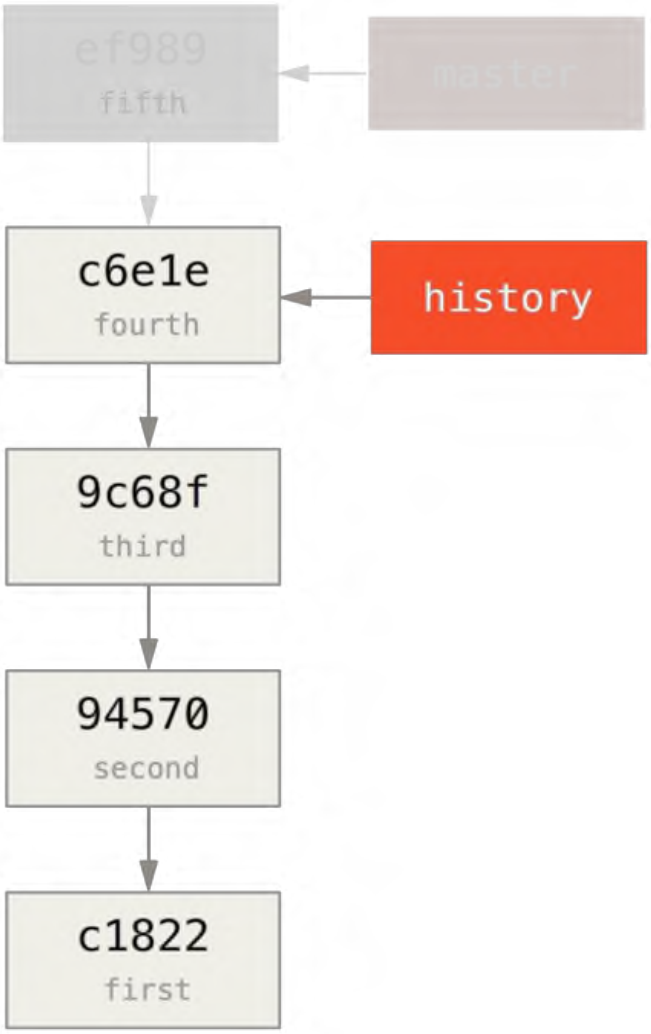
```
$ git log --oneline
ef989d8 Fifth commit
c6e1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Мы хотим разбить его на два семейства историй. Одно семейство, которое начинается от первого коммита и заканчивается четвёртым, будет историческим. Второе, состоящее пока только из четвёртого и пятого коммитов — будет семейством со свежей историей.



Создать историческое семейство легко, мы просто создаём ветку с вершиной на нужном коммите и затем отправляем эту ветку как `master` в новый удалённый репозиторий.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) Fifth commit
c6e1e95 (history) Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```



Теперь мы можем отправить только что созданную ветвь history в ветку master нашего нового репозитория:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
```

```
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master
```

Таким образом, наша история опубликована, а мы теперь займёмся более сложной частью — усечём свежую историю. Нам необходимо перекрытие, так чтобы мы смогли заменить коммит из одной части коммитом из другой, то есть мы будем обрезать историю, оставив четвёртый и пятый коммиты (таким образом четвёртый коммит будет входить в пересечение).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) Fifth commit
c6e1e95 (history) Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

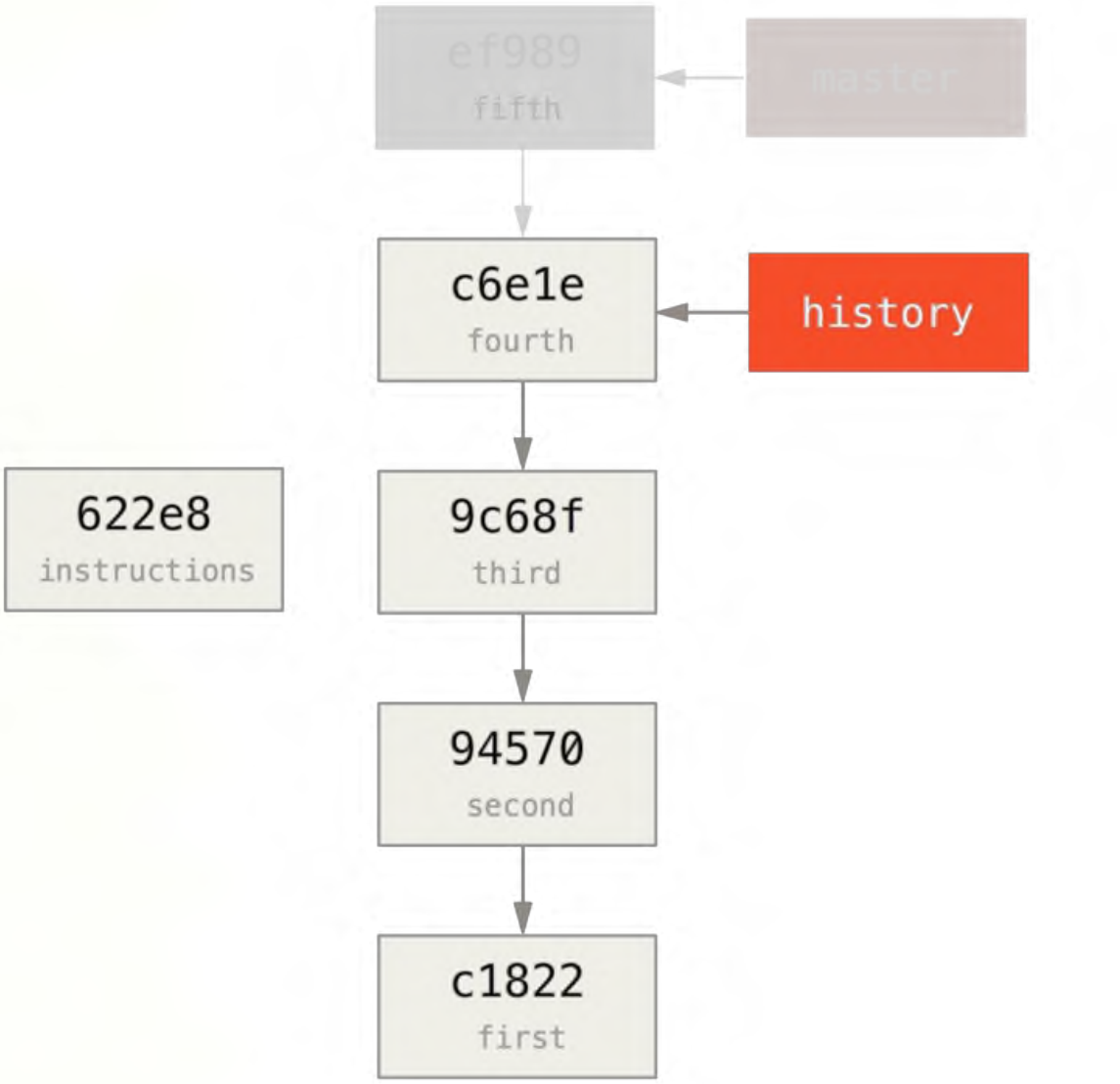
В данном случае будет полезным создать базовый коммит, содержащий инструкции о том, как раскрыть историю, так другие разработчики будут знать, что делать, если они столкнулись с первым коммитом урезанной истории и нуждаются в остальной истории. Итак, далее мы создадим объект заглавного коммита, представляющий нашу отправную точку с инструкциями, а затем перебазируем оставшиеся коммиты (четвёртый и пятый) на этот коммит.

Для того, чтобы сделать это, нам нужно выбрать точку разбиения, которой для нас будет третий коммит, хеш которого 9c68fdc. Таким образом, наш базовый коммит будет основываться на этом дереве. Мы можем создать наш базовый коммит, используя команду commit-tree, которая просто берет дерево и возвращает SHA-1 объекта, представляющего новый сиротский коммит.

```
$ echo 'Get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

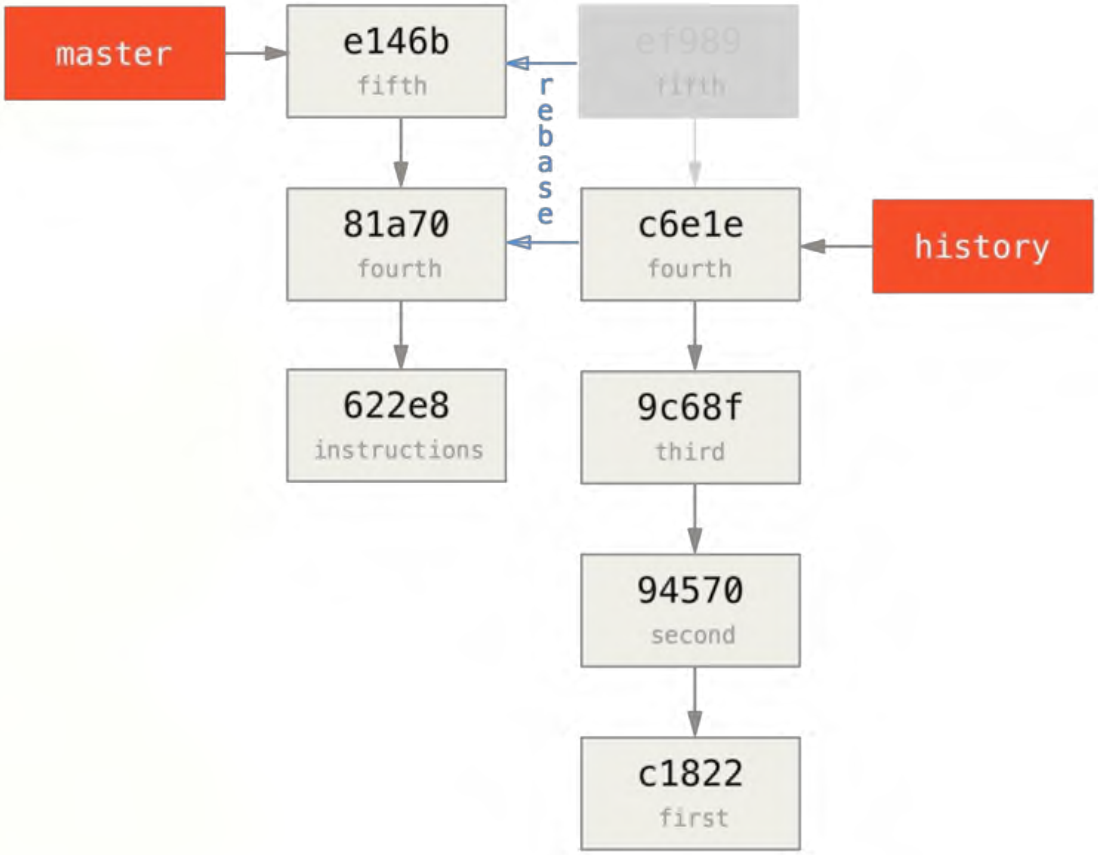
Примечание	Команда commit-tree входит в набор команд, которые обычно называются «сантехническими». Это команды, которые обычно не предназначены для непосредственного использования, но вместо этого

	используются **другими** командами Git для выполнения небольших задач. Периодически, когда мы занимаемся странными задачами подобными текущей, эти команды позволяют нам делать низкоуровневые вещи, но все они не предназначены для повседневного использования.



Хорошо. Теперь, когда у нас есть базовый коммит, мы можем перебазировать нашу оставшуюся историю на этот коммит используя git rebase --onto. Значением аргумента --onto будет SHA-1 хеш коммита, который мы только что получили от команды commit-tree, а перебазируемой точкой будет третий коммит (родитель первого коммита, который мы хотим сохранить, 9c68fdc):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



Таким образом, мы переписали нашу свежую историю поверх вспомогательного базового коммита, который теперь содержит инструкции о том, как при необходимости восстановить полную историю. Мы можем отправить эту историю в новый проект и теперь, когда люди клонируют его репозиторий, они будут видеть только два свежих коммита и после них базовый коммит с инструкциями.

Давайте представим себя на месте кого-то, кто впервые клонировал проект и хочет получить полную историю. Для получения исторических данных после клонирования, усечённого репозитория, ему нужно добавить в список удалённых репозиториев исторический репозиторий и извлечь из него данные:

```
$ git clone https://github.com/schacon/project
$ cd project
```

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
622e88e Get history from blah blah blah
```

```
$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch]      master     -> project-history/master
```

Теперь у этого пользователя его собственные свежие коммиты будут находиться в ветке master, а исторические коммиты в ветке project-history/master.

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
622e88e Get history from blah blah blah
```

```
$ git log --oneline project-history/master
c6e1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

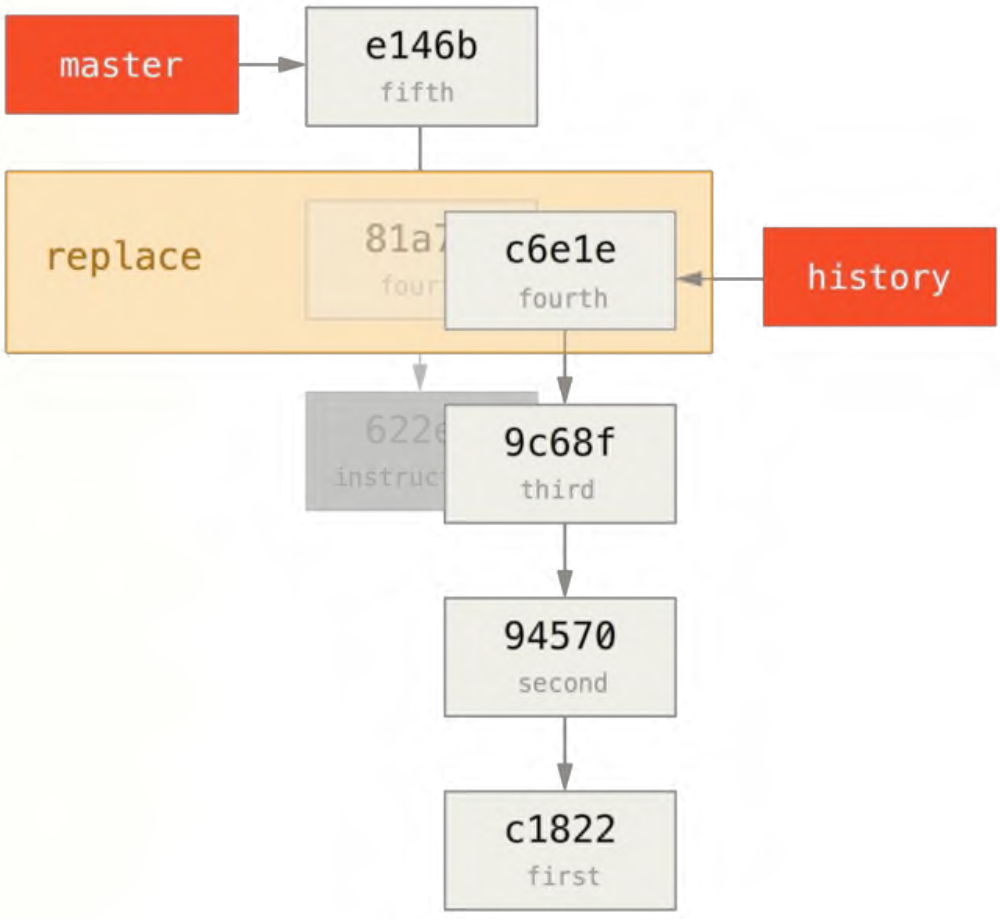
Для объединения этих веток вы можете просто вызывать git replace, указав коммит, который вы хотите заменить, и коммит, которым вы хотите заменить первый. Так мы хотим заменить «четвёртый» коммит в основной ветке «четвёртым» коммитом из ветки project-history/master:

```
$ git replace 81a708d c6e1e95
```

Если теперь вы посмотрите историю ветки master, то должны увидеть нечто подобное:

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Здорово, не правда ли? Не изменяя SHA-1 всех коммитов семейства, мы можем заменить один коммит в нашей истории совершенно другим коммитом и все обычные утилиты (bisect, blame и т. д.) будут работать как от них это и ожидается.



Интересно, что для четвёртого коммита SHA-1 хеш выводится равный 81a708d, хотя в действительности он содержит данные коммита

c6e1e95, которым мы его заменили. Даже если вы выполните команду типа cat-file, она отобразит заменённые данные:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Помните, что настоящим родителем коммита 81a708d был наш вспомогательный базовый коммит (622e88e), а не 9c68fdce как это отмечено здесь.

Другое интересное замечание состоит в том, что информация о произведённой замене сохранена у нас в ссылках:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

Следовательно, можно легко поделиться заменами — для этого мы можем отправить их на наш сервер, а другие люди могут легко скачать их оттуда. Это не будет полезным в случае, если вы используете replace для пересадки истории (так как в этом случае все люди будут скачивать обе истории, тогда зачем мы разделяли их?), но это может быть полезным в других ситуациях.

Хранилище учётных данных

Если для подключения к удалённым серверам вы используете протокол SSH, то вы можете использовать ключ вместо пароля, что позволит вам безопасно передавать данные без ввода логина и пароля. Однако, это невозможно при использовании HTTP-протоколов — каждое подключение требует пары логин, пароль. Всё ещё сложнее для систем с двухфакторной аутентификацией, когда выражение, которое вы используете в качестве пароля, генерируется случайно и его сложно воспроизвести.

К счастью, в Git есть система управления учётными данными, которая может помочь в этом. В Git «из коробки» есть несколько опций:

- По умолчанию Git не кеширует учётные данные совсем. Каждое подключение будет запрашивать у вас логин и пароль.
- В режиме «cache» учётные данные сохраняются в памяти в течение определённого периода времени. Ни один из паролей никогда не сохраняется на диск и все они удаляются из кеша через 15 минут.
- В режиме «store» учётные данные сохраняются на неограниченное время в открытом виде в файле на диске. Это значит, что, до тех пор пока вы не измените пароль к Git-серверу, вам не потребуется больше вводить ваши учётные данные. Недостатком такого подхода является то, что ваш пароль хранится в открытом виде в файле в вашем домашнем каталоге.
- На случай если вы используете Mac, в Git есть режим «osxkeychain», при использовании которого учётные данные хранятся в защищённом хранилище, привязанному к вашему системному аккаунту. В этом режиме учётные данные сохраняются на диск на неограниченное время, но они шифруются с использованием той же системы, с помощью которой сохраняются HTTPS-сертификаты и автозаполнения для Safari.
- В случае если вы используете Windows, вы можете установить помощник, называемый «Git Credential Manager for Windows». Он похож на «osxkeychain», описанный выше, но для управления секретной информацией использует Windows Credential Store. Найти его можно по ссылке https://github.com/Microsoft/Git-Credential-Manager-for-Windows.

Вы можете выбрать один из этих методов, изменив настройки Git:

```
$ git config --global credential.helper cache
```

Некоторые из этих помощников имеют опции. Помощник «store» может принимать аргумент --file <path>, который определяет, где будет храниться файл с открытыми учётными данный (по умолчанию

используется ~/.git-credentials). Помощник «cache» принимает опцию --timeout <seconds>, которая изменяет промежуток времени, в течение которого демон остаётся запущенным (по умолчанию «900», или 15 минут). Ниже приведён пример как вы можете настроить помощник «store» на использование определённого файла:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git позволяет настраивать сразу несколько помощников. При поиске учётных данных для конкретного сервера, Git будет по порядку запрашивать у них учётные данные и остановится при получении первого ответа. При сохранении учётных данных Git отправит их **всем** помощникам в списке, которые уже в свою очередь могут решить, что с этими данными делать. Ниже приведено как будет выглядеть .gitconfig, если у вас есть файл с учётными данными на флэш-диске, но, на случай его отсутствия, вы хотите дополнительно использовать кеширование в оперативной памяти.

```
[credential]
   helper = store --file /mnt/thumbdrive/.git-credentials
   helper = cache --timeout 30000
```

Под капотом

Как же это всё работает? Корневой командой Git для системы помощников авторизации является git credential, которая принимает команду через аргумент, а все остальные входные данные через стандартный поток ввода.

Возможно, это проще понять на примере. Допустим, помощник авторизации был настроен и в нем сохранены учётные данные для mygithost. Ниже приведена рабочая сессия, в которой используется команда «fill», вызываемая Git при попытке найти учётные данные для сервера:

```
$ git credential fill (1)
protocol=https (2)
host=mygithost
(3)
protocol=https (4)
host=mygithost
```

```
username=bob
password=s3cre7
$ git credential fill (5)
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

1. Это команда, которая начинает взаимодействие.
2. После этого Git-credential ожидает данные из стандартного потока ввода. Мы передаём ему то, что знаем: протокол и имя сервера.
3. Пустая строка обозначает, что ввод закончен и система управления учётными данными должна ответить, что ей известно.
4. После этого Git-credential выполняет какую-то работу и выводит обнаруженную информацию.
5. Если учётные данные не найдены, Git спрашивает у пользователя логин/пароль, и выводит их обратно в задействованный поток вывода (в данном примере это одна и та же консоль).

В действительности система управления учётными данными вызывает программы, отделённые от самого Git; какие и как зависит в том числе и от настроек `credential.helper`. Существует несколько вариантов вызова:

Настройки	**Поведение**
`foo`	Выполняется git-credential-foo
`foo -a --opt=bcd`	Выполняется git-credential-foo -a --opt=bcd

Настройки	Поведение
/absolute/path/foo -xyz	Выполняется /absolute/path/foo -xyz
!f() { echo "password=s3cre7"; }; f	Код после символа ! выполняется в шелле

Итак, помощники, описанные выше на самом деле называются git-credential-cache, git-credential-store и т. д. и мы можем настроить их на приём аргументов командной строки. Общая форма для этого git-credential-foo [args] <action>. Протокол ввода/вывода такой же, как и у git-credential, но они используют немного другой набор операций:

- get запрос логина и пароля.
- store запрос на сохранение учётных данных в памяти помощника.
- erase удаляет учётные данные для заданных параметров из памяти используемого помощника.

Для операций store и erase не требуется ответа (в любом случае Git его игнорирует). Однако, для Git очень важно, что помощник ответит на операцию get. Если помощник не знает что-либо полезного, он может просто завершить работу не выводя ничего, но, если знает — он должен добавить к введённой информации имеющуюся у него информацию. Вывод обрабатывается как набор операций присваивания; выведенные значения заменят те, что Git знал до этого.

Ниже приведён пример, используемый ранее, но вместо git-credential напрямую вызывается git-credential-store:

```
$ git credential-store --file ~/git.store store (1)
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get (2)
protocol=https
host=mygithost
```

```
username=bob (3)
password=s3cre7
```

1. Здесь мы просим git-credential-store сохранить некоторые учётные данные: логин «bob» и пароль «s3cre7», которые будут использоваться при доступе к https://mygithost.
2. Теперь мы извлечём эти учётные данные. Мы передаём часть уже известных нам параметров подключения (https://mygithost) и пустую строку.
3. git-credential-store возвращает логин и пароль, которые мы сохранили ранее.

Ниже приведено содержимое файла ~/git.store:

```
https://bob:s3cre7@mygithost
```

Это просто набор строк, каждая из которых содержит URL, включающий в себя учётные данные. Помощники osxkeychain и wincred используют форматы, лежащие в основе их хранилищ, а cache использует его собственный формат хранения во внутренней памяти (который другие процессы прочитать не могут).

Собственное хранилище учётных данных

Поскольку git-credential-store и подобные ей утилиты являются отдельными от Git программами, не сложно сделать так, чтобы **любая** программа могла быть помощником авторизации Git. Помощники, предоставляемые Git, покрывают наиболее распространённые варианты использования, но не все. Для примера допустим, что ваша команда имеет некоторые учётные данные, совместно используемые всей командой, например, для развёртывания. Эти данные хранятся в общедоступном каталоге, но вы не хотите копировать их в ваше собственное хранилище учётных данных, так как они часто изменяются. Ни один из существующих помощников не покрывает этот случай; давайте посмотрим, что будет стоить написать свой собственный. Есть несколько ключевых особенностей, которым должна удовлетворять эта программа:

1. Мы должны уделить внимание только одной операции get; store и erase являются операциями записи, поэтому мы не будем ничего делать при их получении.
2. Формат файла с совместно используемыми учётными данными такой же, как и у git-credential-store.

3. Расположение это файла более-менее стандартное, но, на всякий случай, мы должны позволять пользователям передавать свой собственный путь.

Мы снова напишем расширение на Ruby, но подойдет любой язык, так как Git может использовать всё, что сможет запустить на выполнение. Ниже приведён полный исходный код нашего нового помощника авторизации:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' # (1)
OptionParser.new do |opts|
  opts.banner = 'USAGE: git-credential-read-only [options] <action>'
  opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
    path = File.expand_path argpath
  end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' # (2)
exit(0) unless File.exists? path

known = {} # (3)
while line = STDIN.gets
  break if line.strip == ''
  k,v = line.strip.split '=', 2
  known[k] = v
end
```

```
        File.readlines(path).each do |fileline| # (4)
          prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
          if prot == known['protocol'] and host == known['host'] and user == known['username'] then
            puts "protocol=#{prot}"
            puts "host=#{host}"
            puts "username=#{user}"
            puts "password=#{pass}"
            exit(0)
          end
        end
```

1. Здесь мы разбираем аргументы командной строки, позволяя указывать пользователям входной файл. По умолчанию это ~/.git-credentials.
2. Эта программа отвечает только если операцией является get и файл хранилища существует.
3. В цикле считываются данные из стандартного ввода, до тех пор, пока не будет прочитана пустая строка. Введённые данные для дальнейшего использования сохраняются в отображении known.
4. Этот цикл читает содержимое файла хранилища, выполняя поиск соответствия. Если протокол и сервер из known соответствуют текущей строке, программа выводит результат и завершает работу.

Мы сохраним нашего помощника как `git-credential-read-only`, поместим его в один из каталогов, содержащихся в списке PATH, а также сделаем его исполняемым. Ниже приведено на что будет похож сеанс взаимодействия:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
```

```
username=bob
password=s3cre7
```

Так как его имя начинается с «git-», мы можем использовать простой синтаксис для настройки:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

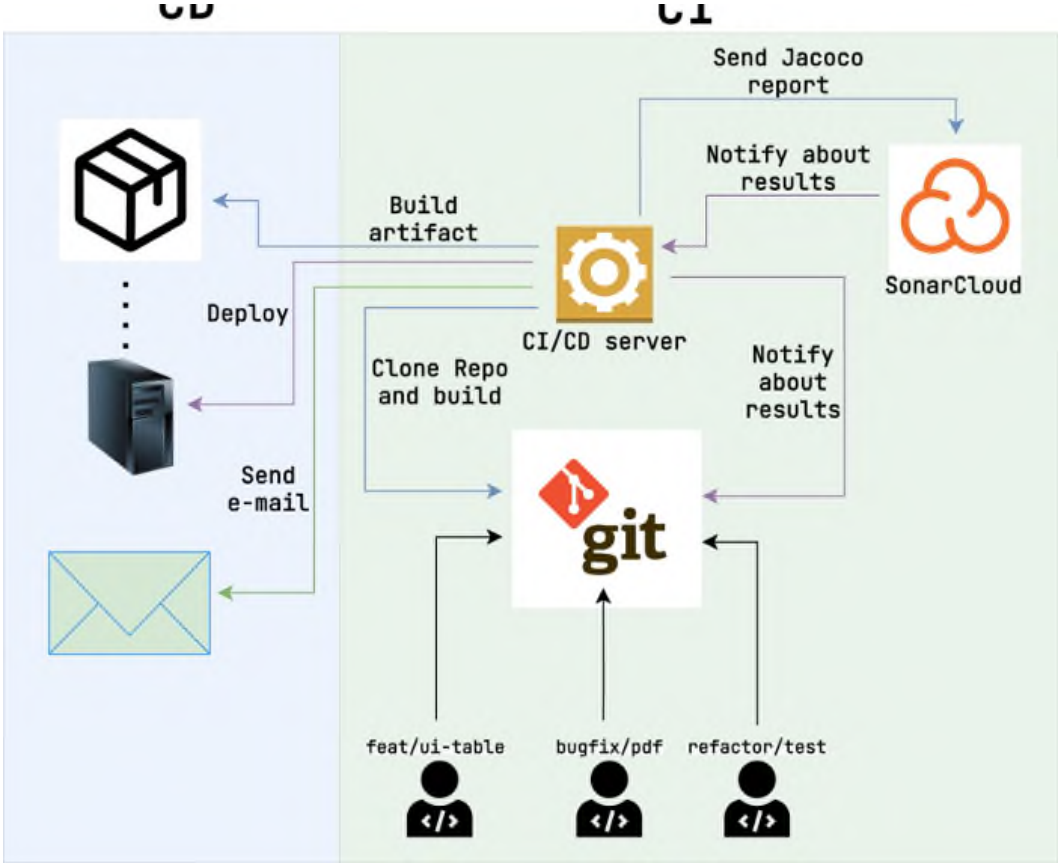
Как вы видите, расширять эту систему довольно просто и это позволяет решить некоторые общие проблемы, которые могут возникнуть у вас и вашей команды.



МОДУЛЬ 14. Жизненный цикл ПО

1. CI\CD
2. Практическое знакомство с Jenkins
3. Практическое знакомство с TeamCity
4. Практическое знакомство с Gitlab

1. CI\CD

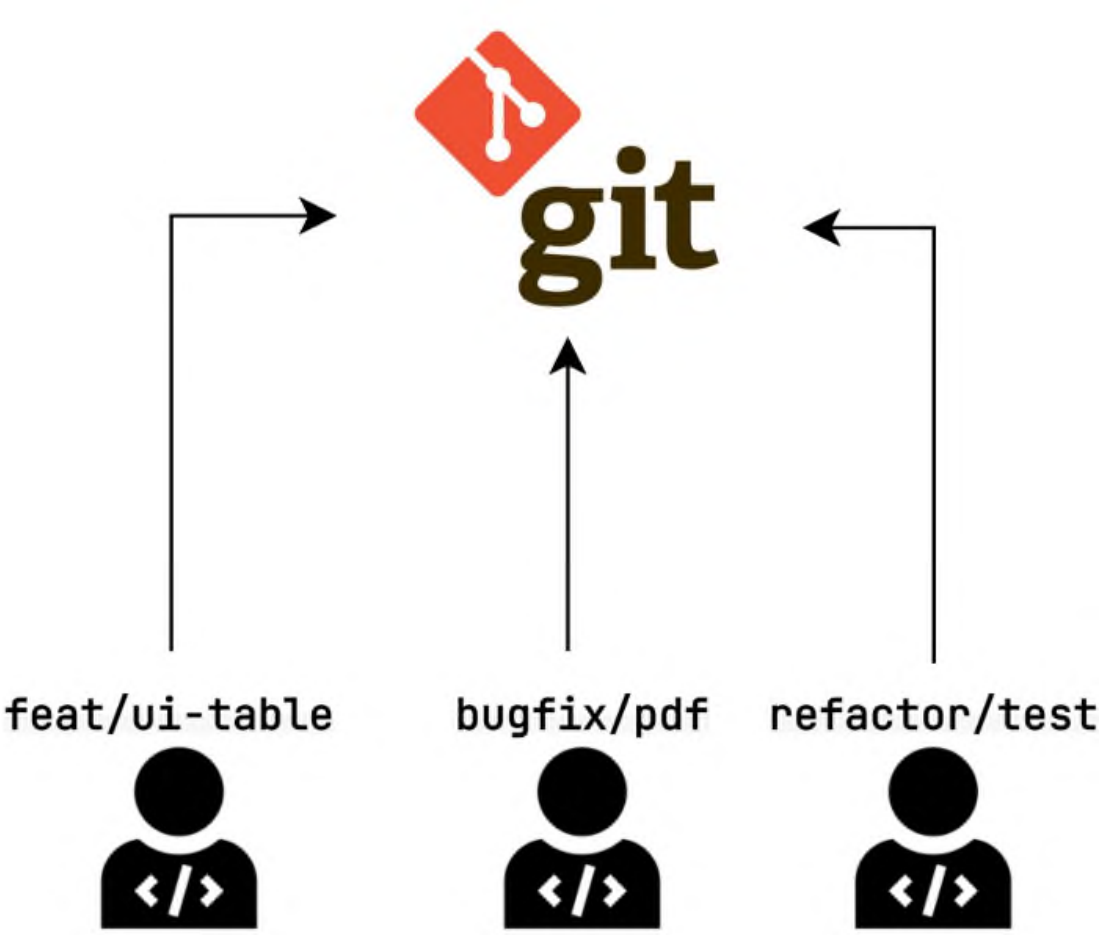


Основная цель любого программного проекта — зарабатывать деньги за счет автоматизации бизнес-процессов. Чем быстрее вы можете выпускать новые версии для клиентов, тем лучше для вашей компании. Но как быстро реализовать процесс релизов? Что ж, вы можете делать все вручную. Например, можно подключиться к удаленному серверу через SSH. Затем клонировать репозиторий с новым кодом, собрать его и запустить с помощью командной строки. Хотя это действительно работает, это неэффективный подход. Итак, сегодня мы обсуждаем автоматизацию выпуска продуктов и самого процесса разработки.

CI и CD — это два аббревиатуры, обозначающие непрерывную интеграцию (Continuous Integration) и непрерывную доставку (Continuous Delivery).

Непрерывная интеграция (Continuous Integration, CI)

Непрерывная интеграция описывает процесс прохождения изменений d репозитория b. Давайте рассмотрим простую схему, которая дает пример командной разработки.



Базовая командная разработка

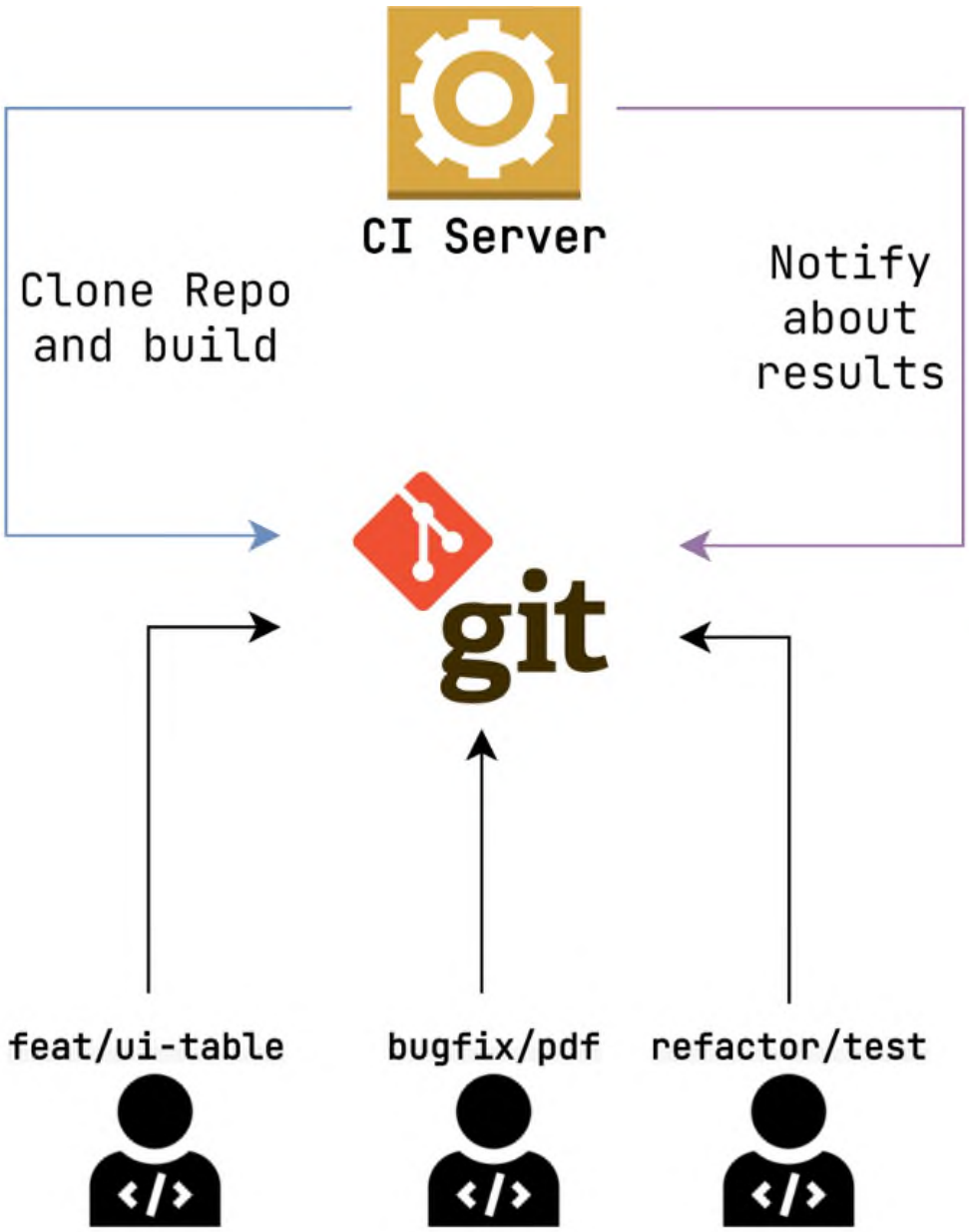
Группа людей может работать одновременно. Но со временем все изменения передаются в главную ветку. В любом случае, даже такая простая модель вызывает пару вопросов.

1. Как мы можем узнать, что код, поступающий в главную ветку, компилируется?
2. Мы хотим, чтобы разработчики писали тесты для кода. Как мы можем убедиться, что тестовое покрытие не уменьшается?
3. Все члены команды должны отформатировать код в соответствии с указанным стилем кода. Как мы можем проверить возможные нарушения?

Конечно, все описанные требования можно проверить вручную. Хотя это довольно неорганизованный подход. Более того, когда команда растет, поддерживать его становится труднее.

CI был сделан для автоматизации проверки заявленных вопросов.

Начнем с первого пункта. Как убедиться, что предстоящие изменения не испортят сборку? Для этого нам понадобится еще один блок в нашей схеме.



Простой CI

В соответствии с этим алгоритмом можно описать большинство процессов CI.

1. При каждом открытии пул реквеста (а также при отправке новых изменений) сервер Git отправляет уведомление серверу CI.
2. CI-сервер клонирует репозиторий, извлекает из ветки исходники (например, bugfix/wrong-sorting) и объединяет с master.
3. Затем запускается скрипт сборки. Например, ./gradlew build.
4. Если команда возвращает код 0, то сборка успешна. В противном случае она считается неудачной.
5. CI-сервер отправляет запрос с результатом сборки на Git-сервер.
6. Если сборка прошла успешно, разрешается мердж Pull Request-а. В противном случае слияние блокируется.

Процесс гарантирует, что любой код, поступающий в основную ветку, не нарушит дальнейшие сборки.

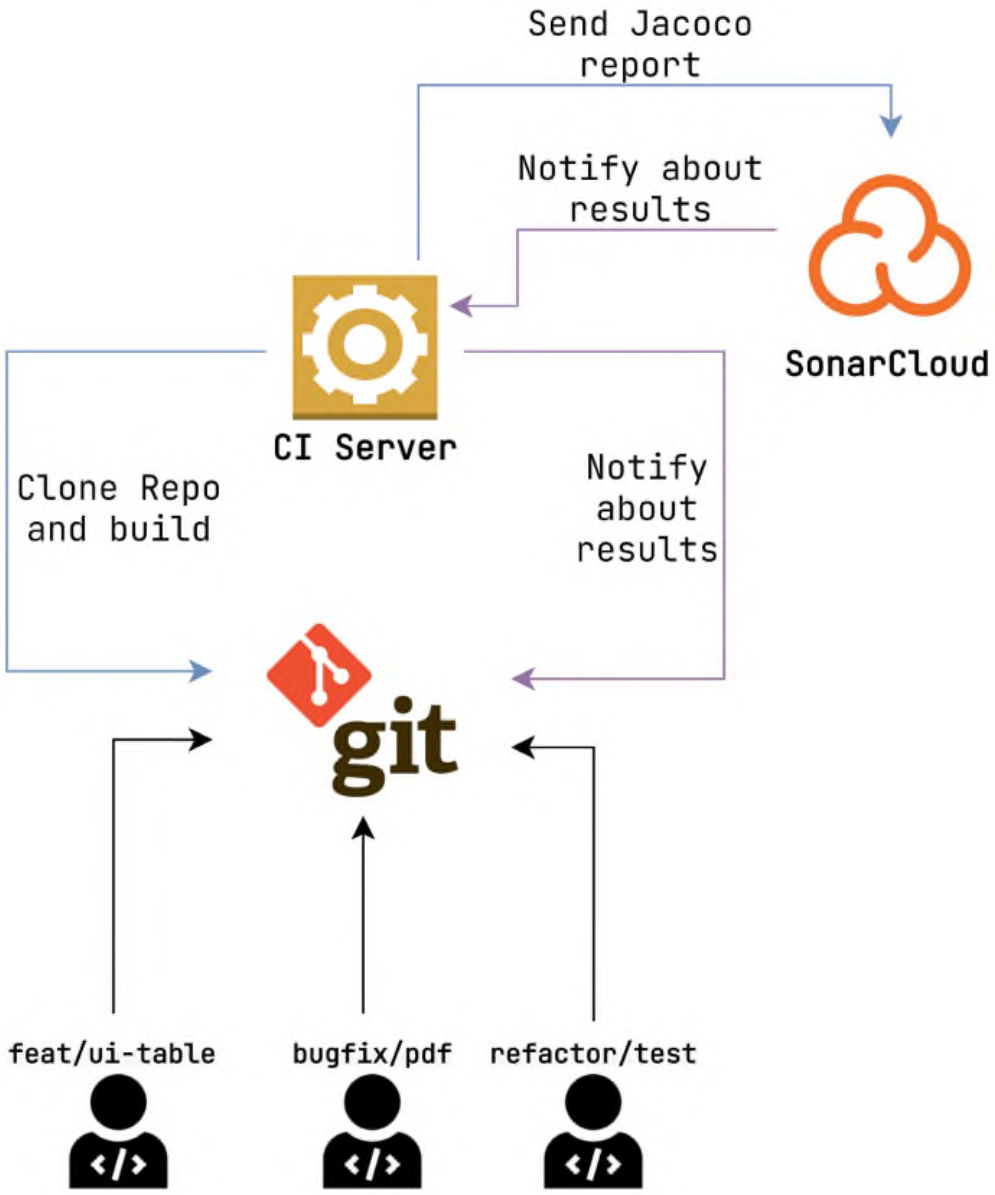
Проверка тестового покрытия

Давайте усложним задачу. Предположим, мы хотим установить минимальную планку тестового покрытия. Так что в любой момент покрытие ветки master не должно быть ниже 50%. Плагин Jacoco может легко решить проблему. Вам просто нужно настроить его таким образом, чтобы сборка не проходила, если значение тестового покрытия меньше принятого.

Реализация подхода — проста. Но есть нюанс. Она может работать только в том случае, если плагин был настроен с момента запуска проекта.

Представьте, что вы работаете над продуктом, которому пять лет. С момента его первого коммита проверка тестового покрытия не проводилась. Разработчики добавляли тесты случайным образом без какого-либо порядка. Но однажды вы решили увеличить количество тестов. Вы настраиваете плагин Jacoco так, чтобы минимальное значение равнялось 60%. Через некоторое время разработчик открывает новый пул реквест. Затем он внезапно понимает, что тестовое покрытие составляет всего 30%. Таким образом, для успешного завершения задачи необходимо покрыть еще не менее 30% кода продукта. Как вы понимаете, для проекта пятилетней давности это практически неразрешимая задача.

Что, если бы мы проверили только предстоящие изменения кода, а не весь продукт? Если разработчик изменил 200 строк в пул реквесте, ему нужно было бы покрыть как минимум 120 из них (если уровень тестового покрытия равен 60%). Но нет необходимости обрабатывать сотни других модулей, которые не являются частью задачи. Это может решить проблему. Как мы можем применить это к проекту? К счастью, решение есть.



CI с проверкой тестового покрытия

Отчет Jacoco отправляется на сервер тестового покрытия.

SonarCloud — одно из самых популярных решений.

Сервер хранит статистику предыдущих вычислений. Полезно рассчитать тестовое покрытие для предстоящих изменений, а также для всего кода. Затем результат анализа отправляется на сервер CI, который отправляет его обратно на сервер Git.

Такой рабочий процесс дает возможность применить культуру обязательного тестирования на любом этапе развития продукта. Потому что проверяются только новые изменения.

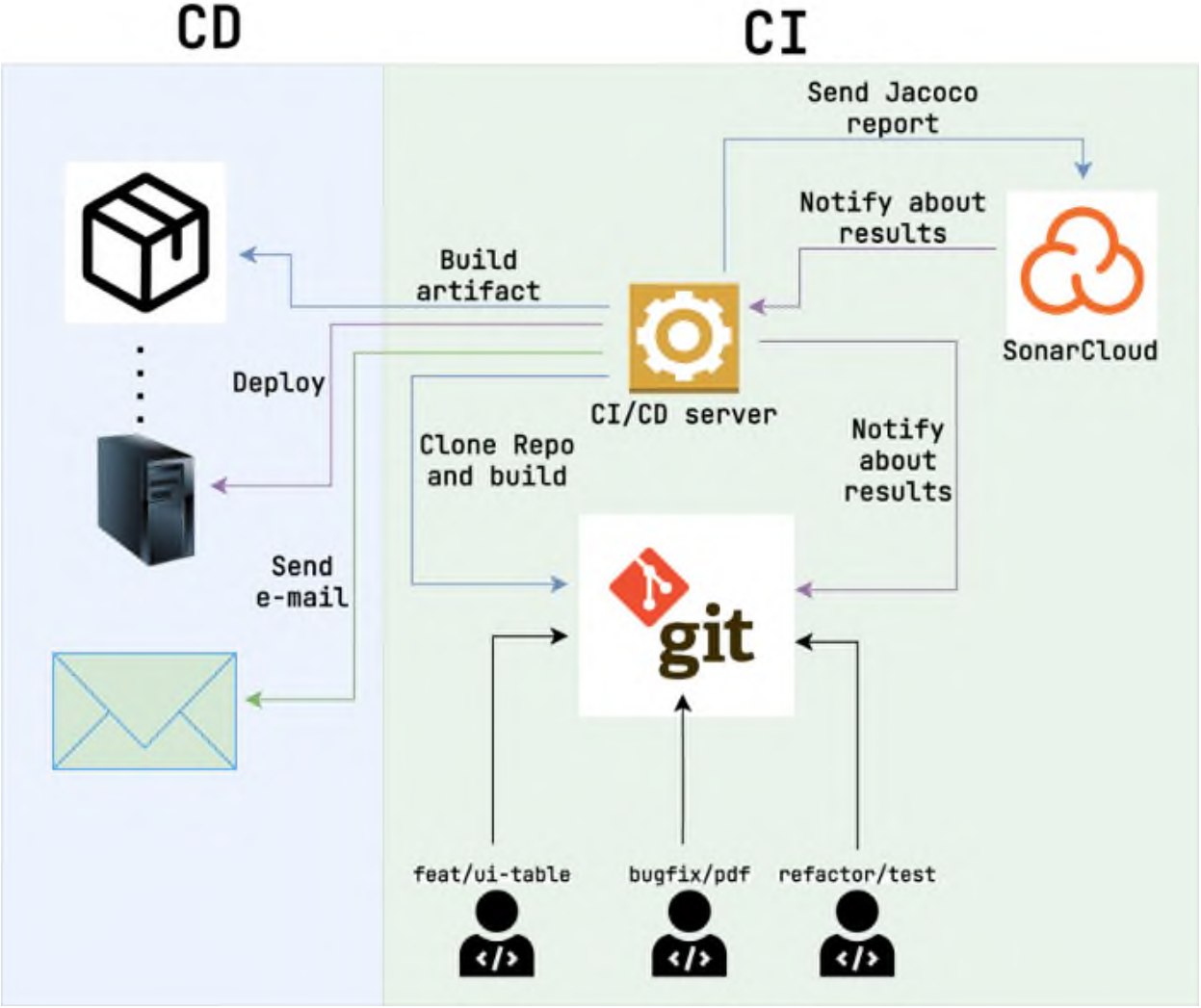
Говоря о стиле кода, здесь не так много различий. Вы можете попробовать плагин Checkstyle. Он автоматически завершает сборку, которая нарушает любое из заявленных требований. Например, в коде может быть неиспользованный импорт. Кроме того, вы можете использовать облачные

сервисы, которые запускают анализ кода и показывают результат в виде набора диаграмм (SonarCloud также может это делать).

Непрерывная доставка (Continuous Delivery, CD)

Непрерывная доставка описывает процесс автоматического развертывания новой версии продукта.

Давайте внесем некоторые изменения в схему CI. Вот так процесс CI/CD может выглядеть в реальном проекте.



Процесс CI/CD

Во-первых, теперь CI-сервер называется CI/CD-сервером. Дело в том, что часто задачи CI и CD выполняются одним и тем же диспетчером задач. Итак, мы рассматриваем этот подход.

Хотя это не обязательно. Например, можно делегировать задания CI GitLab CI, а задания CD — Jenkins.

Правая часть схемы представляет собой CI. Мы уже обсуждали это выше. Левая яасть изображения — CD. Джоб CD создает проект (или повторно использует артефакты, созданные на этапе CI) и развертывает его на конечном сервере.

Стоит отметить, что в нашем случае сервер — это абстракция. Например, развертывание может продолжиться в кластере Kubernetes. Или серверов может быть несколько.

После завершения этапа развертывания обычно отправляются электронные письма. Например, CD-сервер может уведомить подписчиков об успешном или неудачном развертывании.

Во всяком случае, есть важный вопрос. Когда нам следует запускать задачи в CD? Триггеры могут отличаться.

- Развертывать после каждого мерджа Pull Request.
- Развертывание по расписанию.
- Деплой после каждого слияния пул реквеста с определенной веткой.
- Комбинированный вариант.

Первый вариант устанавливает процесс таким образом, чтобы задания CI и CD всегда выполнялись последовательно. Этот подход довольно популярен в разработке с открытым исходным кодом. Библиотека Semantic Release помогает настроить проект для прозрачной интеграции этого процесса.

Важно знать определение развертывания (deploy). Это не обязательно означает, что что-то где-то запускается. Если вы разрабатываете библиотеку, то запуска фактически нет. Вместо этого процесс развертывания означает выпуск новой версии библиотеки.

Второй вариант не зависит от процесса CI. Потому что проект деплоится по заранее определенному графику. Например, каждый день в 01:00.

Третий пункт аналогичен первому. Хотя есть отличия. Предположим, что у нас есть две основные ветки в нашем репозитории. Ветка develop и master. В разработке собраны самые актуальные изменения. А во второго только релизы. Если нам нужно развернуть только master ветвь, нет необходимости запускать джоб CD при слиянии с develop.

Последний пункт — совокупность всех подходов. Например, ветвь develop может быть развернута в соответствии с расписанием разработки. А master развертывается в производственной среде при каждом слиянии пул реквеста.

Инструменты

Рынок предлагает десятки решений для автоматизации процессов CI/CD. Давайте посмотрим на некоторые из них.

1. **Jenkins**. Один из самых востребованных инструментов CI/CD в мире. Он стал настолько популярным из-за своей политики открытого исходного кода. Так что вам не нужно ничего платить. Jenkins позволяет в императивном порядке описывать конвейеры сборки с помощью Groovy. С одной стороны, это обеспечивает большую гибкость. Но, с другой стороны, это требует более высокого уровня компетентности.

2. **GitHub Actions.** Инструмент CI/CD включен в GitHub и GitHub Enterprise. В отличие от Jenkins, GitHub Actions предоставляет декларативные сборки с конфигурацией YAML. Кроме того, решение имеет множество интеграций с различными системами контроля качества (например, SonarCube). И сборки можно описать всего несколькими строками кода.

3. **GitLab CI**. Он очень похож на GitHub Actions. Тем не менее, у него есть особенности. Например, GitLab CI может указать на конкретные тесты, которые не прошли сборку.

4. **Travis CI**. Облачная служба CI/CD. Она предлагает множество возможностей, не требующих сложной настройки. Например, шифрование данных, которые должны быть скрыты в публичном репозитории. Кроме того, приятным бонусом является то, что Travis CI можно применять к общедоступным проектам с открытым исходным кодом GitHub, GitLab и BitBucket абсолютно бесплатно.

2. Практическое знакомство с Jenkins

Как установить Jenkins и настроить автоматическую сборку maven-проекта на Ubuntu 20.04

Jenkins является одним из популярных инструментов, позволяющий реализовывать подход непрерывной интеграции (Continuous Integration, CI) и непрерывной поставки (Continuous Delivery, CD) программного обеспечения.

CI существенно облегчает интеграцию разрозненных копии проекта с основной ветвью разработки, позволяет настраивать автоматическую сборку проекта и запускать тесты.

CD обеспечивает в автоматическом режиме поставку и развертывание результатов сборки на целевых серверах.

В этой статье мы рассмотрим, как установить Jenkins, и на примере веб-приложения на Java покажем, как автоматизировать процесс сборки и поставки его на веб-сервер Jetty.

Исходный код веб-приложения лежит в git-репозитории, а в качестве инструмента сборки используется Maven.

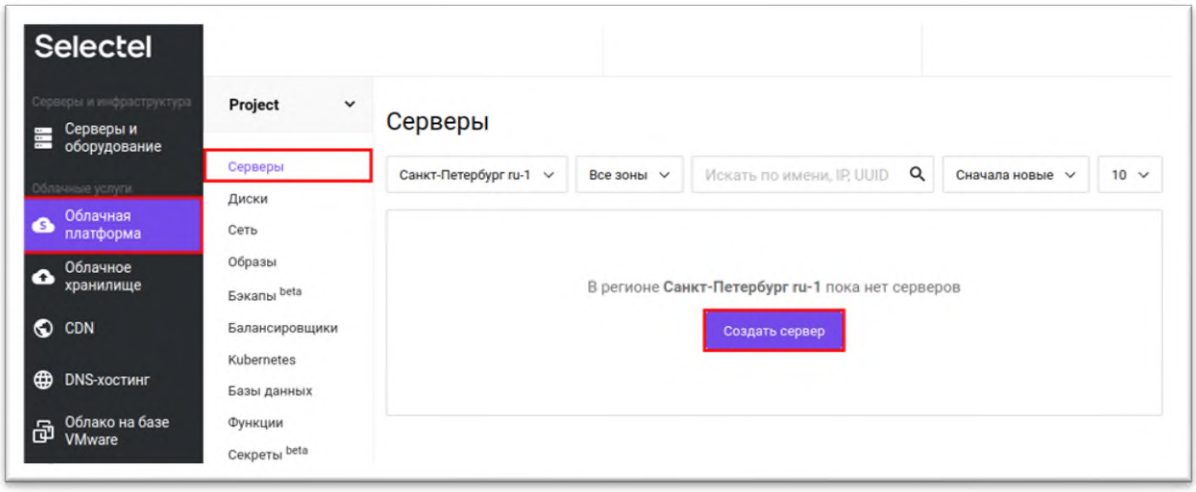
Требования

- Сервер Ubuntu 20.04 ;
- Пользователь с root-правами или доступ через sudo.

Шаг 1. Подготовка инфраструктуры

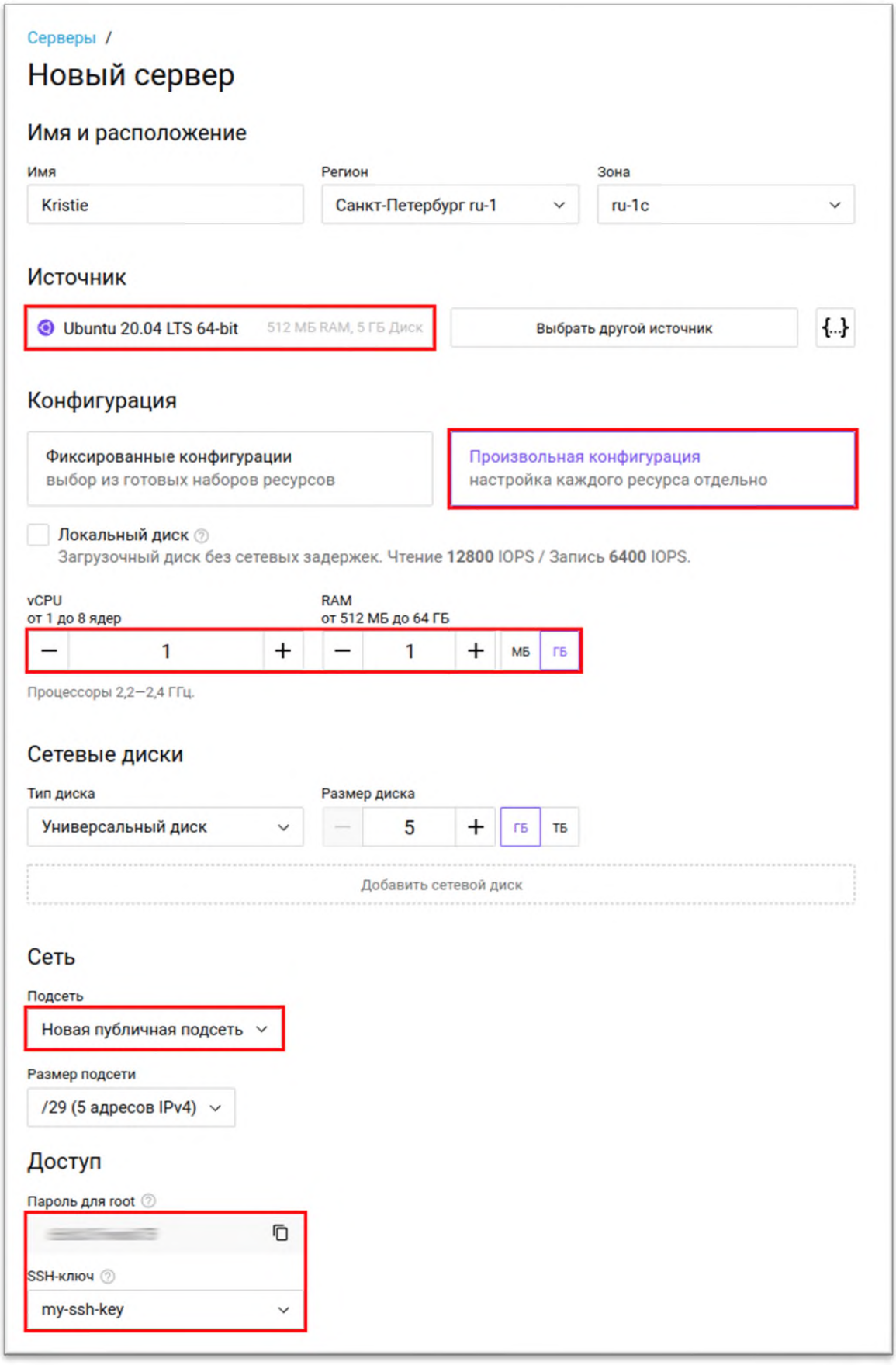
Мы будем устанавливать Jenkins на облачную виртуальную машину, развернутую на платформе Selectel.

В панели управления платформой переходим в раздел «Облачная платформа» — «Серверы», и нажимаем кнопку «Создать сервер».

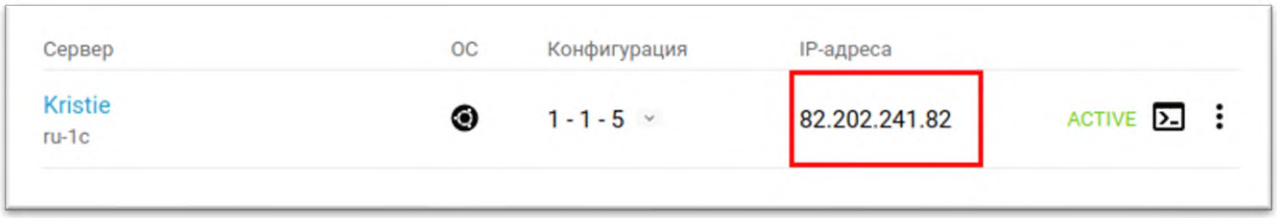


На следующем экране нужно выбрать параметры создаваемой ВМ. Выберем ОС — Ubuntu 20.04, произвольную конфигурацию — 1 vCPU, 1 ГБ

оперативной памяти и 5 ГБ диска. В разделе «Сеть» выберем или создадим подсеть с публичным IP-адресом, чтобы к машине можно было подключаться из интернета. Также обратите внимание на раздел «Доступ»: нужно либо записать пароль root-пользователя, либо выбрать SSH-ключ.



Через несколько минут виртуальная машина будет создана, и нам нужно скопировать ее IP-адрес:



Чтобы подключиться к машине, выполните команду, указав IP-адрес вашего сервера:

ssh root@

Если при создании машины вы выбрали SSH-ключ, то подключиться можно без пароля:

ssh -i <приватный_ключ> root@

Шаг 2. Установка JDK

Для работы Jenkins и Maven нужно установить Java, причем для Maven нужен именно JDK. Установим его:

apt-get update

apt-get install -y openjdk-11-jdk

По завершении установки выполняем проверку:

java -version

Вывод:

openjdk version "11.0.13" 2021-10-19 OpenJDK Runtime Environment (build 11.0.13+8-Ubuntu-0ubuntu1.20.04) OpenJDK 64-Bit Server VM (build 11.0.13+8-Ubuntu-0ubuntu1.20.04, mixed mode, sharing)

Шаг 3. Установка Jenkins

Выполняем в командной строке Ubuntu следующие команды:

curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo tee \ /usr/share/keyrings/jenkins-keyring.asc > /dev/null

echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \ https://pkg.jenkins.io/debian-stable binary/ | sudo tee \ /etc/apt/sources.list.d/jenkins.list > /dev/null

$ apt-get update

$ apt-get install -y jenkins

После установки система автоматически запустит сервис Jenkins. Проверяем его статус с помощью команды:

$ systemctl status jenkins

Вывод:

● jenkins.service - LSB: Start Jenkins at boot time Loaded: loaded (/etc/init.d/jenkins; generated) Active: active (exited) since Wed 2021-12-22 05:09:54 UTC; 23s ago Docs: man:systemd-sysv-generator(8) Tasks: 0 (limit: 1105) Memory: 0B CGroup: /system.slice/jenkins.service

Как видим, сервис запущен и находится в активном статусе. Если сервис неактивен, то необходимо проверить, свободен ли порт 8080 и хватает оперативной памяти (Jenkins требует минимум 256 МБ).

Шаг 4. Установка Git

Git нам понадобится для того, чтобы клонировать репозиторий с исходным кодом на нашу машину.

Выполняем команду:

$ apt-get install -y git

Проверяем установленную версию:

$ git --version

Вывод:

git version 2.25.1

Шаг 5. Установка Maven

Устанавливаем Maven:

$ apt-get install -y maven

Проверяем успешность установки:

$ mvn --version

Вывод:

Maven home: /usr/share/maven Java version: 11.0.13, vendor: Ubuntu, runtime: /usr/lib/jvm/java-11-openjdk-amd64 Default locale: en, platform encoding: UTF-8 OS name: "linux", version: "5.4.0-91-generic", arch: "amd64", family: "unix"

Здесь необходимо запомнить значение Maven home. В нашем случае это /usr/share/maven. Этот путь будет нужен в дальнейшем для настройки Jenkins.

Шаг 6. Настройка Jenkins

Все необходимые установки сделаны, и можно перейти к настройке Jenkins.

Управление Jenkins осуществляется через веб-интерфейс, по умолчанию доступный на порту 8080. Открываем в браузере страницу http://[IP-адрес сервера]:8080:



Если интерфейс не появился, то необходимо открыть порт 8080 и проверить статус работы сервиса Jenkins.

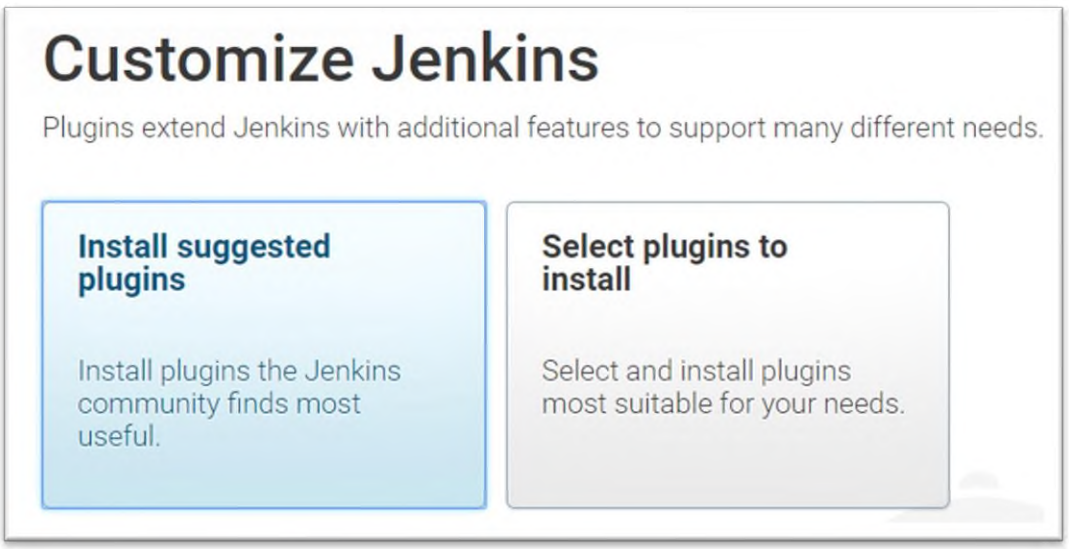
Для продолжения настройки нам потребуется ввести пароль администратора, который хранится в файл /var/lib/jenkins/secrets/initialAdminPassword.

Возвращаемся в командную строку Ubuntu и просматриваем его с помощью команды cat:

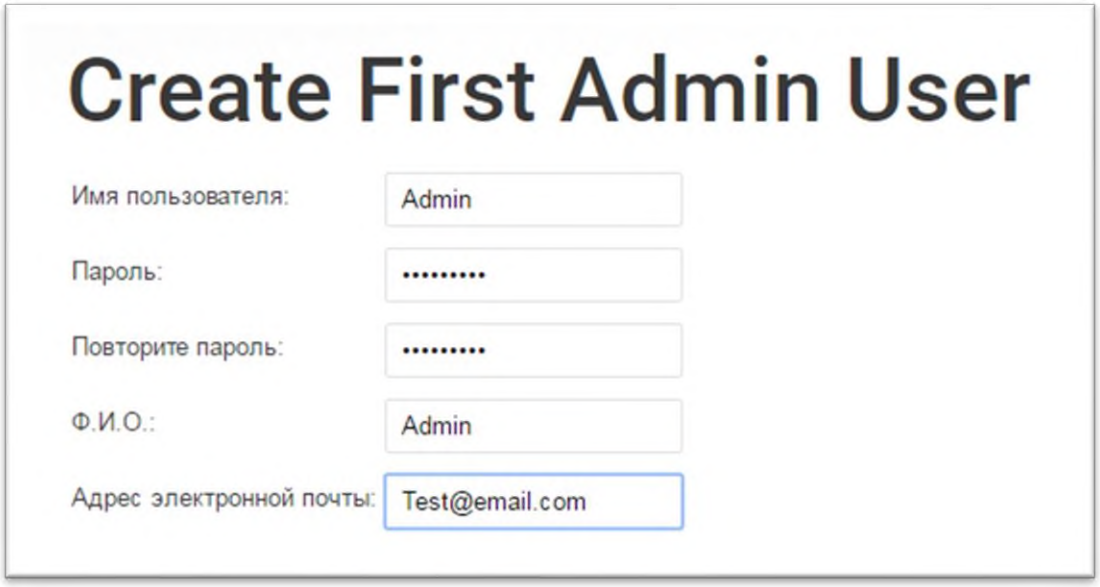
$ cat /var/lib/jenkins/secrets/initialAdminPassword

Копируем пароль, вставляем его в поле ввода и нажимаем «Продолжить».

На следующем шаге нам необходимо выбрать режим установки плагинов, которые будут установленные вместе с Jenkins. Выбираем Install suggested plugins:



После установки всех плагинов создаем профиль администратора:



Имя пользователя и пароль запоминаем для дальнейшей работы.

На следующем экране Jenkins попросит проверить и подтвердить публичный адрес сервера.



Нажимаем «Продолжить». На экране появится сообщение, свидетельствующее о готовности Jenkins к работе:

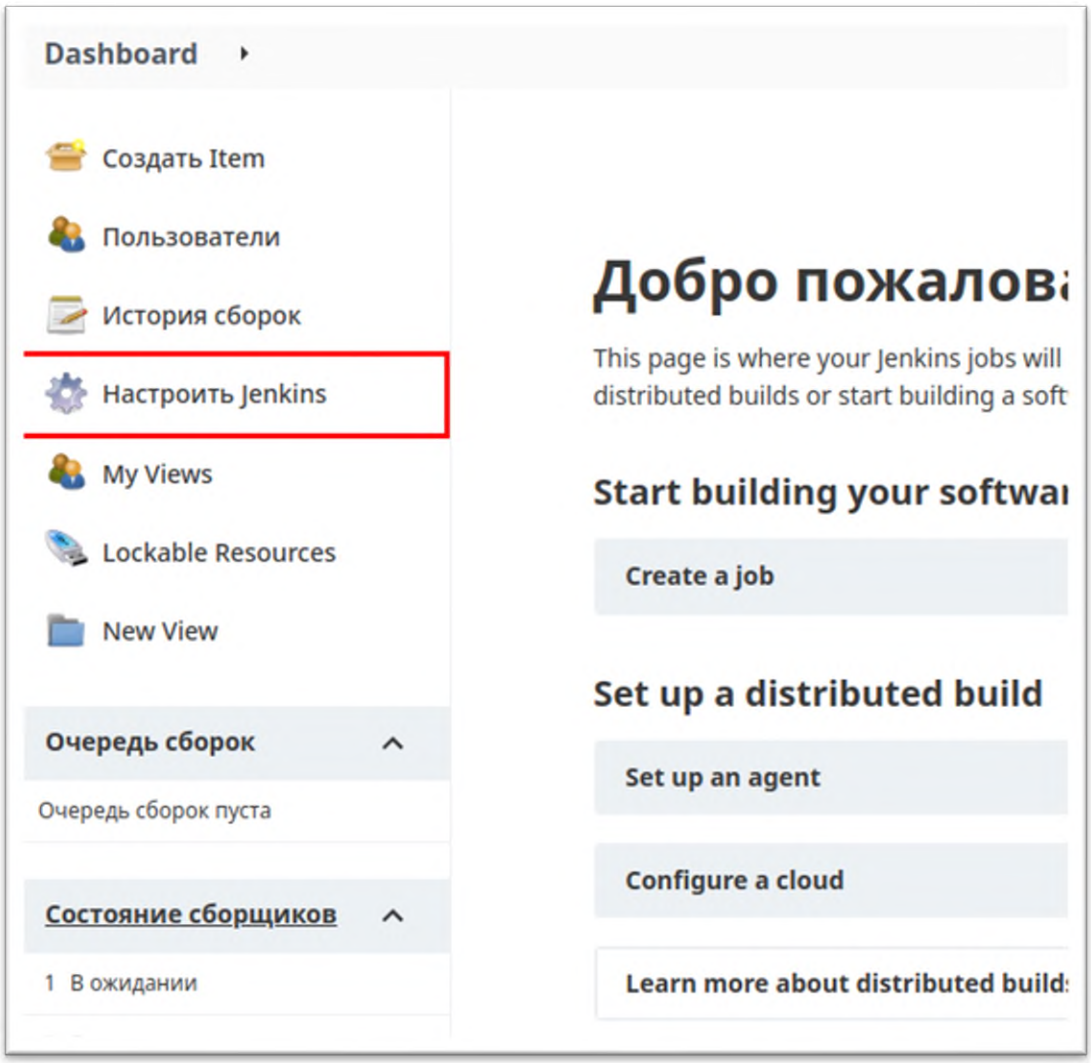


На этом основная настройка закончена. Нажимаем на кнопку Start using Jenkins и переходим в рабочую панель Jenkins.

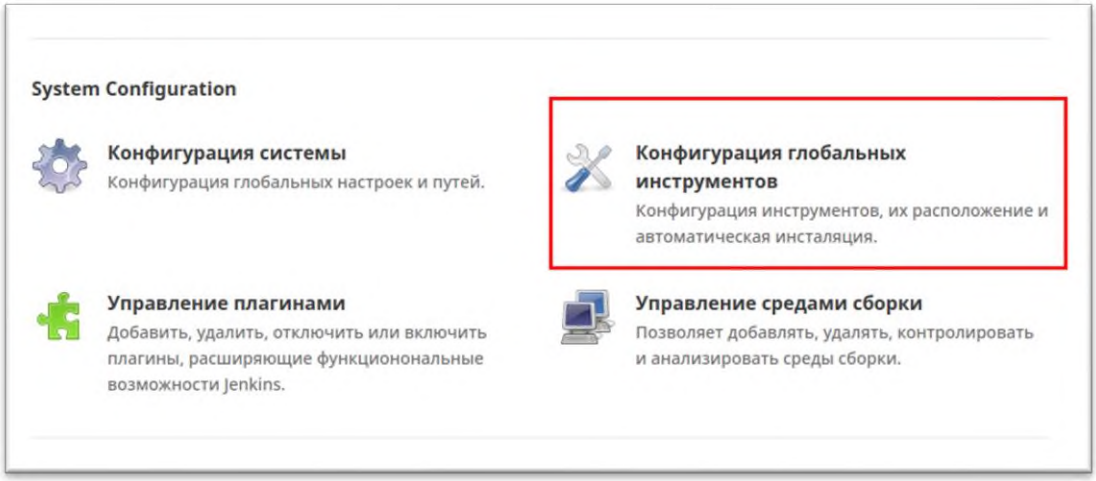
Шаг 7. Настройка Maven в Jenkins

Jenkins не всегда распознает установленный в системе Maven, поэтому его нужно указать вручную.

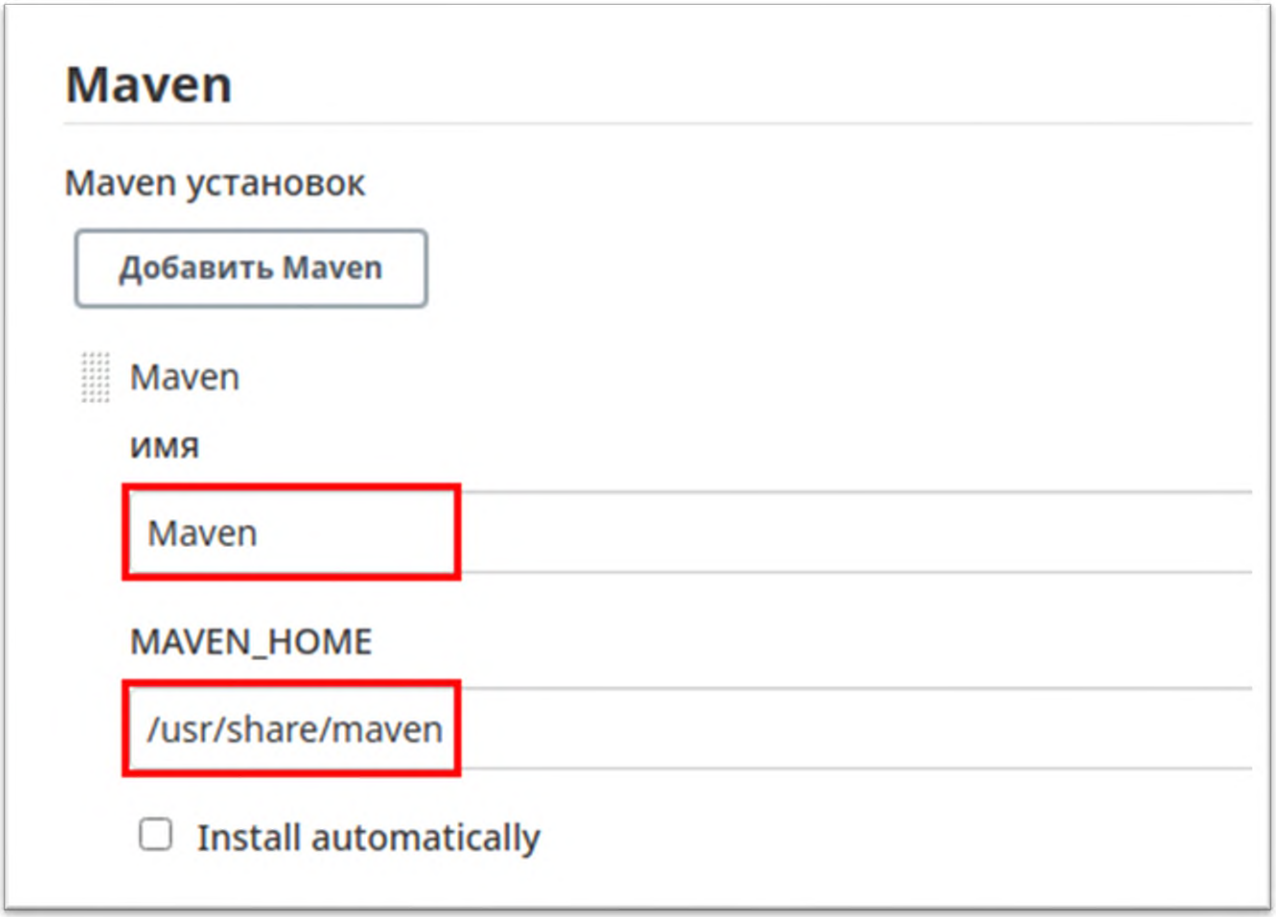
Выбираем «Настроить Jenkins» в левом меню:



Переходим в «Конфигурация глобальных инструментов»:



В появившемся окне ищем раздел Maven и нажимаем на кнопку «Добавить». Снимаем флажок Install automatically, и открывается два поля для ввода.



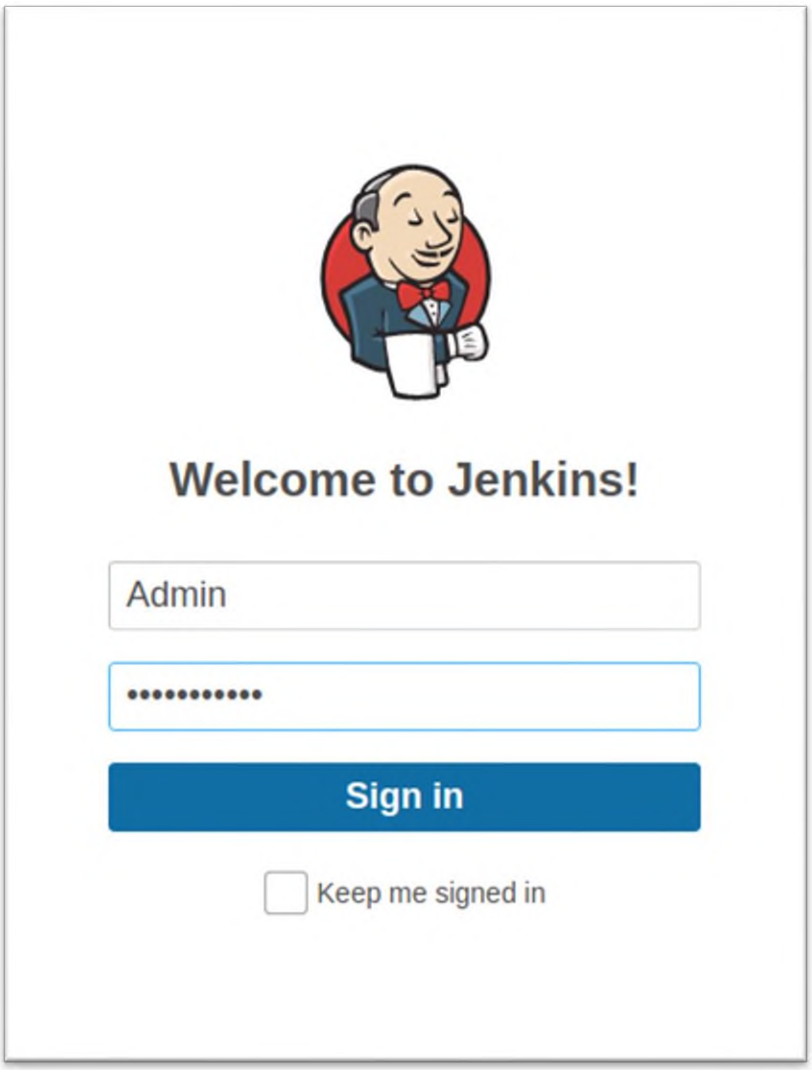
Вводим произвольное имя и путь до установочной папки Maven home, полученный нами на 5-м шаге во время проверки версии.

Сохраняем настройки, и чтобы изменения вступили в силу, перезапускаем Jenkins:

$ systemctl restart jenkins

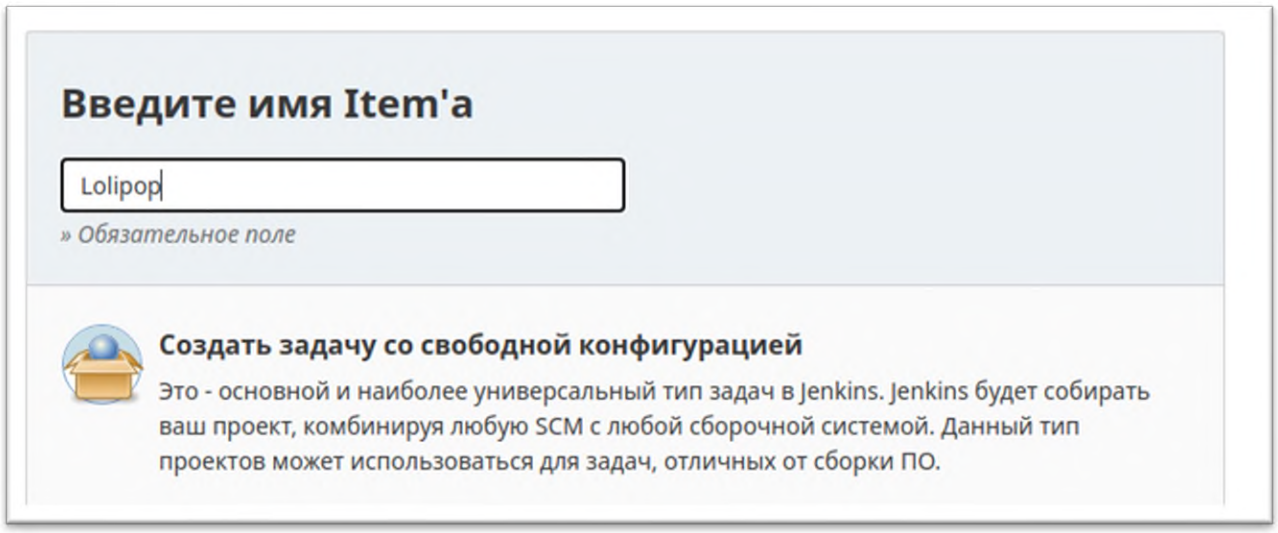
Шаг 8. Создание автоматической сборки

После перезагрузки открываем снова веб-интерфейс Jenkins и заходим под учетной записью администратора, которую мы создавали на этапе настройки:

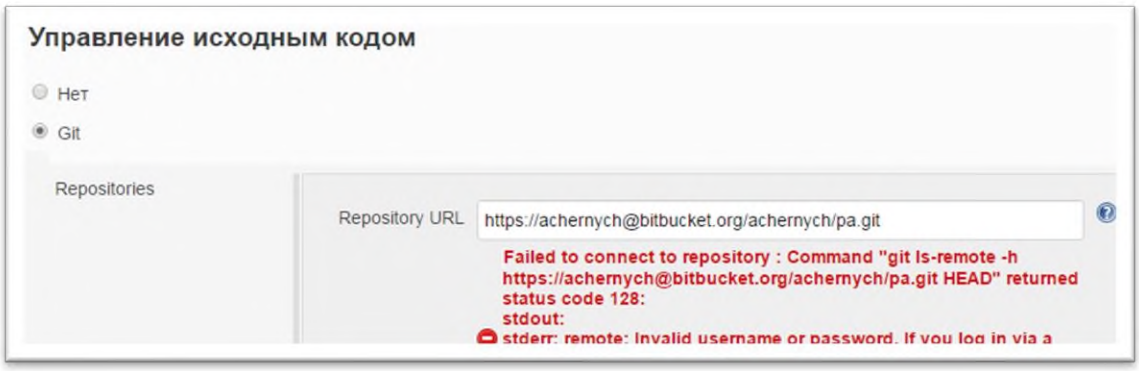


В появившемся окне в левом меню нажимаем на пункт «Создать Item».

Вводим название проекта, выбираем «Создать задачу со свободной конфигурацией» и нажимаем «ОК» внизу страницы.



На вкладке General находим раздел «Управление исходным кодом» и выбираем Git. Система предлагает ввести нам URL Git репозитория. Вводим его:

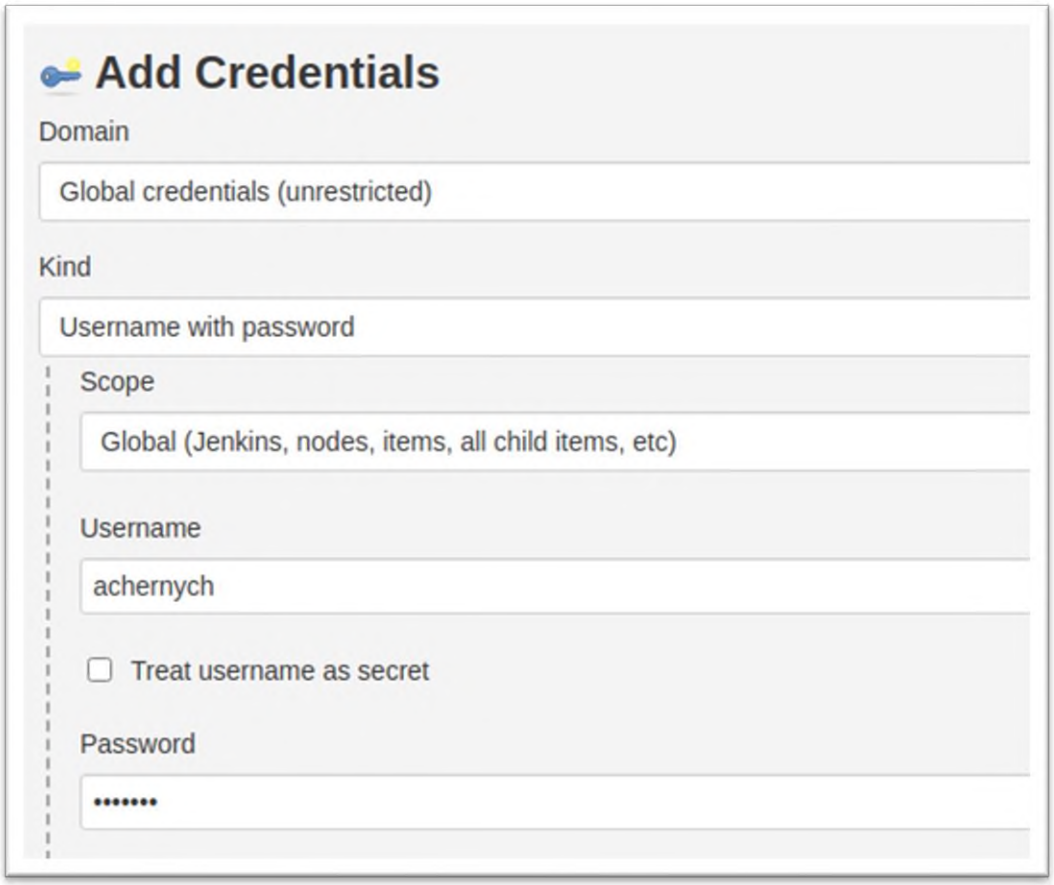


Если репозиторий приватный, то система выдаст ошибку о том, что нет доступ к репозиторию.

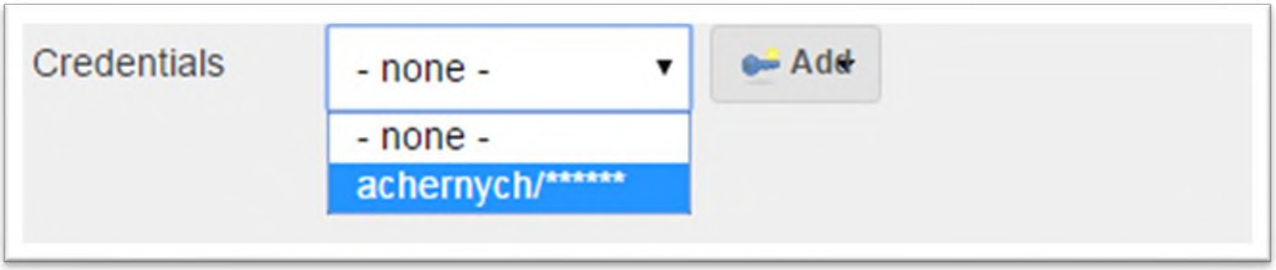
Добавляем логин и пароль в подразделе Credentials. Выбираем из выпадающего списка Jenkins:



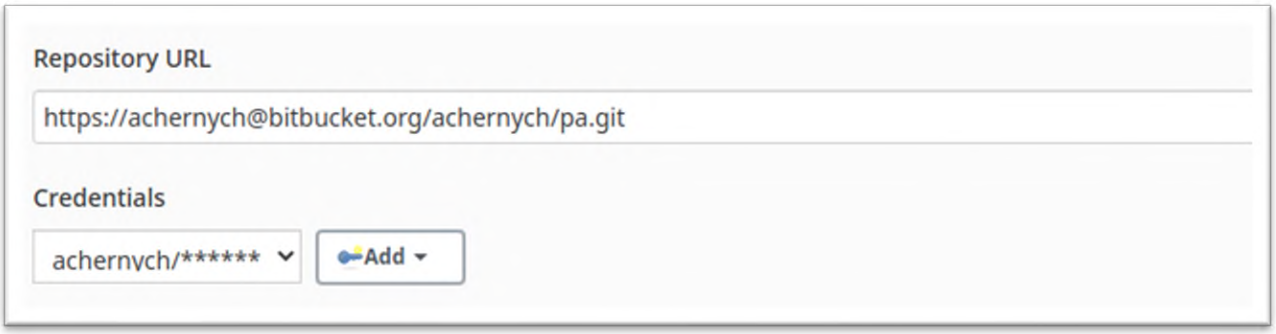
В появившемся окне вводим Username и Password:



Добавляем и выбираем из выпадающего списка созданные Credentials:

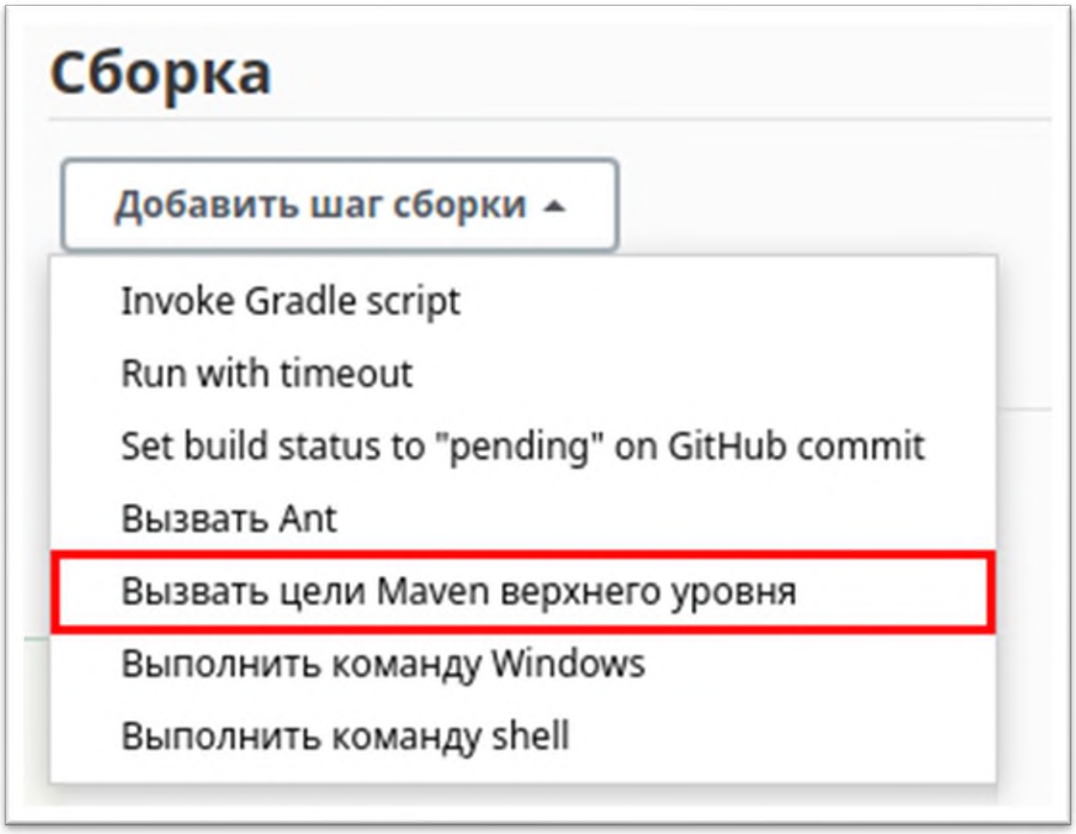


Ошибка доступа больше не выводится:



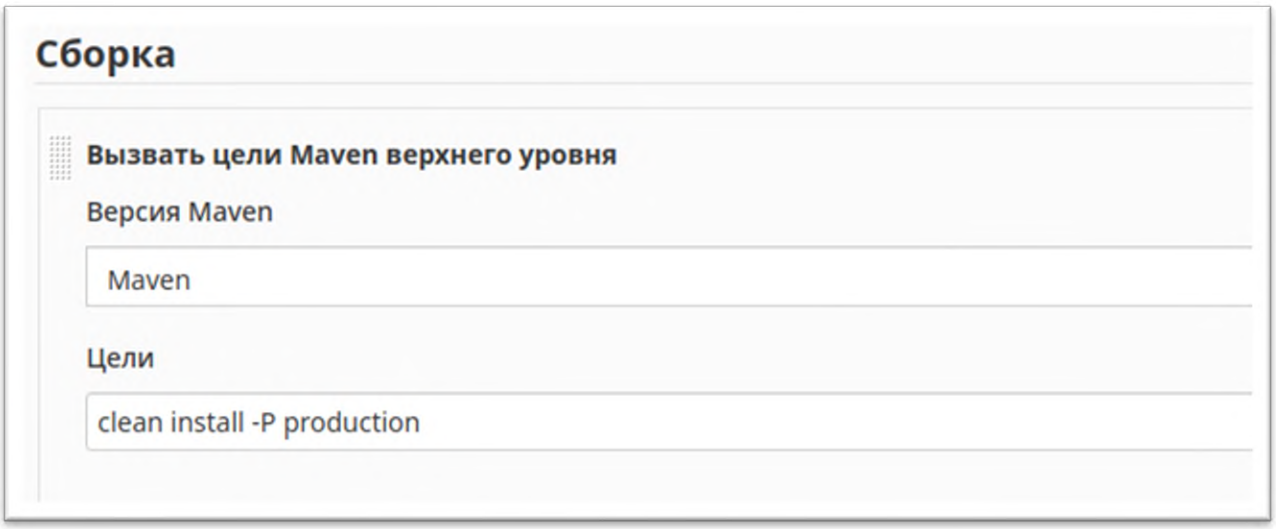
После того как мы указали репозиторий, из которого необходимо получать исходный код, переходим к настройке сборки.

Находим раздел «Сборка», нажимаем «Добавить шаг сборки» и выбираем из выпадающего списка «Вызвать цели Maven верхнего уровня».



Выбираем нашу настройку Maven и в поле «Цели» вставляем команду для сборки maven-проекта:

clean install -P production



Результат сборки будет сохраняться в директории /var/lib/jenkins/workspace/Lolipop/target/, где Lolipop — название нашего проекта.

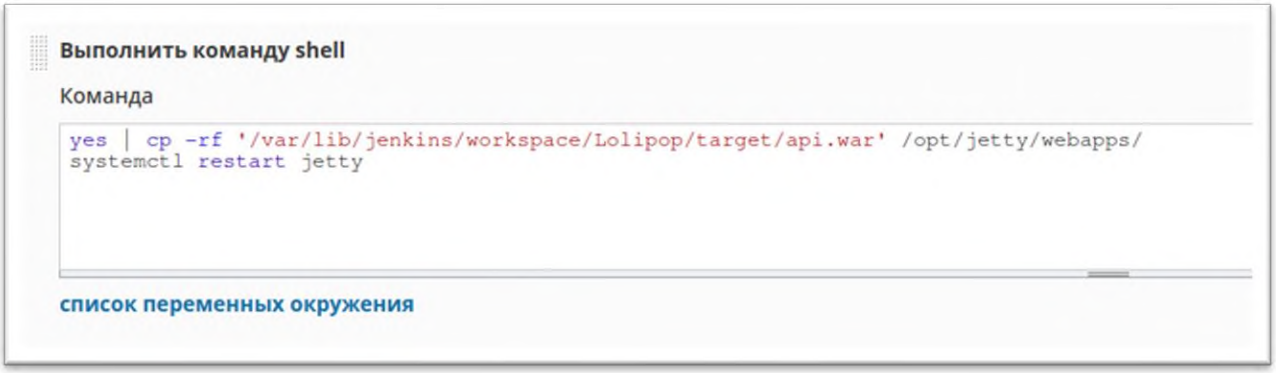
В данном примере результатом сборки будет готовый к развертыванию war-файл.

В качестве примера рассмотрим, каким образом автоматизировать процесс развертывания war-файла на веб-сервер Jetty, который работает на этой же машине.

Добавим еще один шаг сборки, который будет копировать war-файл в папку webapps Jetty и перезапускать его. На этот раз выбираем из выпадающего списка «Выполнить команду shell».

yes | cp -rf '/var/lib/jenkins/workspace/Lolipop/target/api.war' /opt/jetty/webapps/

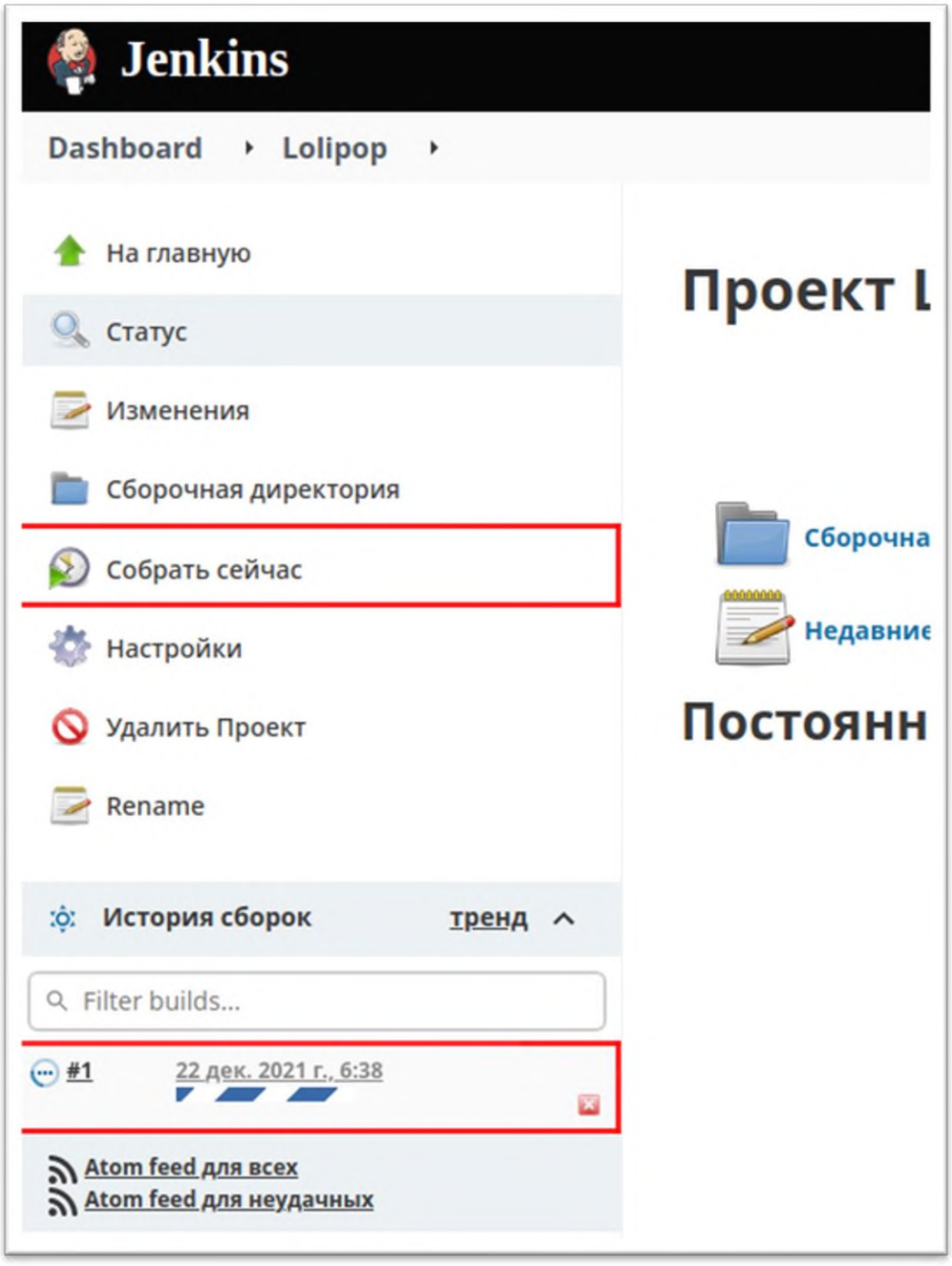
systemctl restart jetty



На этом настройка сборки закончена, нажимаем «Сохранить» и переходим к запуску.

Шаг 9. Запуск сборки

Для запуска сборки в меню выбираем «Собрать сейчас». После этого в левом нижнем углу в разделе «История сборок» появится индикатор процесса сборки с датой и временем запуска:

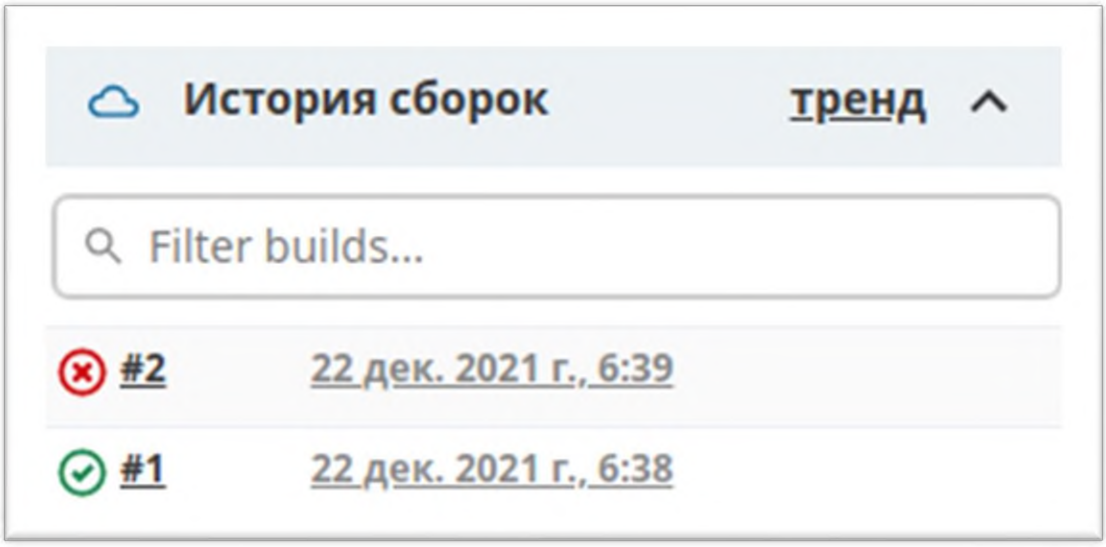


Во время сборки Jetty будет выполнять последовательно следующие задачи:

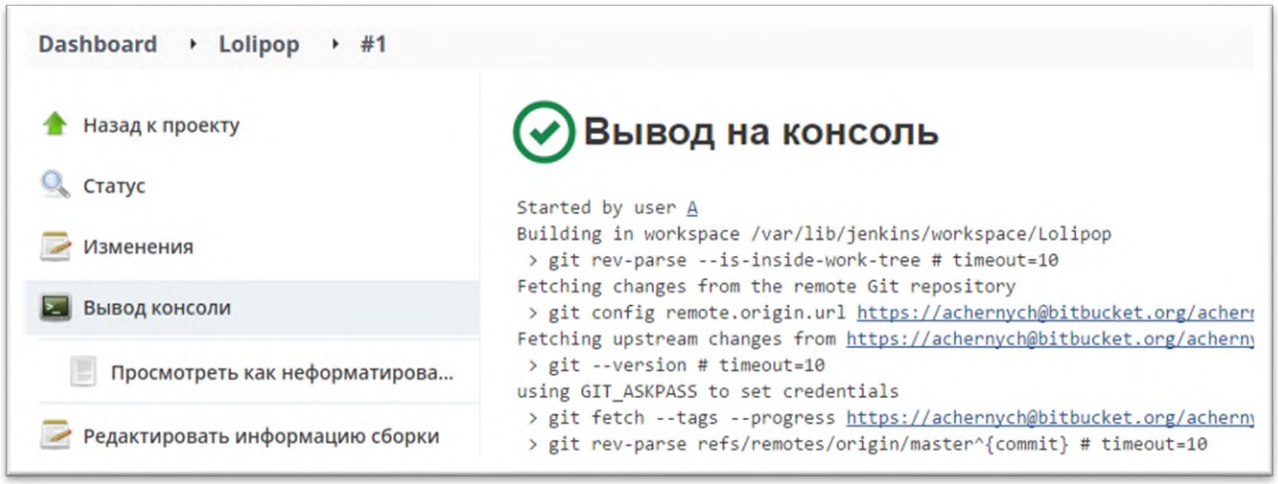
1. Загрузка из Git репозитория последней версии исходного кода проекта.
2. Запуск сборки загруженного Maven проекта.
3. Размещение результатов сборки в папке workspace.
4. Копирование war-файла в папку webapps веб-сервера Jetty.

5. Перезапуск Jetty.

Если сборка завершилась с ошибкой, то маркер будет красного цвета, а если успешно, то зеленого.



Нажав на номер сборки и перейдя в левом меню в раздел Console Output, можно отследить отладочный вывод сборки:



Автоматическая сборка и доставка на веб-сервер настроена. Теперь мы можем запускать ее в нужный момент с помощью кнопки «Собрать сейчас».

Если у вас несколько серверов выкладки с разными конфигурациями, вы можете настроить сборку под каждую выкладку.

Шаг 10. Как изменить порт подключения Jenkins

В ряде случаев порт 8080 занят или его необходимо освободить под другие сервисы. В рассмотренном примере Jetty по умолчанию запускается на порту 8080.

Чтобы Jenkins подключался к другому порту, в файле /etc/default/jenkins нужно изменить значение переменной HTTP_PORT на желаемый порт и перезапустить сервис Jenkins.

port for HTTP connector (default 8080; disable with -1) HTTP_PORT=8081

Обратите внимание, что этот способ не сработает, если вы хотите запустить Jenkins на одном из привилегированных портов, например, на 80. Jenkins запускается не от root-пользователя, а значит у него нет прав слушать порты до 1024.

Лучший способ это сделать — оставить Jenkins работать на порту 8080 и настроить переадресацию трафика с помощью iptables.

Заключение

Jenkins поддерживает технологии для работы с разными системами контроля версий, включая Git, CVS, Subversion, Clearcase и Mercurial, может собирать проекты на Ant, Maven и Gradle, а также исполнять команды Windows, shell-скрипты и отправлять уведомления о состоянии сборки в автоматическом режиме.

Благодаря такому разнообразному инструментарию Jenkins можно применять на проектах с разными технологиями и любой сложности.

Jenkins также поддерживает триггеры сборки, которые могут вызывать сборку по определенному расписанию или событию.

Все эти преимущества делают Jenkins незаменимым инструментом при построении процесса разработки**.**

3.Практическое знакомство с TeamCity

TeamCity — пожалуй, один из самых популярных CI/CD-серверов, который используют разработчики по всему миру. В этой статье я расскажу про базовые концепции, на которых он построен, и на примерах покажу, как можно настроить CI/CD в вашей команде.

Build Agent

Перед тем как начать разбираться в TeamCity, стоит понять, как в нем построен процесс сборки и выполнения CI/CD задач.

В TeamCity есть такое понятие, как build agent — это какая-то машина, на которой происходит процесс сборки. Единовременно агент может выполнять всего одну сборку.

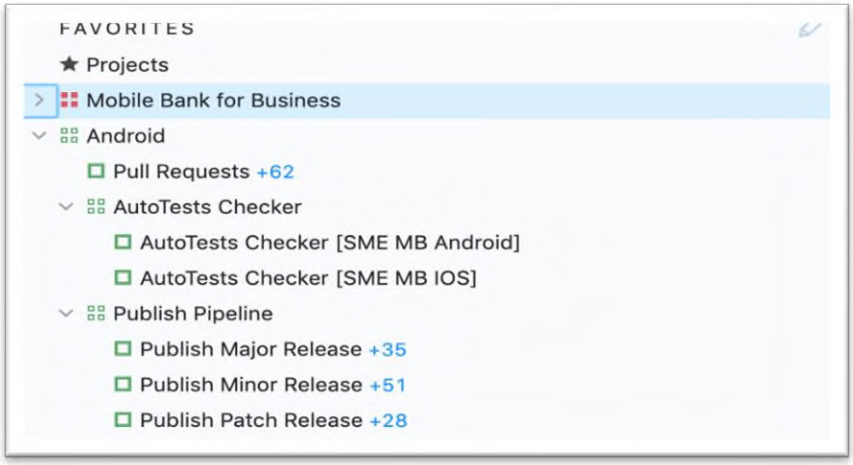
Сборка проходит в несколько шагов:

- После некоторого события, которое было поймано триггером сборки (build trigger), сборка попадает в очередь и будет начата сразу, как только появится свободный агент.
- Начинается сборка, которая поочередно выполняет все шаги сборки (build steps), которые были указаны в конфигурации (build configuration).
- Сборка заканчивается, и ее результаты сохраняются в истории конфигурации.

Project

Один из основных концептов в TeamCity — проект, именно он содержит в себе так называемые конфигурации сборки (build configurations), которые и выполняют нужные нам задачи (запуск автотестов, линт-проверок и других суперважных и полезных вещей).

Проекты могут быть вложены друг в друга и принимать древовидную структуру.

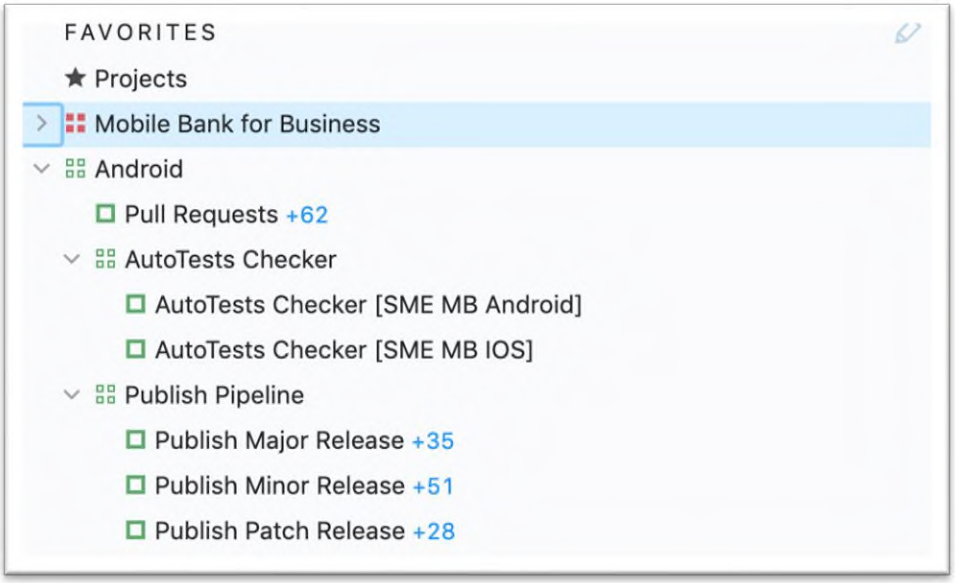


Здесь мы видим, что проект Android имеет вложенную в него конфигурацию Pull Requests, а также два проекта: AutoTests Checker и Publish Pipeline

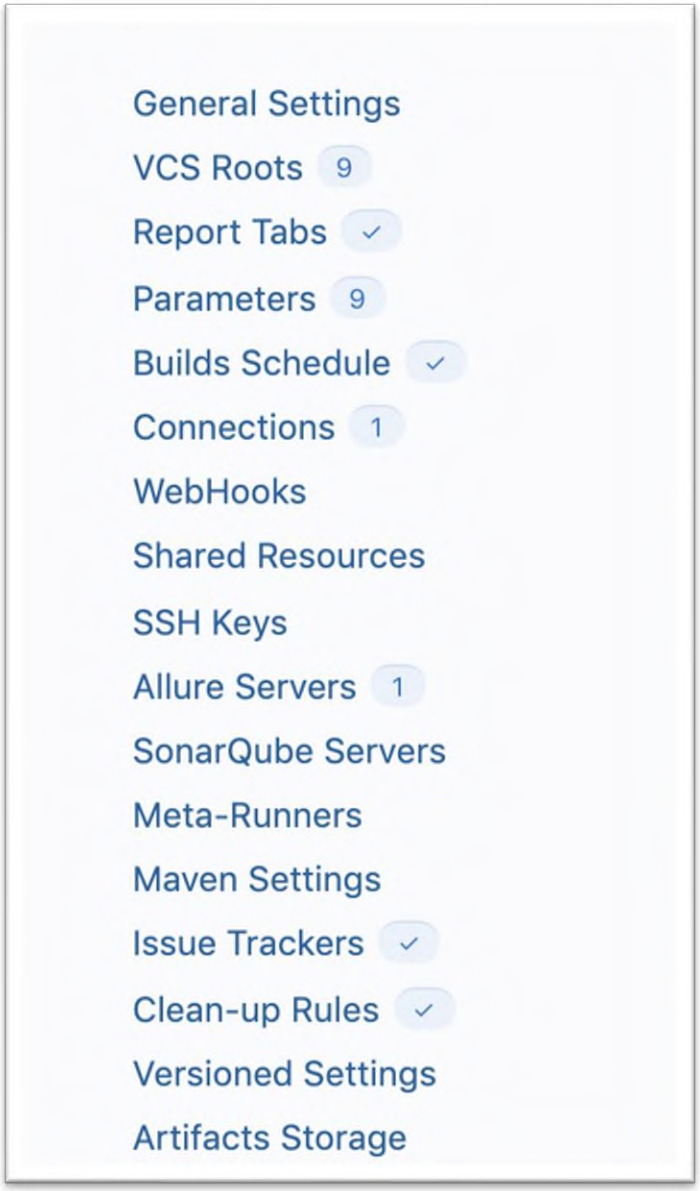
Здесь мы видим, что проект Android имеет вложенную в него конфигурацию Pull Requests, а также два проекта: AutoTests Checker и Publish Pipeline

Настройки проекта

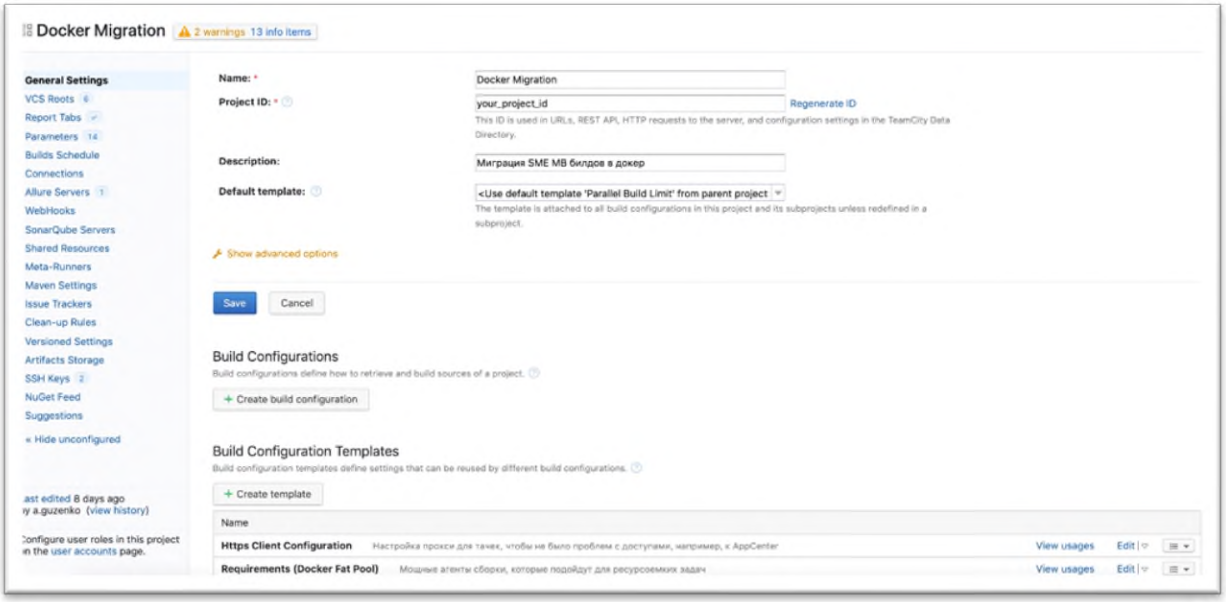
У проекта есть множество настроек и параметров, к которым можно перейти, кликнув на название проекта и нажав Edit project в правом верхнем углу. Не у всех есть доступ к настройкам, так что, если он вам нужен, обращайтесь к людям, отвечающим за CI/CD в вашей команде.



Давайте разберем самые важные настройки.



General Settings



Здесь мы видим следующее:

Name - Имя проекта

Project ID - Уникальный ID, который потом можно будет использовать в REST API

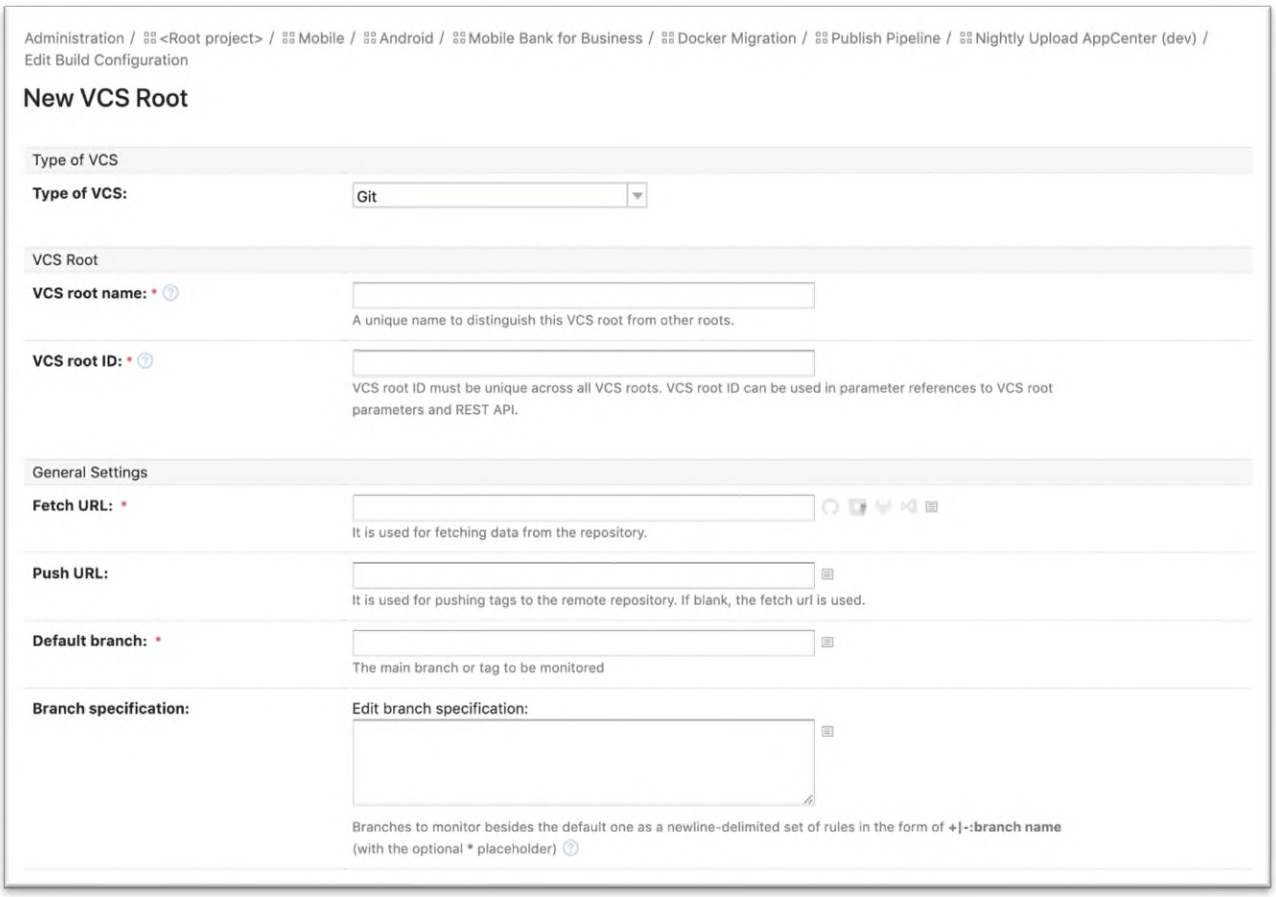
Description - Описание (опционально)

Все настройки и параметры (о них мы поговорим позже), которые были заданы в проекте, распространяются на все вложенные в этот проект подпроекты.

Под этими настройками мы видим список конфигураций сборки (build configurations) — о них тоже поговорим чуть позже.

VCS Roots

Здесь содержится набор настроек и правил, нужных для того, чтобы TeamCity мог подключиться к вашей системе контроля версий. При добавлении нового VCS Root он становится доступен внутри всех конфигураций, которые содержатся в проекте.



Для «соединения» TeamCity с системой контроля версий необходимо указать следующие параметры:

Type of VCS - Тип системы контроля версий (например, Subversion или Git)

VCS root name - Уникальное имя нового рута

VCS root ID - Уникальный ID рута, который затем может использоваться в REST API

Fetch/Push URL - URL до репозитория в формате https://, ssh://git@

Default branch - Основная ветка, изменения которой будет «отлавливать» TeamCity

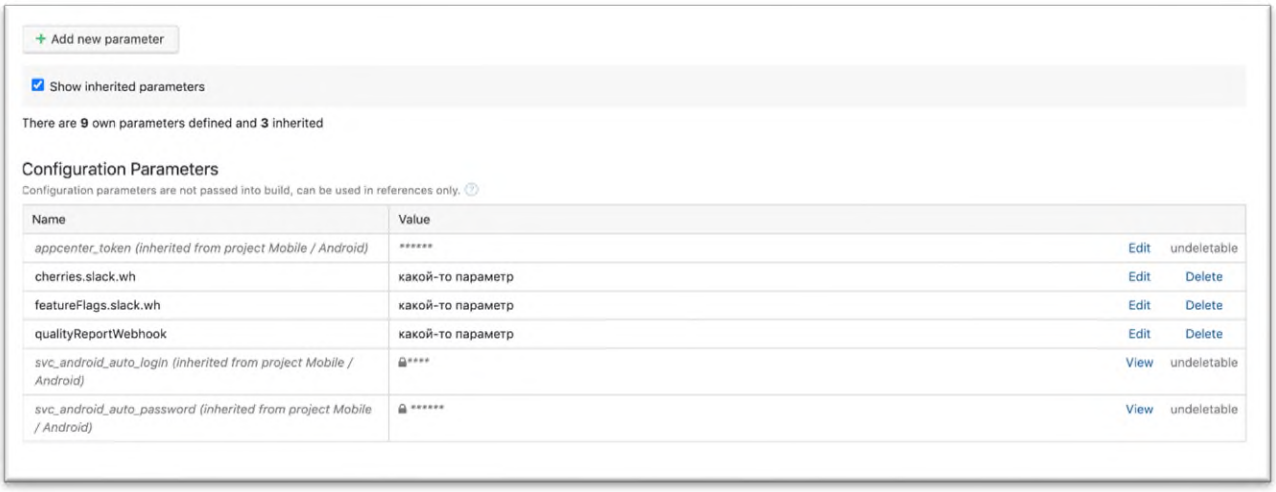
Далее нужно указать способ авторизации. Это может быть пароль или приватный ключ. На наших агентах установлены ключи, поэтому выбран default private key.

Крайне рекомендую на каждый репозиторий заводить всего лишь один VCS Root.

Parameters

Для изменения поведения билда можно использовать параметры TeamCity.

Некоторые из них позволяют передать настройки непосредственно в процесс, в котором идет билд. Другие служат для подстановки настроек, например в *build steps*.



Параметры можно использовать при сборке, они делятся на три категории:

Параметры окружения (env.) - Передаются в окружение процессу, в котором стартуется билд, и в дальнейшем их можно будет использовать в конфигурациях сборки

Системные параметры (system.) - Используются раннерами сборки (build runners) и автоматически передаются в них.

Например, в случае с Gradle любая переменная, указанная в system., будет передана в gradle.properties, но без префикса system.

Параметры конфигурации - Эти параметры не передаются в сборку, к ним можно только обращаться из конфигураций сборки. TeamCity делает

подстановку параметра через %param_name% в полях, отмеченных специальным значком

Итак, мы поняли, чем параметры отличаются друг от друга: системные передаются автоматически в систему сборки (gradle, например), параметры конфигурации можно только считывать, обратившись к ним по ссылке, а параметры окружения добавляются в окружение процесса и доступны на всех шагах сборки.

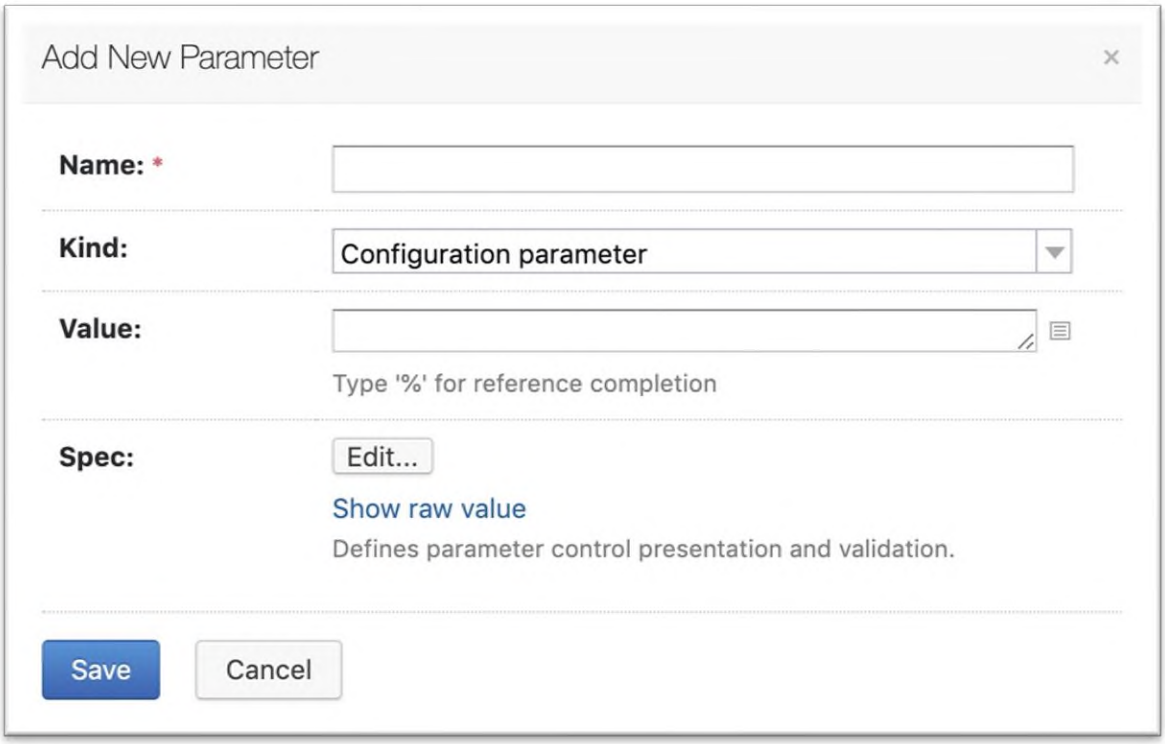
Теперь поговорим о создании параметров.

Создание параметров

Здесь все очень просто:



Нажимаем на **Add new** parameter на вкладке с параметрами. В появившемся окне видим несколько полей, которые необходимо заполнить:



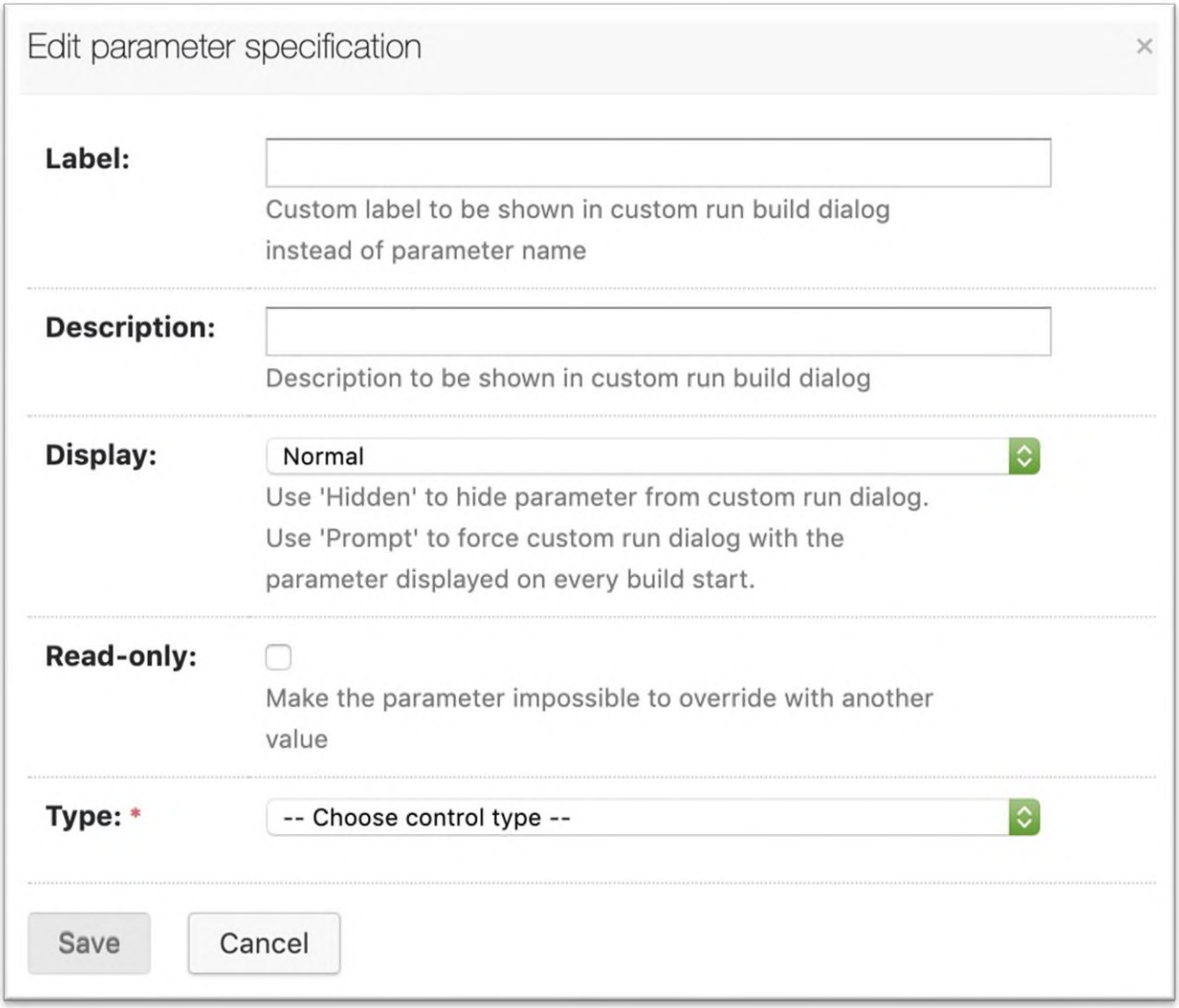
Name - Имя параметра

Kind - Тип параметра (environment, system, configuration)

Value - Значение параметра, в нем можно ссылаться на другие параметры в формате %parameter_name%

Spec - Самое интересное поле. В нем можно настроить тип параметра, его видимость, ограничить его редактирование, добавить к нему описание и сделать его обязательным для указания

Parameter spec



Здесь мы можем настроить следующее:

Label - Метка параметра, которая поясняет, зачем этот параметр вообще нужен

Description - Описание параметра, которое будет видно при мануальном запуске сборки

Display - Normal, Hidden (его не будет видно при мануальном запуске сборки), Prompt — этот параметр обязательно нужно будет указывать при запуске сборки

Read-only - Параметр нельзя будет изменять в течение сборки

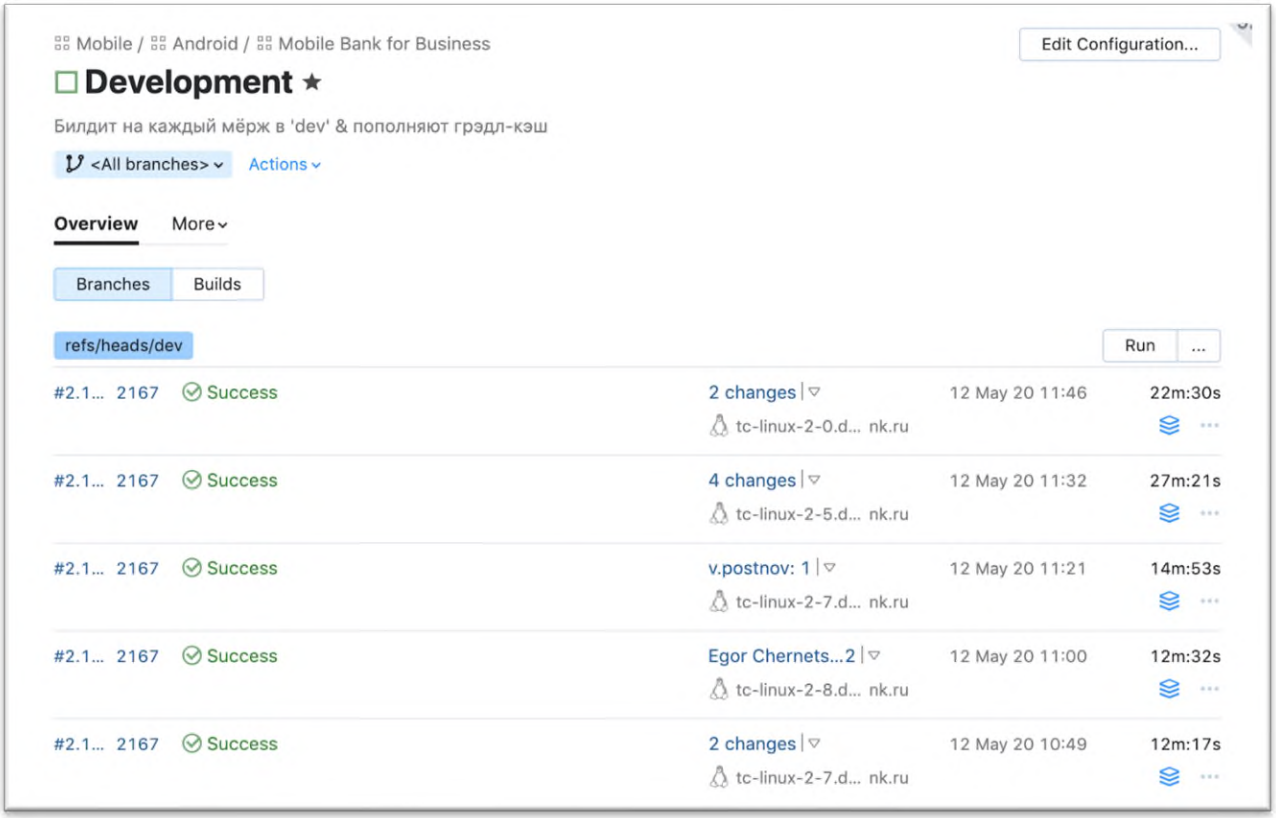
Type - Интересное поле: можно указать password — и тогда значение поля будет скрыто от всех пользователей. Полезно для всяких паролей и хуков

Build Configurations

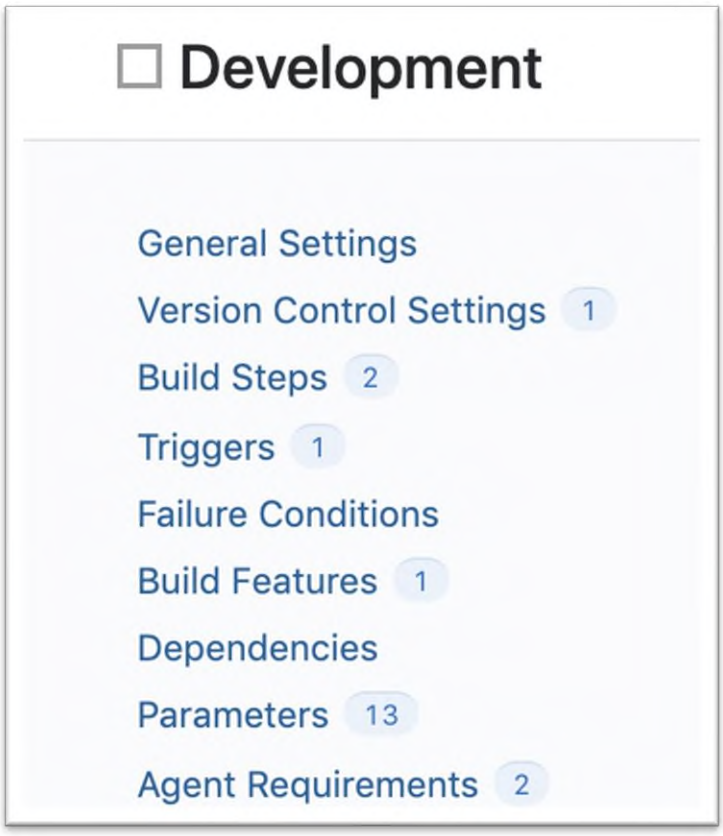
Мы разобрались, что такое проект, как его «соединять» с системой контроля версий, как добавлять параметры в сборку. И теперь, думаю, можно обсудить самое важное: как эту сборку вообще конфигурировать и запускать?

Итак, конфигурация сборки — это набор правил, которые определяют сценарий сборки. Каждая сборка, которая была выполнена по заданному сценарию, может иметь два статуса: Success либо Failed.

Правила, по которым сборка заканчивается с ошибкой, можно настроить — об этом мы поговорим позже. Все сборки в заданной конфигурации попадают в Build Configuration Home, где можно отслеживать их статус.

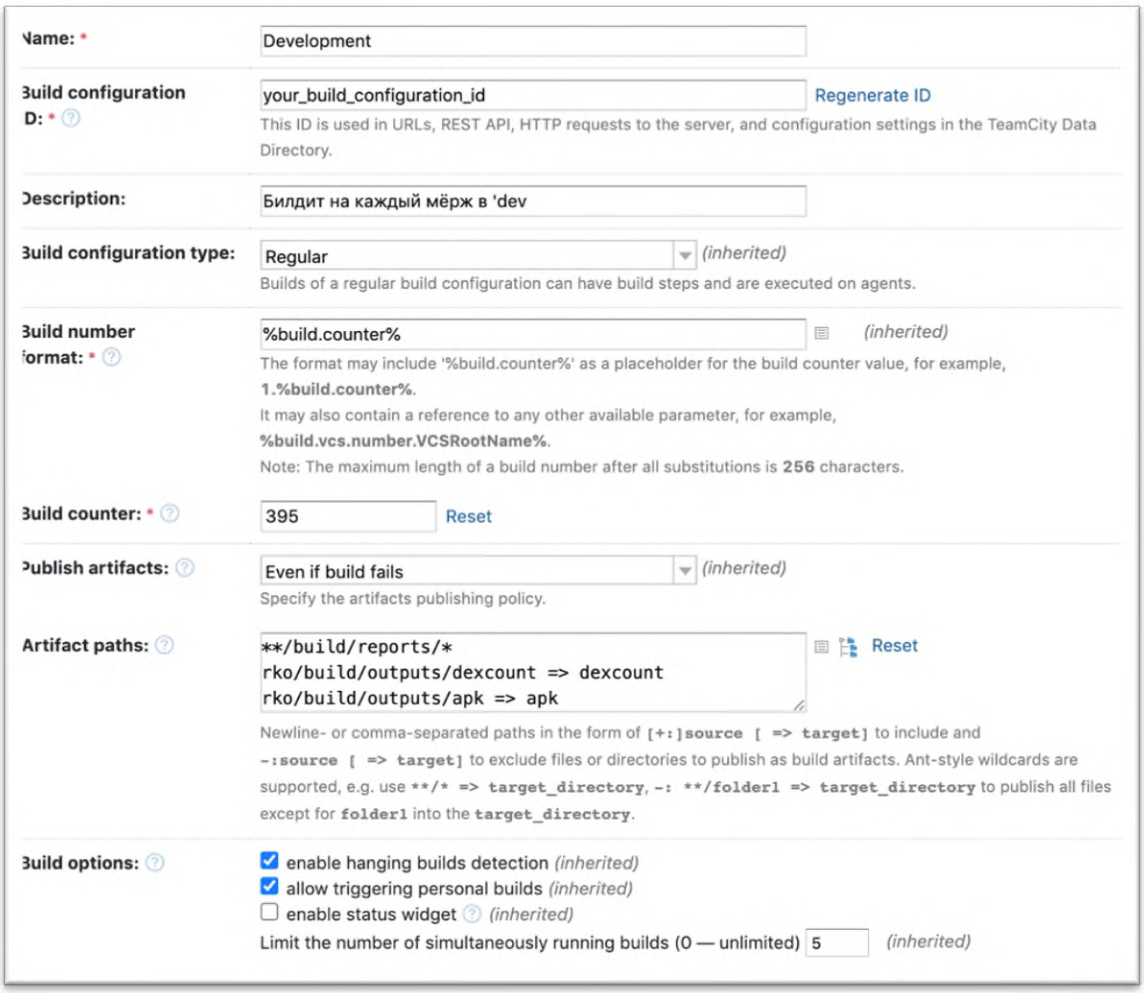


Как обычно это происходит в TeamCity — все нужно настраивать, и конфигурации сборки не являются исключением. Нажимаем на Edit Configuration в правом верхнем углу и видим список настроек:



Давайте подробнее остановимся на самых важных пунктах.

General Settings



Здесь мы видим список обязательных для заполнения полей, а именно:

Name - Уникальное имя конфигурации (например, Development, Upload To Testers, Nightly Builds)

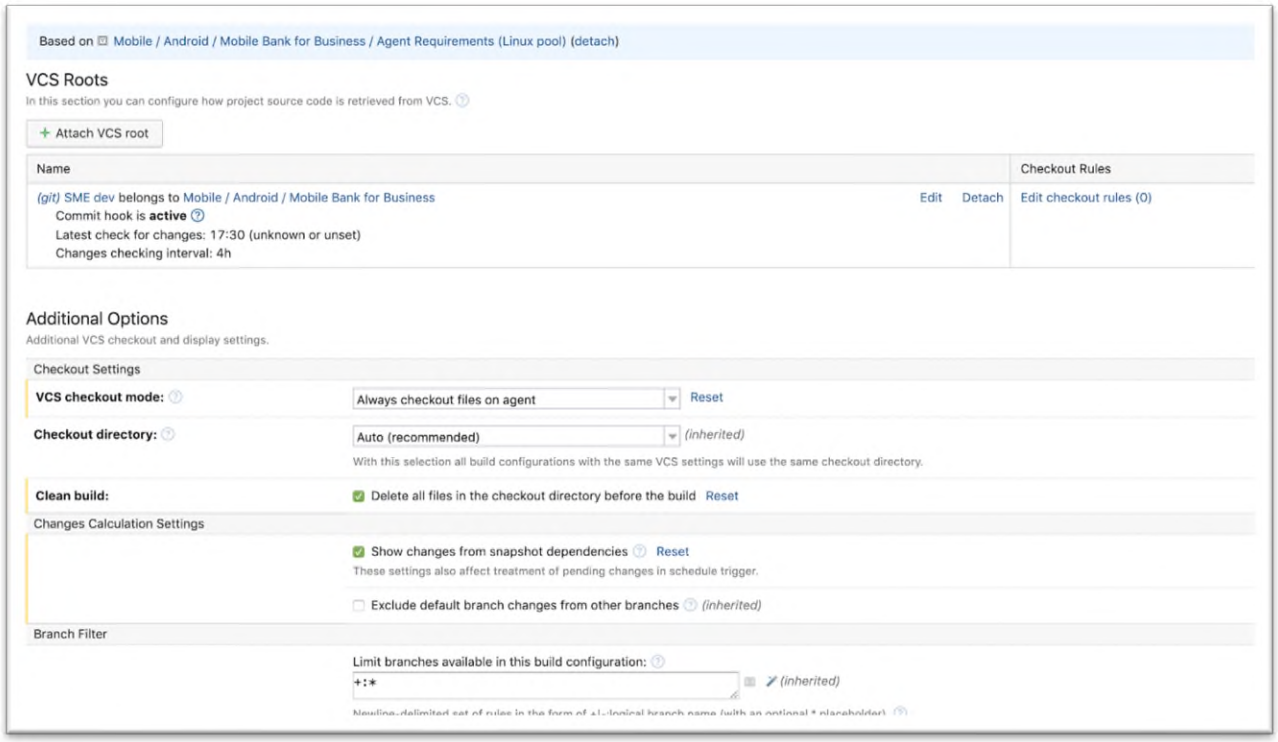
Build Configuration ID - Уникальный ID конфигурации, который можно будет использовать в REST API

Build number format - Каждой сборке TeamCity присваивает свой номер. В этом поле его можно отформатировать. Например, выводить не «1984», а «какая-нибудь-строка-1984»

Artifact paths - После каждой сборки TeamCity может оставлять артефакты (подробнее о них расскажу позже), здесь мы прописываем пути к ним.

Например, все, что будет лежать в */build/reports/, TeamCity заберет к себе и сохранит в конфигурации в качестве артефакта. А все, что лежит в rko/build/outputs/apk, TeamCity поместит в отдельную папку apk и также отдаст нам ее содержимое в качестве артефакта

Version Control Settings

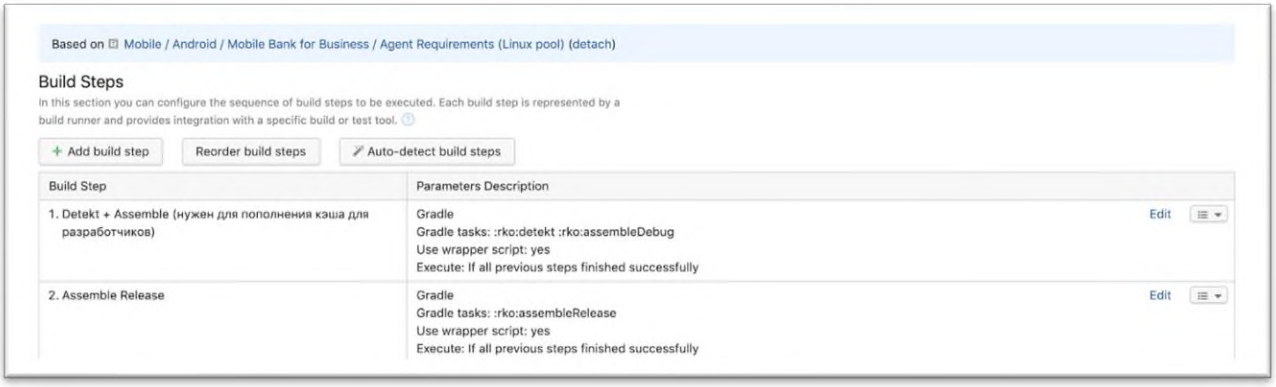


Конфигурации сборки тоже нужно «соединить» с системой контроля версий. Здесь все происходит по тем же правилам, что и в проекте.

На скриншоте видно, что VCS Root был унаследован (inherited) у того, что мы создали в настройках проекта (Mobile Bank For Business), к которому относится данная конфигурация.

Таким образом, эта конфигурация может взаимодействовать с веткой dev.

Build Steps

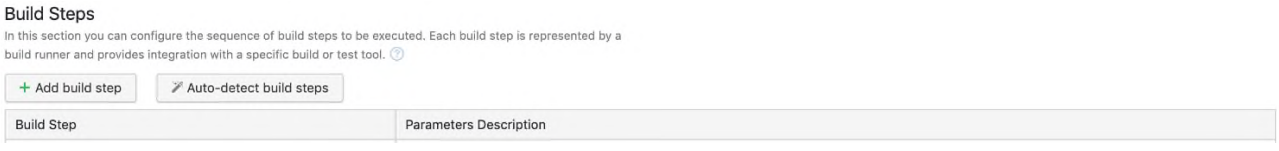


Вот мы и подобрались до самой интересной и важной части всех конфигураций сборки — к месту, в котором поочередно выполняются все нужные нам вещи (сборки приложений, прогон lint-проверок, прогоны тестов и так далее), — шагам сборки (build steps).

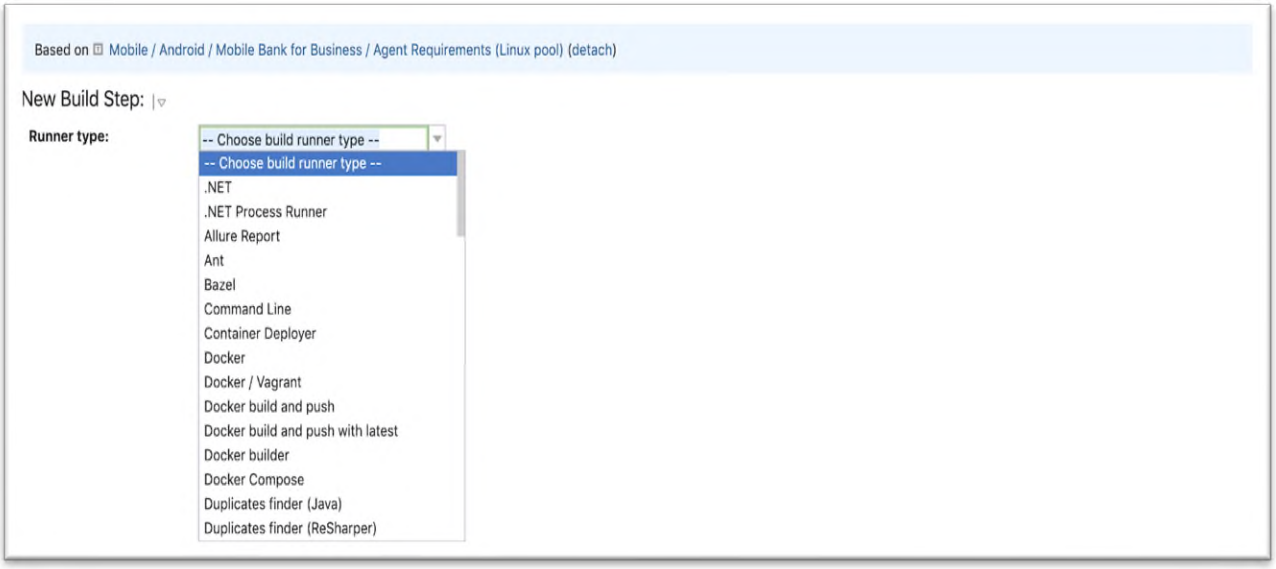
Итак, каждый шаг сборки представлен в виде раннера сборки (build runner), который предоставляет TeamCity интеграцию с инструментом для сборки (build tool), например с Gradle, MSBuild, командной строкой и так далее. То есть, по сути, каждый шаг сборки позволяет запускать .sh-скрипты (если в качестве раннера сборки была выбрана командная строка), Gradle-таски, Bazel-таски, .py скрипты и так далее.

Настройки каждого шага сборки индивидуальны в зависимости от раннера, поэтому давайте выберем самый популярный раннер — командную строку — и посмотрим, как создать с ним шаг сборки.

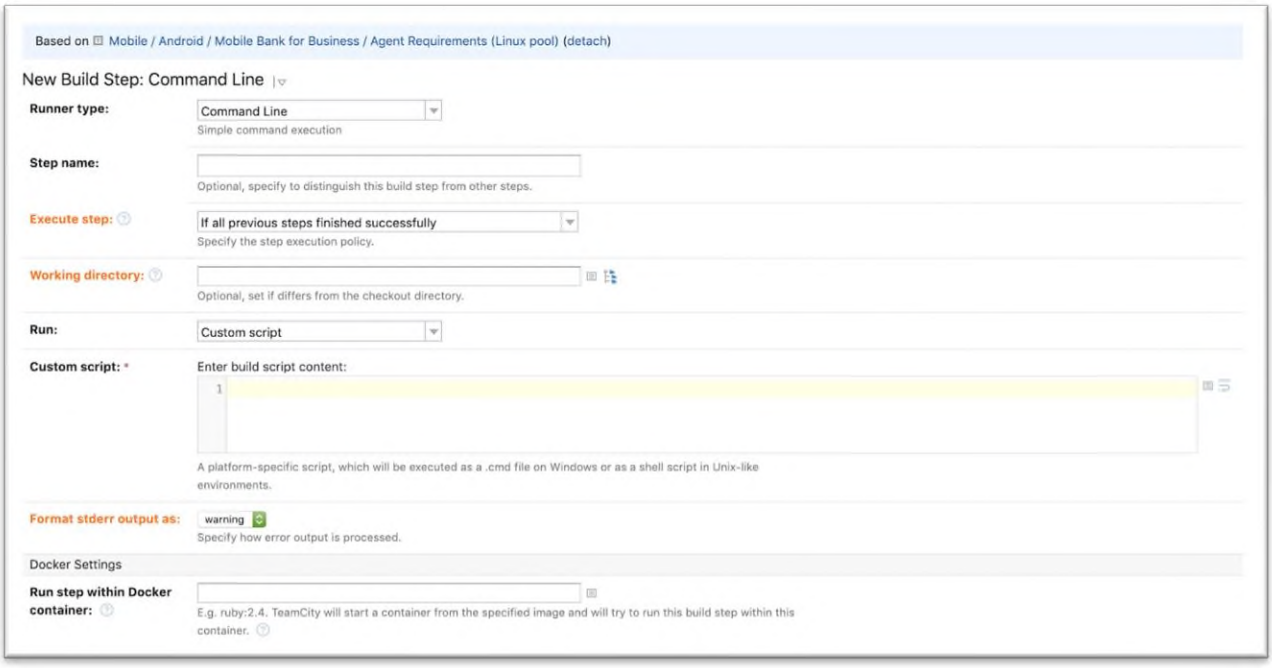
Нажимаем **Add build step** на вкладке Build Steps.



Далее видим следующую страницу:



В качестве Runner type выбираем Command Line:



Здесь мы видим, что нам нужно указать Step name — имя шага сборки, благодаря которому мы будем понимать, на каком этапе находится сборка. Давайте назовем его Hello, World.

Далее мы видим Execute step — это поле позволяет настроить условия, при которых данный шаг будет выполнен. По умолчанию там стоит *If all previous steps finished successfully*, однако нам есть из чего выбрать:



Далее мы видим настройку Run, в которой можно выбрать либо Custom Script (туда прямо можно запихнуть код на bash), либо Executable with parameters, сказав TeamCity, что мы хотим запустить скрипт, который уже лежит где-то в проекте, который он зачекаутил.

Давайте выберем Custom Script, а внутрь поместим следующее:

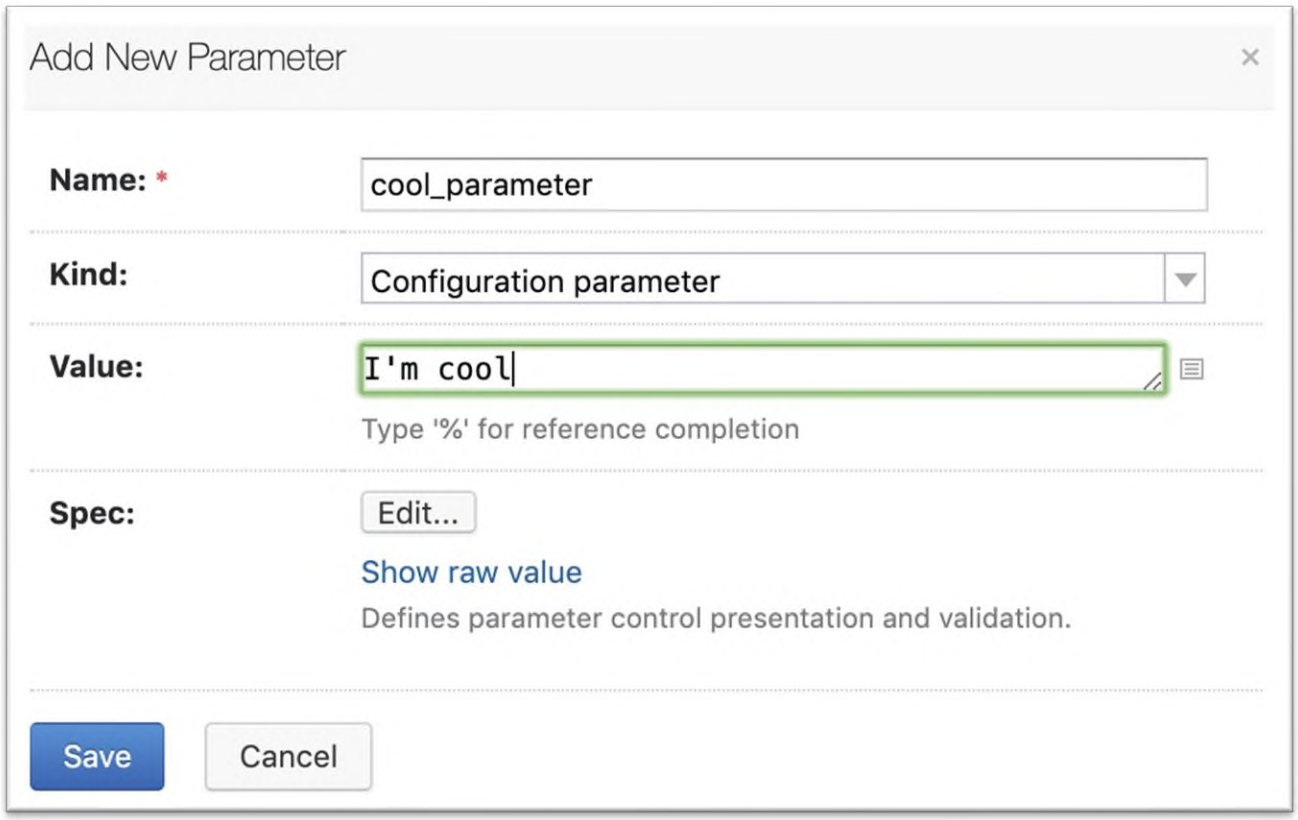
```
#!/bin/bash
echo "Hello, World!"
```

Нажимаем Save и видим, что наш шаг сборки сохранился. Теперь каждый раз, когда мы будем запускать конфигурацию сборки, в ее логах мы будем наблюдать милое “Hello, world!”, полученное из .sh-скрипта.

С другими типами раннеров все примерно так же.

Передача параметров в шаги сборки

Давайте создадим какой-нибудь параметр внутри конфигурации, которую мы рассматривали ранее, и попробуем передать его в шаг сборки. Открываем вкладку *Parameters* в настройках конфигурации, нажимаем **Add new parameter** и заполняем поля:



Так как мы не собираемся его нигде менять и запихивать в окружение, обозначим, что это параметр конфигурации, и назовем его cool_parameter.

Нажимаем **Save**, идем на вкладку Build Steps и нажимаем на шаг сборки *Hello, World*, который создали до этого, и добавляем в скрипт следующее:

```
#!/bin/bash
echo "Hello, World, %cool_parameter%"
```

Завернув название параметра в “%”, TeamCity сможет обращаться к нему и считывать.

Артефакты

Еще одна интересная фича в TeamCity — артефакты. Это все те файлы, которые были получены в результате сборки: файлы с логами, WAR-файлы и так далее.

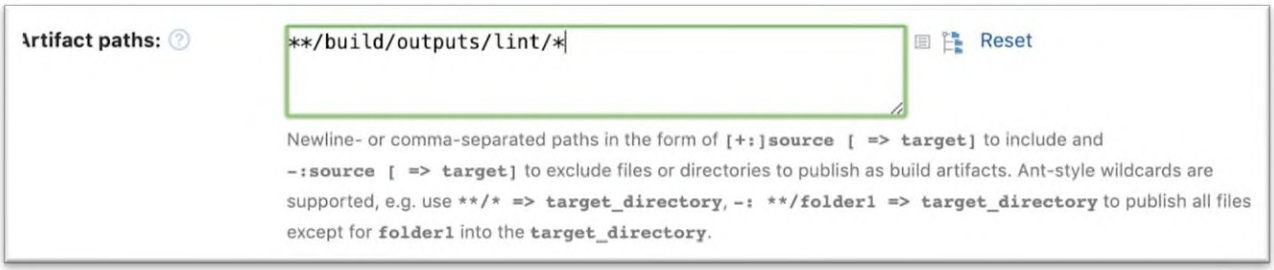
Ранее, настраивая конфигурацию сборки, мы уже видели поле, в котором нужно было прописать путь к артефактам. Давайте остановимся на них подробнее.



Представим такой кейс: в проекте есть некий скрипт, который прогоняет ряд lint-проверок. Результаты этих проверок сохраняются в какой-нибудь файл прямо в проекте, допустим в ../build/outputs/lint.

Мы не хотим прогонять lint вручную, потому что он занимает много времени, и решаем автоматизировать его запуск. Допустим, у нас уже создан проект и конфигурация сборки.

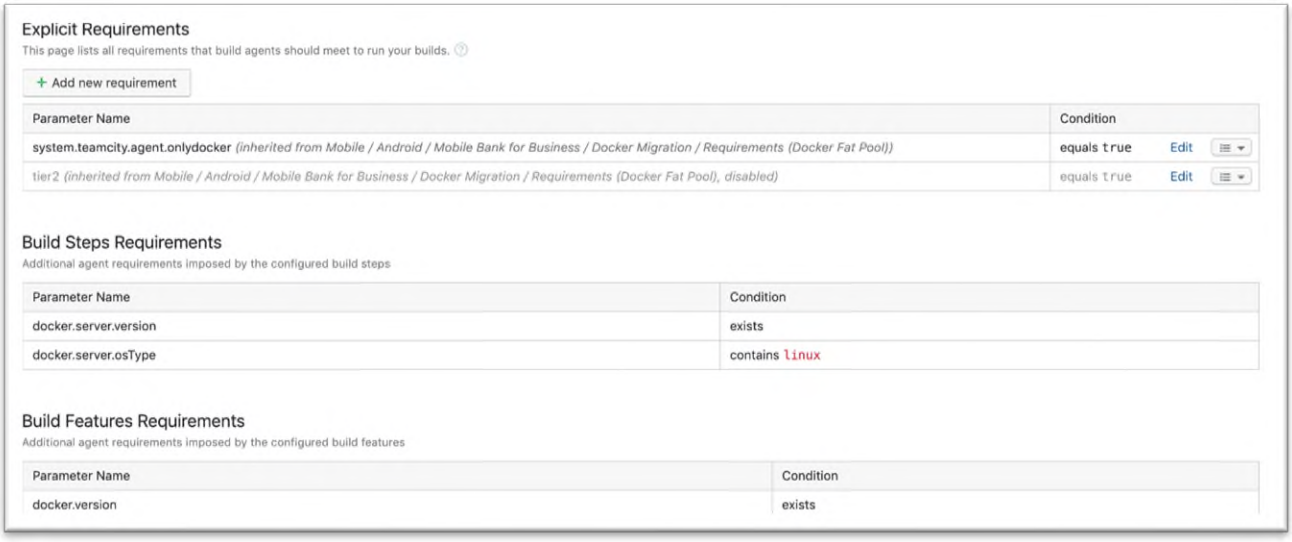
Открываем General Settings проекта и находим поле **Artifact paths**. Прописываем там следующее:



Таким образом, все, что будет лежать в папке lint, TeamCity добавит в артефакты сборки, которые будут доступны после того, как она завершится.

Далее нужно добавить шаг сборки, который будет прогонять lint. Настройки этого шага зависят от раннера.

Agent Requirements



Еще одна важная настройка каждой конфигурации — *Agent Requirements*.

С ее помощью мы можем сказать TeamCity, на каких агентах он может запускать конфигурацию. Например, чтобы наш агент мог собирать Android-проекты, мы можем потребовать, чтобы в переменных окружения этого агента обязательно был указан ANDROID_HOME.

Или, чтобы запускать bash-скрипты, мы можем потребовать, чтобы ОС агента была только Linux. Agent Requirements можно наследовать из проекта или для каждой конфигурации создавать свои.

Промежуточные итоги

Итак, давайте закрепим несколько важных моментов, которые мы только что прошли:

- TeamCity состоит из проектов, каждый из которых может содержать в себе вложенные проекты или конфигурации сборки.
- Если мы хотим что-то запустить (линт-проверки, тесты, какие-нибудь скрипты), нужно создать конфигурацию сборки.
- Каждая конфигурация состоит из шагов сборки (build steps).
- Шаги сборки настраиваются с помощью раннеров сборки (build runners).
- Взаимодействие с системой контроля версий происходит через VCS roots.
- Информацию об окружающем мире TeamCity забирает из параметров.
- Каждая сборка может оставлять после себя артефакты.

О Pipeline и Build Chains

Давайте теперь затронем несколько более сложные и важные вещи, освоив которые можно самостоятельно настраивать взаимодействие конфигураций сборки между разными проектами, а также работать с артефактами.

Итак, конфигурация сборки состоит из шагов сборки (build steps), которые выполняются в рамках одного билда. Ты можешь добавить в одну конфигурацию сколько угодно шагов, но это плохая практика. Пихать большое количество шагов в одну конфигурацию — большая ошибка, и если ты все-таки ее совершил, имеет смысл задуматься: может, одна конфигурация берет на себя слишком много всего?

Но что делать, если ты понял, что шагов действительно много, причем некоторые из них независимы друг от друга и могут запускаться параллельно, что существенно ускорило бы твой билд? К счастью, для решения этой проблемы TeamCity дает нам простой механизм прямо «из коробки» — build chains!

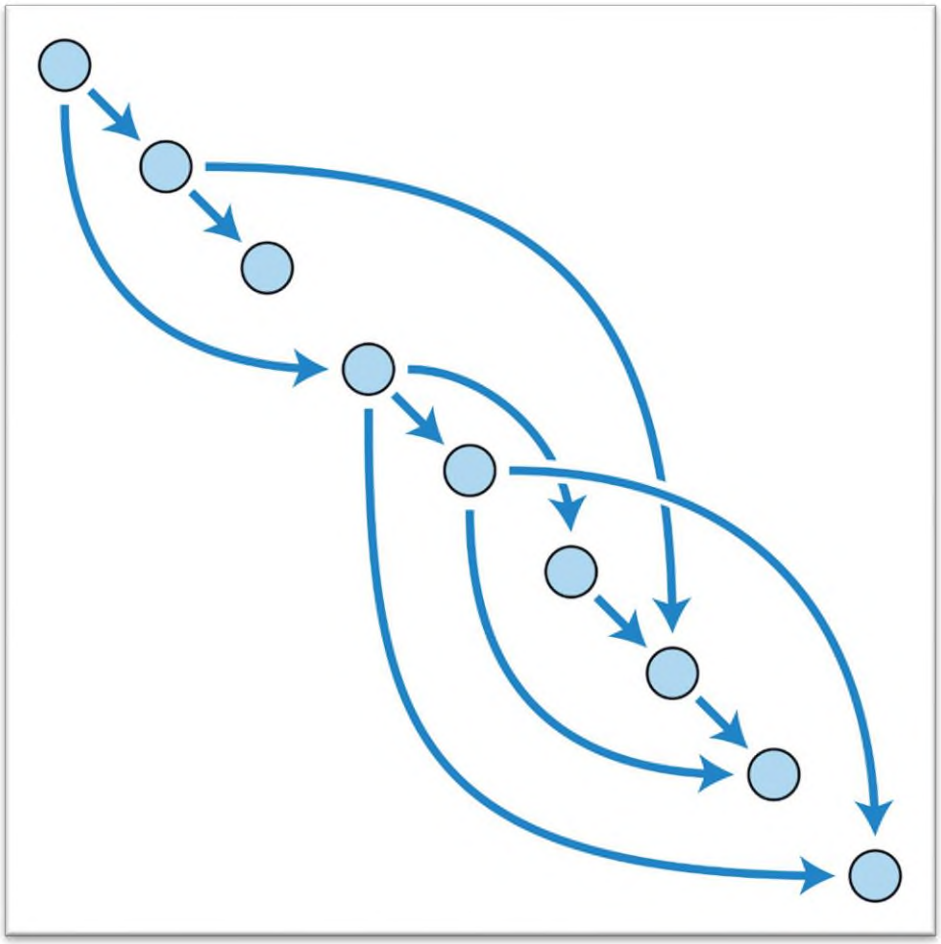
Давайте введем несколько других определений, которые помогут погрузиться в тему.

Pipeline

Здесь все довольно просто. *Пайплайн* — это некоторый (последовательный) набор действий, совершаемых во время сборки. И хотя

пайплайн — это классическое определение для процессов, происходящих на CI/CD, нигде — ни в документации TeamCity, ни в веб-интерфейсе — ты не встретишь это слово.

Дело в том, что у TC есть собственный термин — *build chain*, цепочка сборки. Цепочка сборки в TeamCity представляет собой направленный ациклический граф, поэтому пайплайн, подразумевающий последовательное выполнение шагов, — лишь частный случай цепочки сборки из TC.



Пример направленного ациклического графа

Но кроме того, что шаги в цепочках сборки не всегда могут быть последовательными, есть и другие отличия от стандартного пайплайна, о которых расскажу чуть позже.

Snapshot Dependencies

Еще один термин, который я введу, — *снепшот-зависимость*. Это как раз то, что позволяет нам связать между собой две конфигурации.

Устанавливая снепшот-зависимость для какой-нибудь сборки (например, сборки B) от другой сборки (сборки A), ты можешь быть уверен в

том, что сборка B запустится только после того, как будет запущена и завершена та сборка, от которой она зависит (сборка A).

Простыми словами: если в сборку B добавить снепшот-зависимость на сборку A, то B будет запускаться только после того, как запустится и завершится сборка A. Сборка B в этом случае называется зависимой.

Обрати также внимание на слово «снепшот» — это про синхронизацию исходников. При использовании таких зависимостей все они будут запускаться на одной ревизии VCS root (или на ревизиях, сделанных в одно время в разных VCS).

Snapshot dependencies формируют тот самый ациклический граф, о котором мы говорили ранее, а это не что иное, как цепочка сборки.

Build Chains

Наконец-то мы добрались до самих цепочек сборки — build chains.

Как я уже говорил, build chains позволяет разносить шаги сборки по разным конфигурациям, связывая их при помощи снепшот-зависимостей.

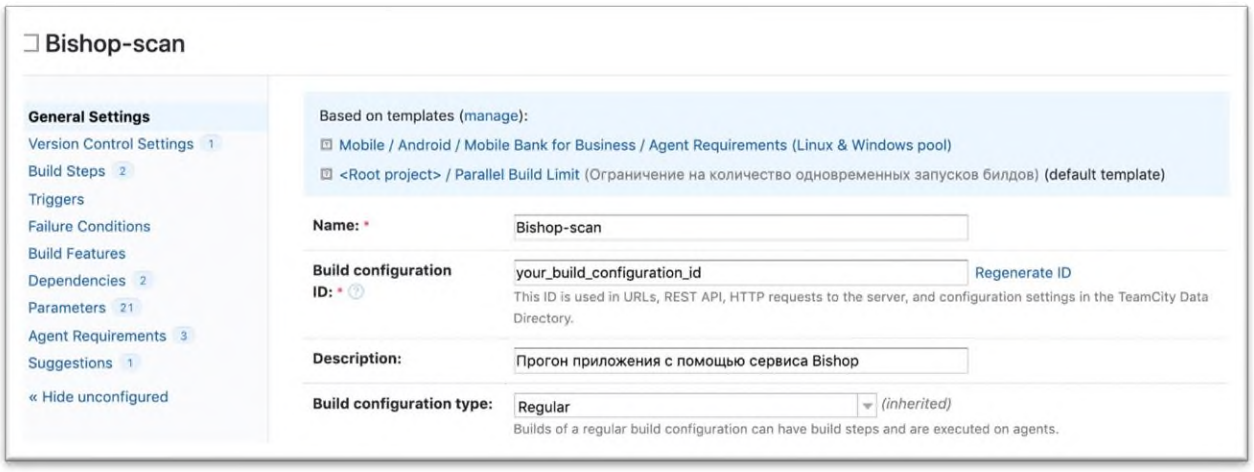
То, как TeamCity работает с цепочками сборки, открывает для нас несколько интересных фичей: параллельная сборка конфигураций (круто, да?), переиспользование результатов сборки, синхронизация конфигураций между разными репозиториями. Но самое главное: build chains существенно облегчают поддержку и развитие конфигураций.

Попрактикуемся?

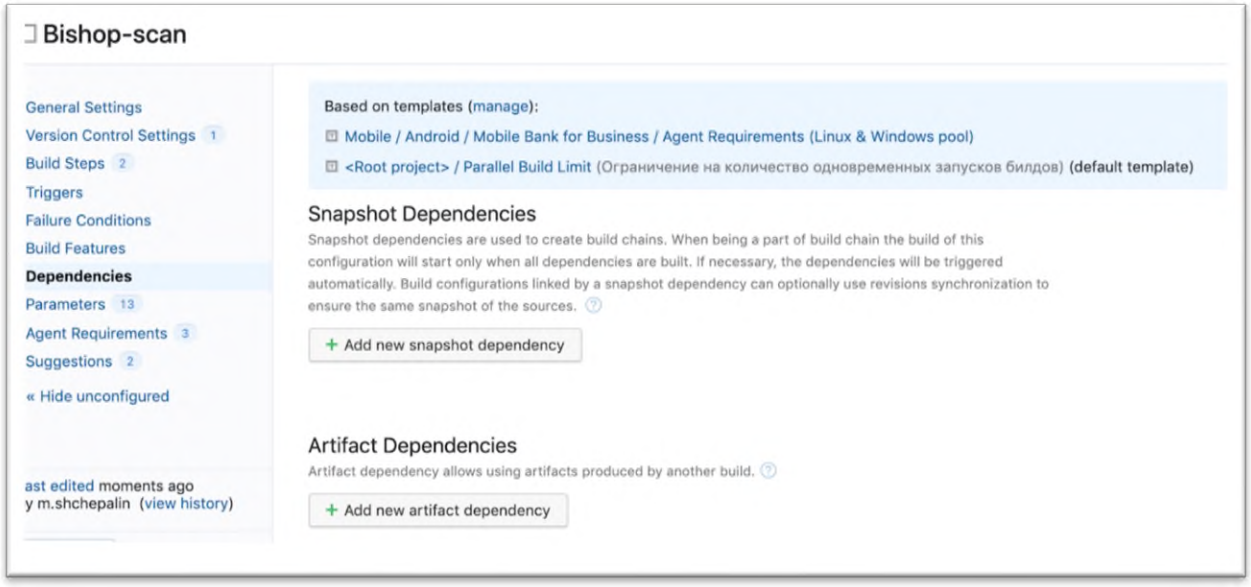
Чтобы создать цепочку сборки, достаточно выбрать две конфигурации сборки и настроить между ними снепшот-зависимость. В качестве примера возьмем две конфигурации:

- Одна будет собирать релизный apk-файл с android-приложением мобильного банка для юридических лиц.
- Вторая будет отправлять этот apk на сервер Bishop — сервиса для проверки android-приложений на наличие уязвимостей.

Итак, открываем конфигурацию сборки Bishop-scan:



Переходим на вкладку Dependencies и нажимаем кнопку **Add new snapshot dependency**:



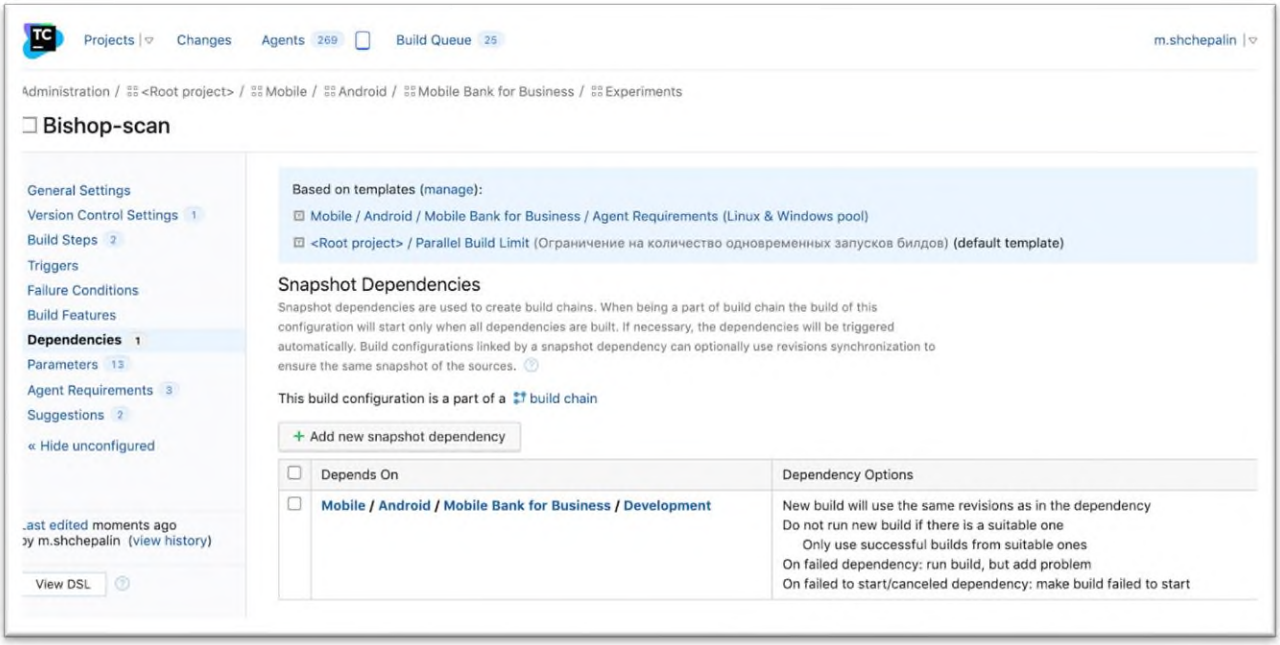
Как видим, кроме snapshot dependency можно добавить artifact dependency, но об этом мы поговорим несколько позже.

В появившемся окне нам предстоит настроить зависимость, указав несколько настроек:

Depend on	Выбираем ту конфигурацию, от которой будет зависеть текущая
Enforce revisions synchronization	Если включить эту опцию, все конфигурации будут работать, используя одни и те же сорсы, и никакие изменения в репозитории, возникшие на момент запуска зависимого билда, не будут браться в расчет
Do not run build if there is a suitable one	С этой опцией TeamCity не будет запускать зависимую сборку, если существует другая запущенная или

	завершенная подходящая (suitable) сборка. О том, что такое suitable build, мы поговорим чуть позже
Run build on the same agent	TeamCity будет собирать зависимую сборку на том же агенте, на котором собиралась основная сборка
Only use successful builds from suitable ones	Из suitable-билдов будут использоваться только те, которые были завершены без ошибок
On failed dependency/ On failed to start/canceled dependency	Здесь указываем, что делать, если зависимый билд упал: игнорировать, сообщать об ошибке, отменять билд

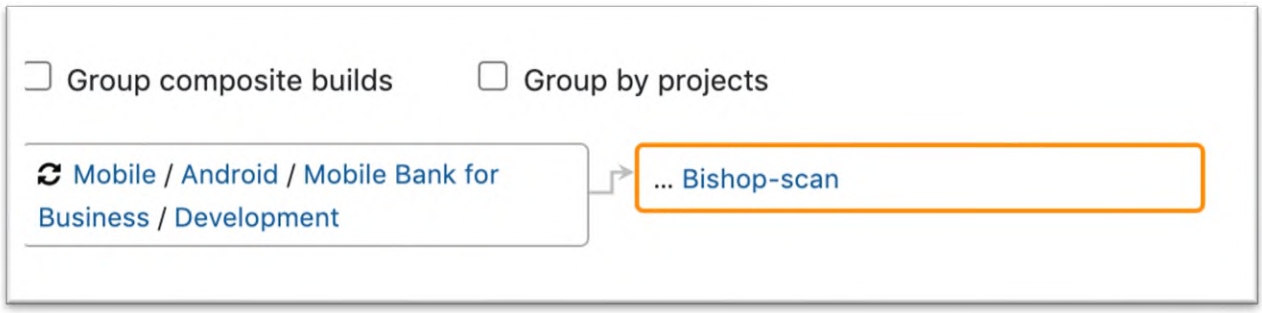
Нажимаем Save и вуаля — мы успешно построили свой первый build chain!



Полный граф цепочки сборки можно посмотреть, нажав на build chain тут:



Откроется отдельная вкладка, которая визуализирует для нас этот граф:



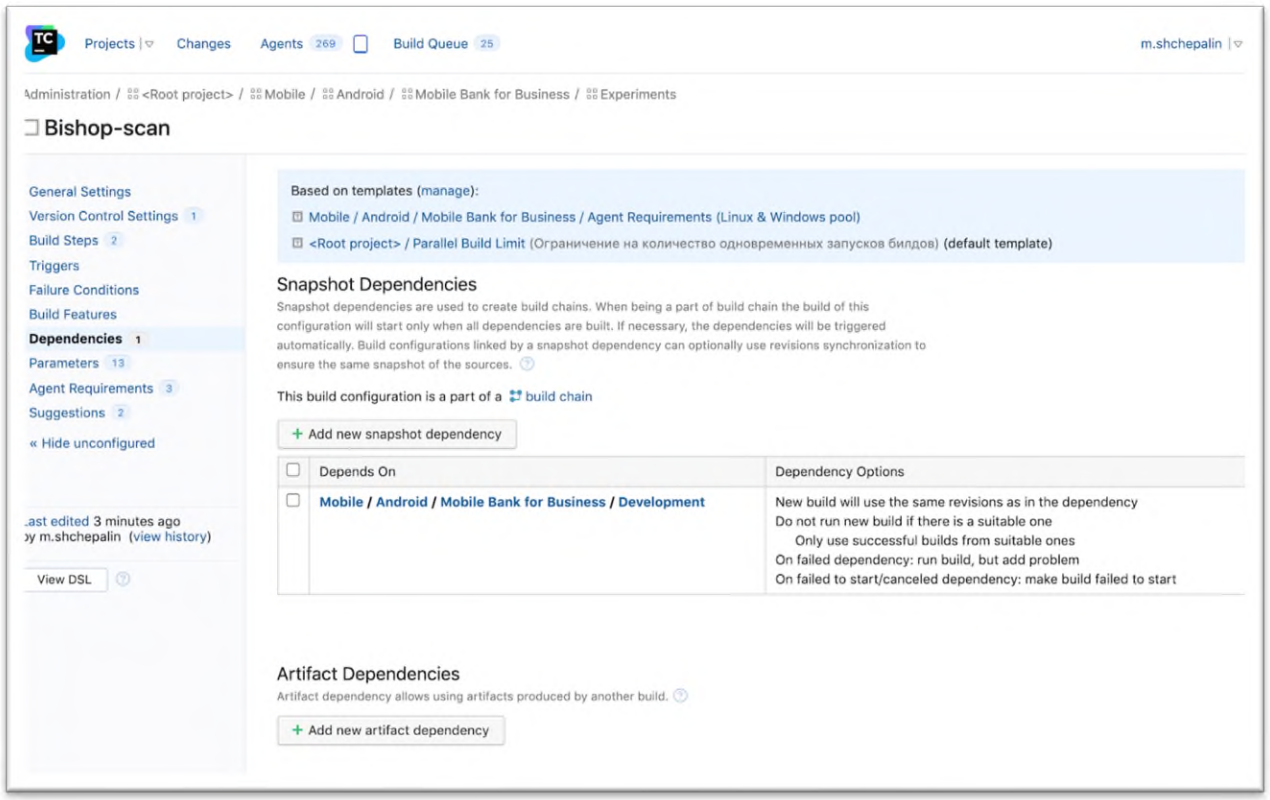
По графу видно, что перед тем, как мы запустим Bishop-scan, сначала будет собран Development, а только потом запустится Bishop-scan. Круто? Конечно!

Но вот проблема: Development собирает для нас артефакт. Как его использовать в Bishop-scan?

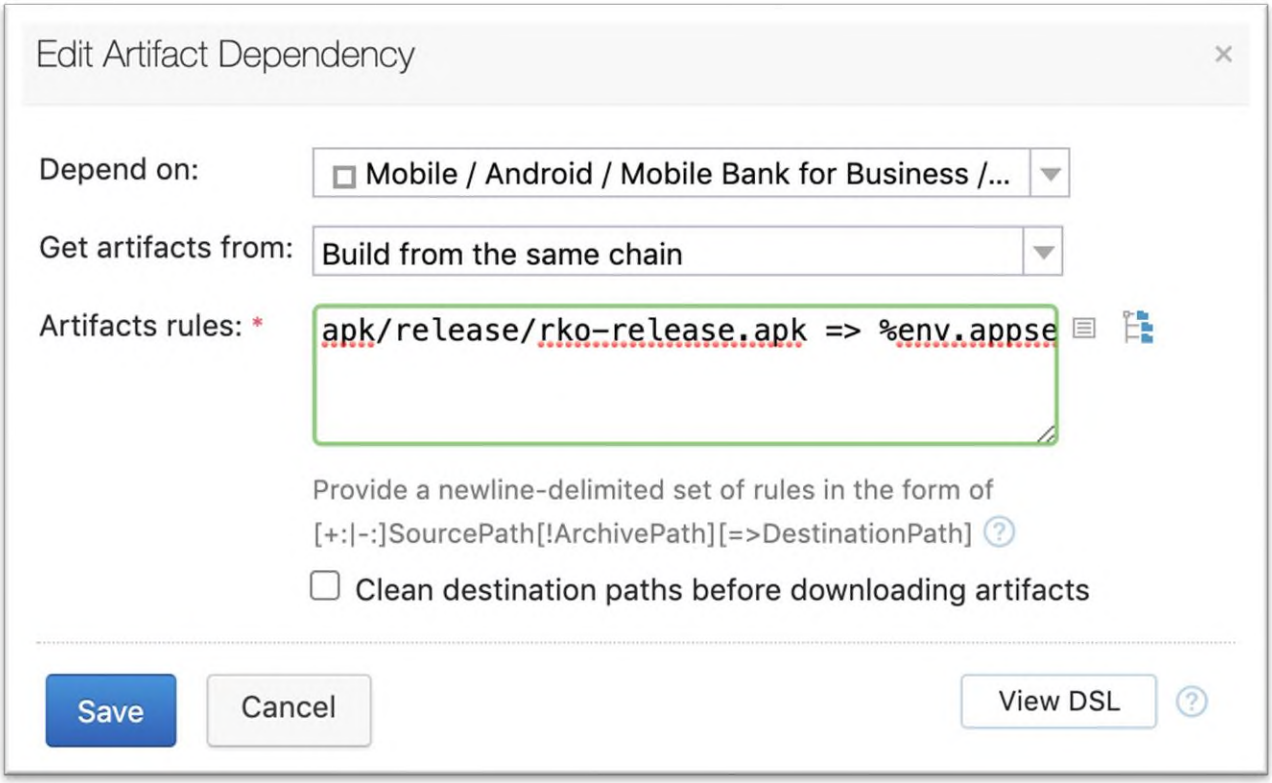
Artifact Dependencies

Все просто: кроме зависимости от конфигурации сборки мы можем зависеть и от артефактов, которые конфигурация сборки генерирует.

Конкретно в моем примере мне нужно будет забрать apk-файл, который был собран при сборке Development. Для этого снова идем во вкладку Dependencies конфигурации Bishop-scan:



Нажимаем на Add new artifact dependency:

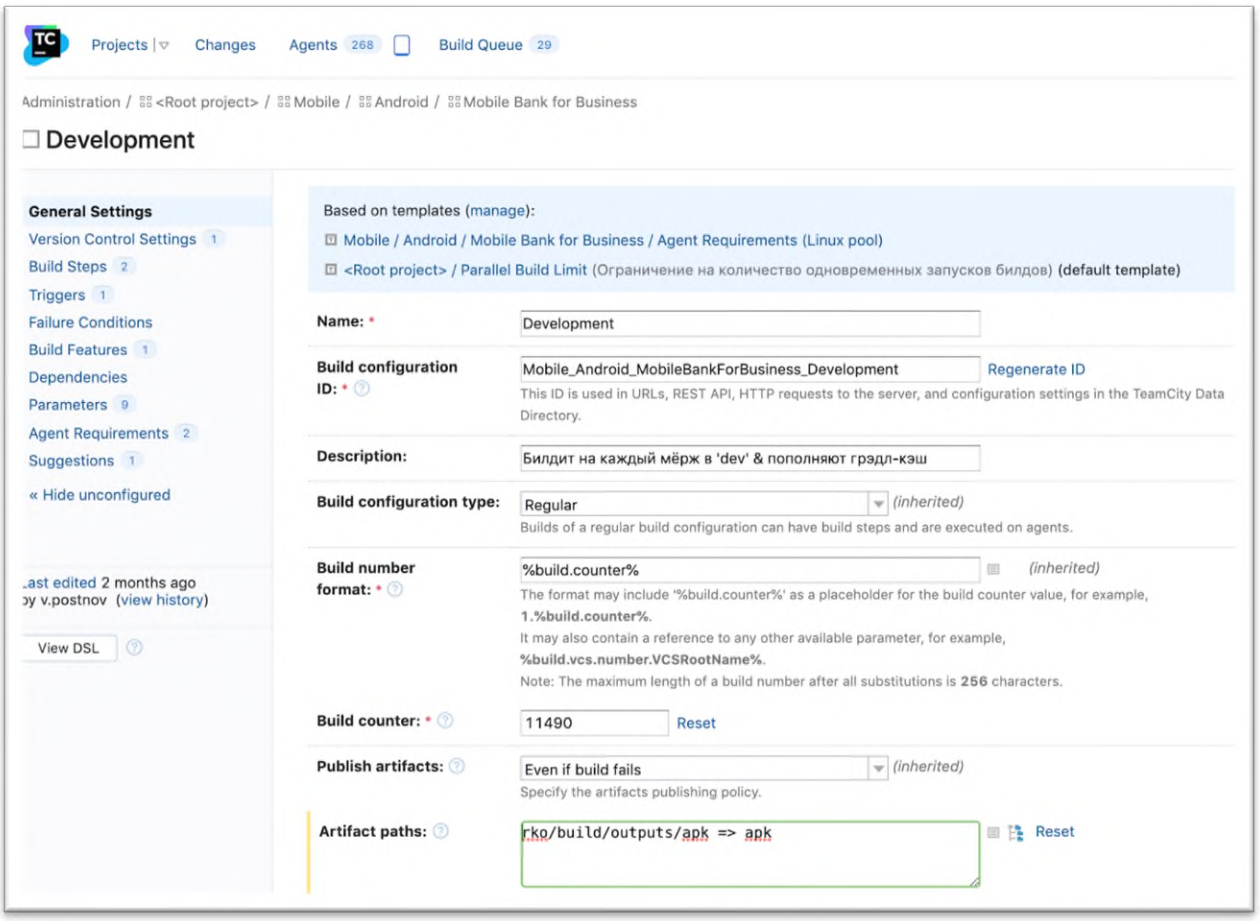


В появившемся окне указываем следующие настройки:

Depend on	У какой конфигурации мы будем забирать артефакт?
Get artifacts from	Откуда будем брать этот артефакт? Можем выбрать несколько вариантов: из этой же цепочки (тогда ждем, когда зависимый билд соберется); из последней успешной сборки, из последней сборки, из последней закрепленной сборки и из сборки с определенным номером
Artifact rules	Путь к артефакту. Очень удобно его указать, нажав на «веточку» справа. После «>» мы указали путь к environment-переменной, в которую положили этот apk

Стоит также обратить внимание на то, что зависимая конфигурация сборки (Development) должна объявить у себя в настройках, что этот артефакт она отдает.

Для этого идем в настройки Development и в графе Artifact Paths указываем путь к этому артефакту:



Теперь все, что лежало в rko/build/outputs/apk, будет перемещено в папку apk в артефактах, сгенерированных этой конфигурацией.

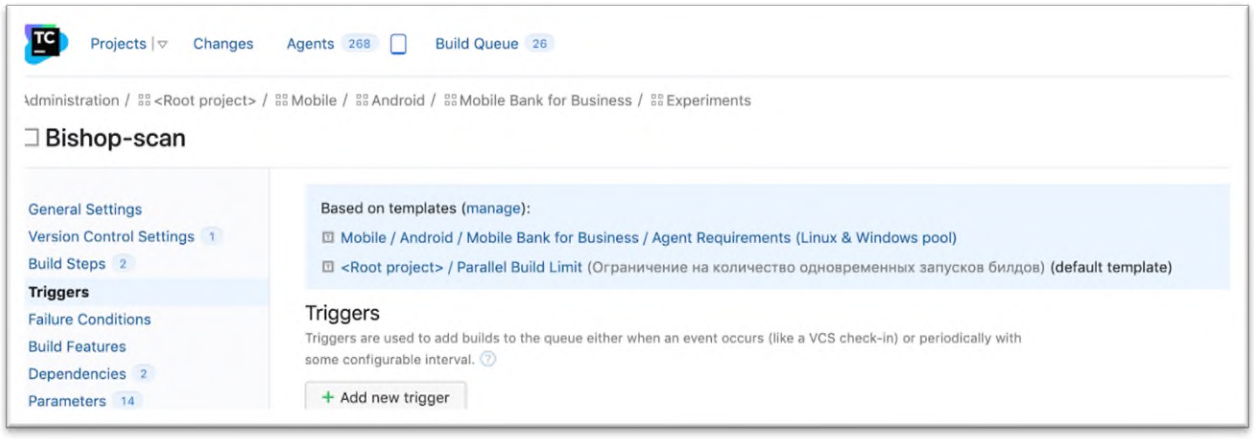
И все! Мы успешно связали между собой две конфигурации.

Теперь перед каждым запуском Bishop-scan сначала будет собираться Development, а только потом — Bishop-scan, использующий артефакт, полученный на предыдущем шаге.

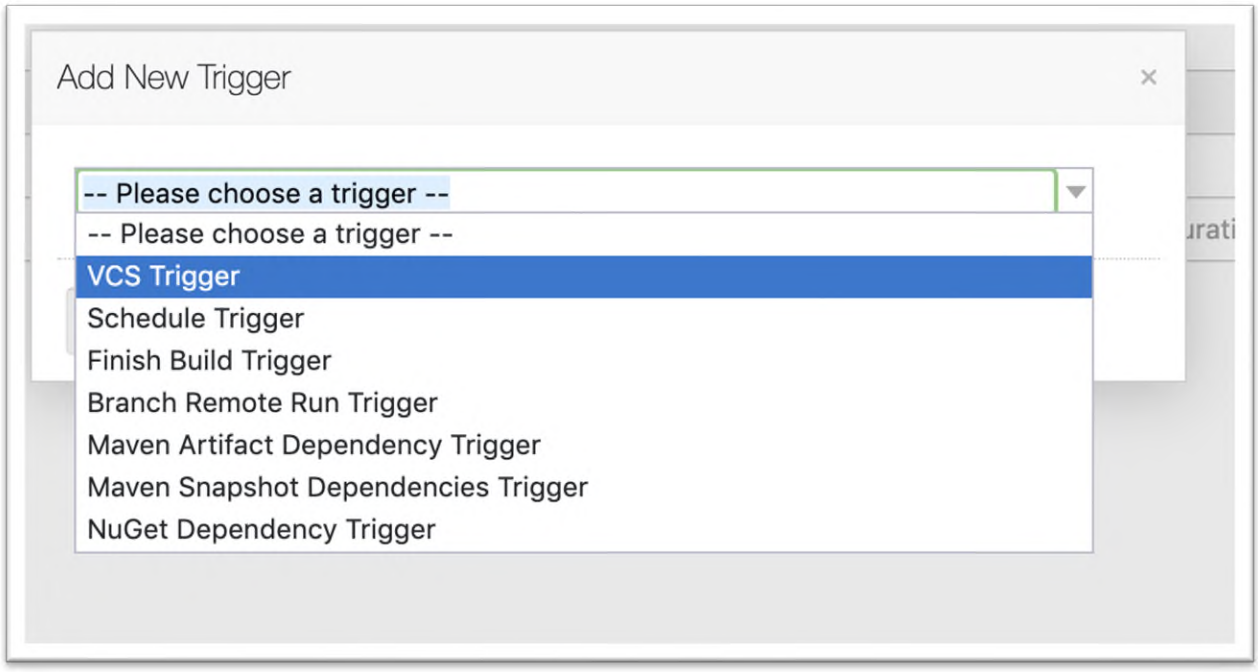
Но что, если мы хотим как-то по-особому настроить правила запуска конфигурации в цепочке сборки?

Build Triggers

Снова открываем настройки конфигурации Bishop-scan и нажимаем на вкладку Triggers:

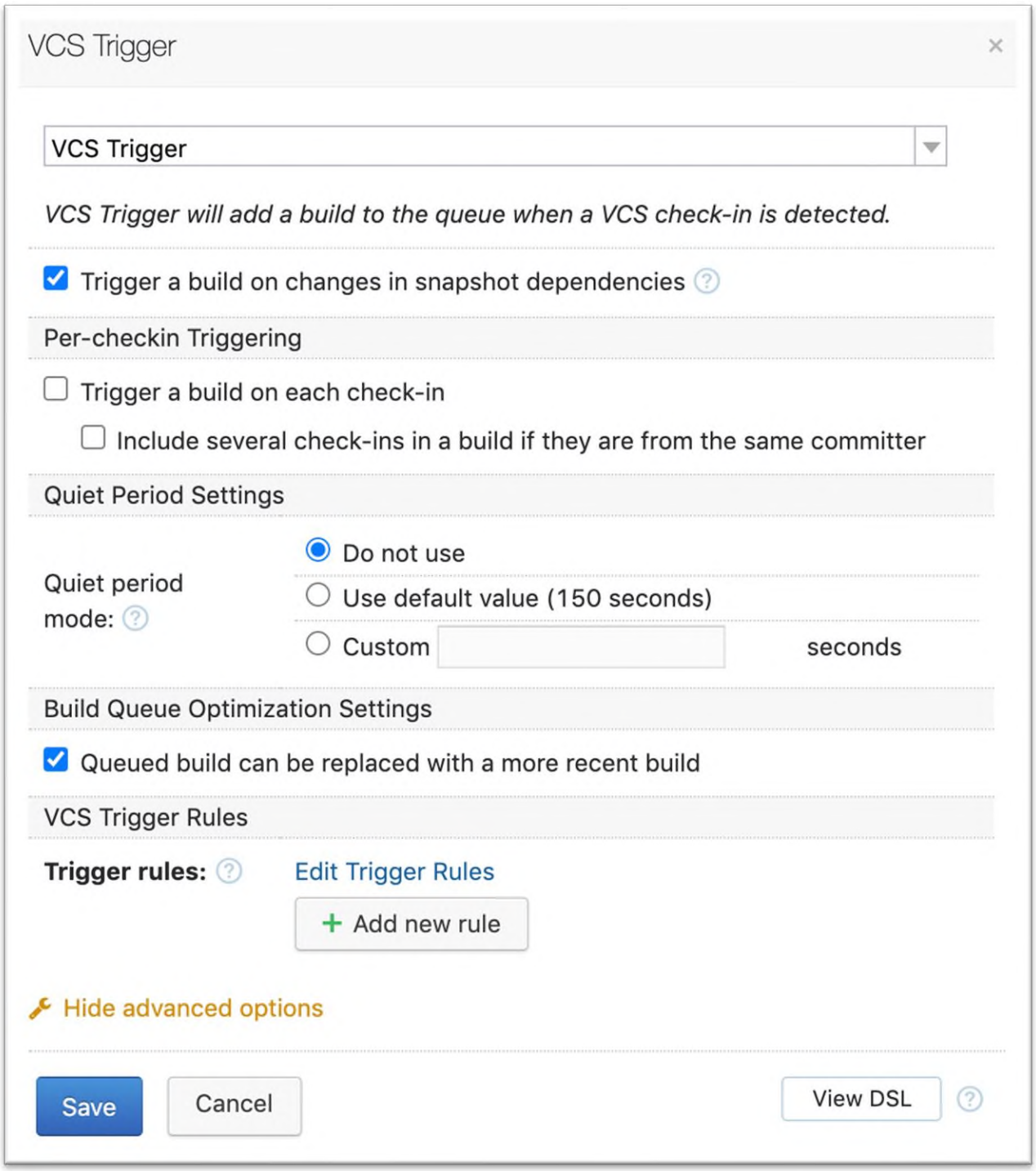


Здесь нажимаем Add new trigger и в появившемся окне видим следующее:



VCS Trigger	Запускаем билд только тогда, когда TeamCity обнаружил какое-то изменения в содержимом VCR Root'a
Schedule Trigger	Запускаем билд точно по расписанию (можно указать любые промежутки времени между билдами)
Finish Build Trigger	Запускаем билд только тогда, когда закончился билд какой-то определенной конфигурации (по сути, это частный случай нашей цепочки сборки, состоящей из двух конфигураций)

Допустим, мы хотим, чтобы сборка в Bishop отсылалась всегда, когда в VCS Root'e Development произошло какое-либо изменение. Для этого выбираем VCS Trigger и в появившемся окне нажимаем на Trigger a build on changes in snapshot dependencies:



В нашем примере две конфигурации сборки, однако понадобится лишь один VCS Trigger. Все потому, что мы можем сказать триггеру, чтобы он отслеживал изменения в каждой из конфигураций, когда находится в цепочке сборки.

Таким образом, если хотя бы в одной из них происходит изменение, TeamCity запускает целую цепочку. Ставить триггер нужно на последнюю конфигурацию цепочки, и тогда TC сам запустит все билды из нее.

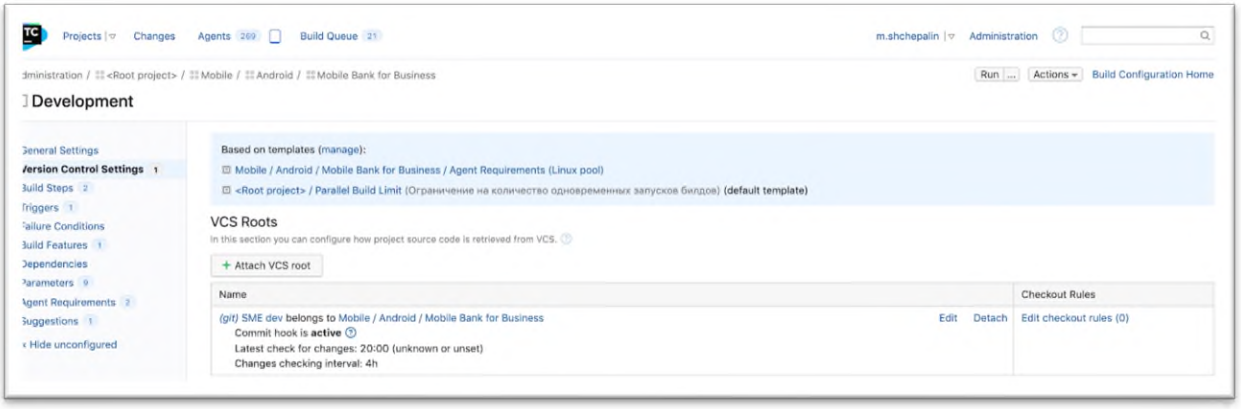
Checkout Rules

Вообще, существует довольно много способов оптимизировать пайплайн. Один из них — настроить правила, по которым TeamCity скачивает репозиторий.

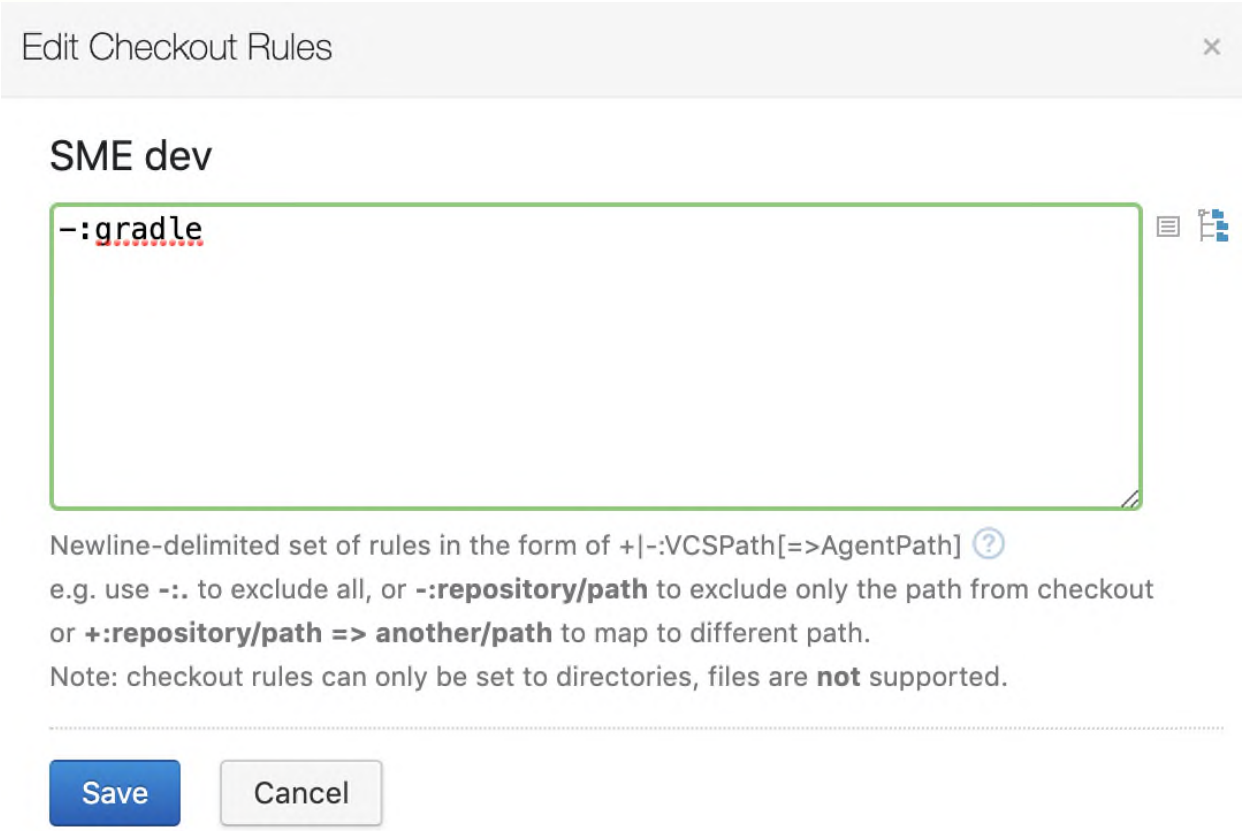
Вернемся к предыдущему примеру: мы сделали так, чтобы любое изменение в Development триггерило Bishop-scan. Но что, если мы не хотим, чтобы изменение в каком-нибудь .gradle-файле влекло за собой полную сборку .apk и анализ ее на серверах Bishop?

Для этого можно правильно настроить Checkout Rules в конфигурации Development.

Открываем Version Control Settings и жмем на Edit checkout rules:



В появившемся окне мы можем, например, исключить папку gradle, в которой лежат все наши .gradle файлы:



Для этого достаточно прописать -:gradle в правилах, и папка gradle будет игнорироваться при мониторинге VCS Root'a.

Переиспользование результатов сборки

Еще один способ оптимизировать пайплайн — переиспользование артефактов. Допустим, цепочка сборки успешно завершилась, и мы запустили ее снова (или она сама запустилась по какой-либо причине), но на этот раз TeamCity будет немного хитрее.

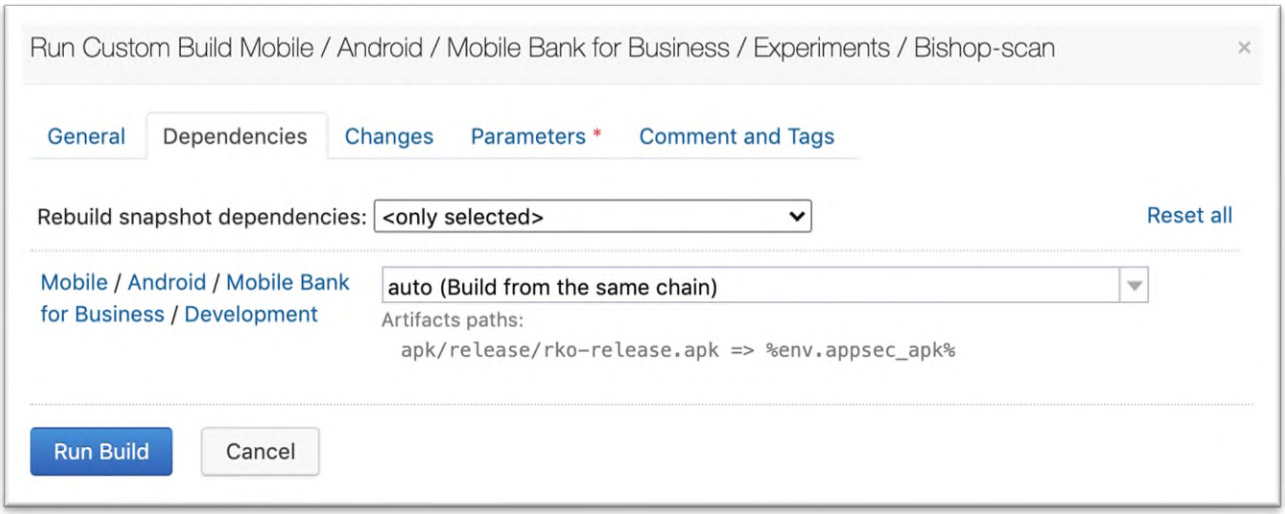
Что, если мы не будем пересобирать конфигурацию зависимости в цепочки, если никаких изменений в кодовой базе не было? Действительно, возвращаясь к нашему примеру, вряд ли нужно каждый раз собирать apk на Development, если на самом деле там ничего не изменилось.

К счастью, чтобы все работало именно так, не нужно ничего делать: все работает прямо «из коробки». Но мы можем ее отключить.

Заходим в Build Configuration Home нашего Bishop-scan:



Нажимаем на многоточие, которое находится рядом с **Run**. В появившемся окне переходим на вкладку Dependencies, нас интересует настройка Rebuild snapshot dependencies:



Нажимаем Run Build, и вуаля — теперь TC заново соберет артефакты в цепочке, даже если не было никаких изменений в кодовой базе.

Параллельный запуск конфигураций

И вот мы добрались до самой интересной и важной фичи, которую дарują нам цепочки сборки, — параллельный запуск.

Допустим, у нас в пайплайне добавилось еще две конфигурации, которые совсем не зависят друг от друга, а значит, могут выполняться параллельно. Обе эти конфигурации в зависимостях имеют снепшот-зависимость на Bishop-scan.

Помните, я говорил, что build chain — ациклический граф?

Так вот, сейчас самое время подумать, почему цепочки были спроектированы именно так: независимые ноды графа могут обрабатываться параллельно при наличии ресурсов для их обработки (то есть билд-агентов).

TeamCity сам позаботится о параллельной сборке независимых конфигураций и настраивать руками ничего не нужно.

Но что теперь делать с триггером сборки? В прошлый раз мы выставляли его на Bishop-scan, а сейчас что, придется добавить еще два — на каждую из новых конфигураций?

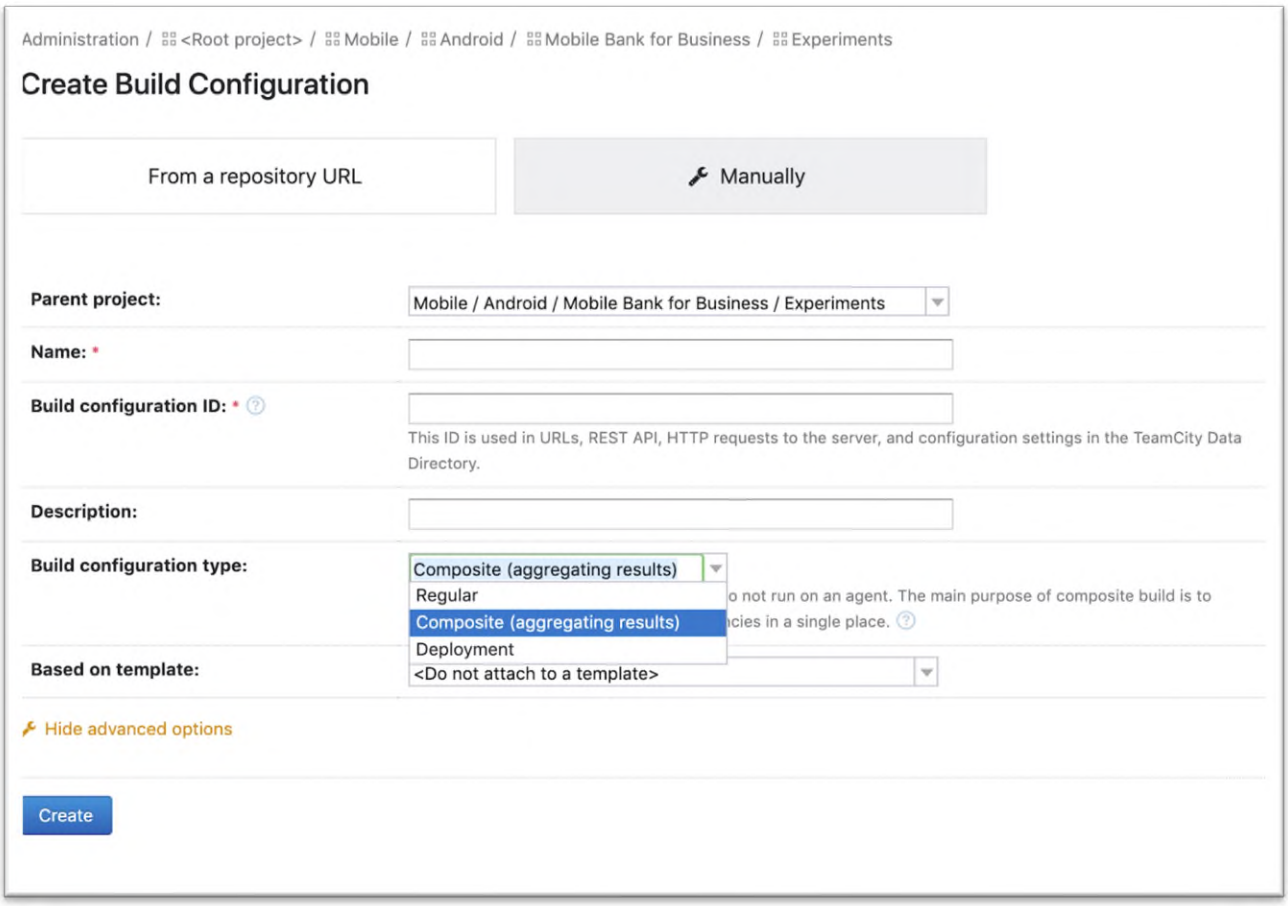
Можно, но есть более удобное решение: Composite Build.

Composite Build

Еще одна классная фича в TC — композитная конфигурация сборки.

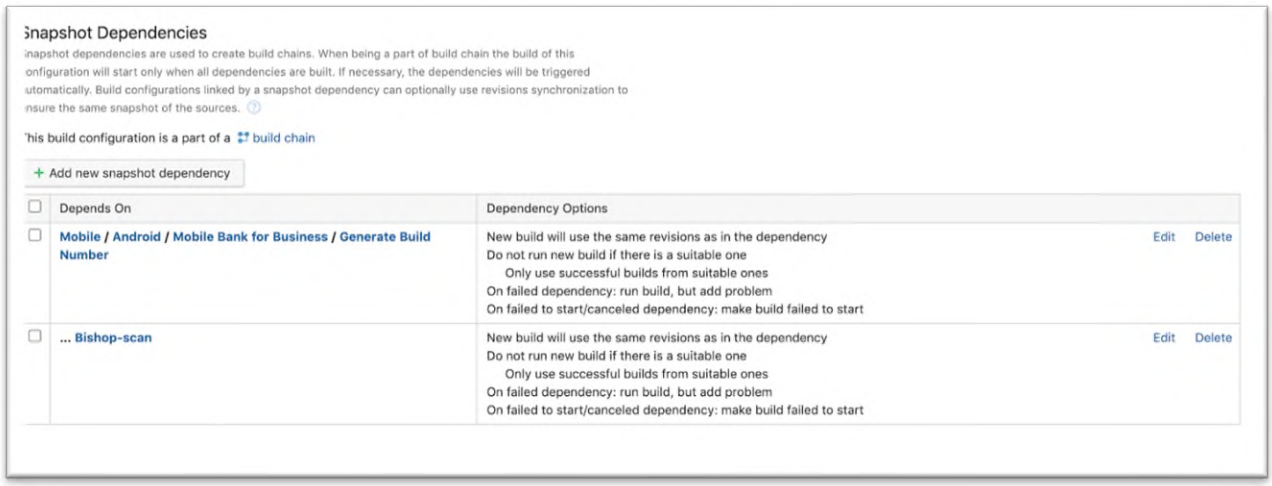
Ее задача заключается в том, чтобы агрегировать результаты сборки других конфигураций, от которых она зависит. А самое крутое, что, создавая такую конфигурацию, нам не нужно отводить место под еще один агент сборки — он просто не нужен.

Создание такой конфигурации отличается от обычной лишь одной настройкой:

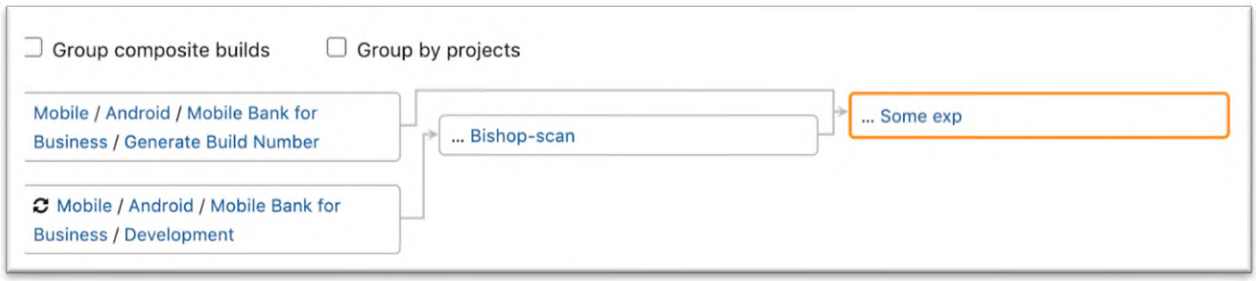


В поле **Build configuration type** необходимо выбрать **Composite**, а далее просто добавить снепшот-зависимости на наши новые конфигурации.

Далее добавляем две конфигурации в зависимости нашего композитного билда:



Добавляем *Generate Build Number* и *Bishop-scan* в зависимости и смотрим итоговый граф:



Теперь эта конфигурация будет частью нашей цепочки сборки, и мы сможем настроить лишь один триггер — на нее.

Заключение

Итак, мы с ознакомились с основными вещами, с которыми сталкивается каждый разработчик, настраивая CI/CD в своей команде:

- Настройка проектов
- Настройка конфигураций сборки
- Взаимодействие конфигураций сборки между собой
- Параллельный запуск конфигураций сборки.

4. Практическое знакомство с Gitlab

Gitlab — один из стандартных сервисов для работы с системой контроля версий *git*. Он поддерживает все необходимые для разработки функции: управление частными и публичными репозиториями, анализ кода на наличие ошибок, управление правами пользователей и доступом к репозиториям... Также Gitlab может быть установлен на локальный сервер, на котором ведётся разработка — это бывает полезно для организации приватного доступа к репозиторию или для более гибкой настройки прав доступа различных пользователей к проекту.

В этой инструкции мы расскажем, как настроить только что установленный Gitlab и познакомимся с некоторыми базовыми функциями: управление правами доступа к проектам, создание и удаление новых проектов, резервное копирование и восстановление проекта из бэкапа, а также импорт и экспорт проектов из публичных репозиториев.

Для начала работы нам понадобится подготовленный к работе сервер на Ubuntu или CentOS с предустановленным Gitlab.

Начало работы

Первый вход в Gitlab осуществляется от имени root-пользователя. Задать пароль root-пользователя можно при первом входе в Gitlab на странице, которую вы указывали в качестве домашней в конфигурационном файле/etc/gitlab/gitlab.rb.

Если при первом посещении страницы Gitlab не предлагает задать пароль root-пользователя, его можно задать вручную, через консоль. Для этого нужно использовать gitlab-rake — менеджер заданий Gitlab:

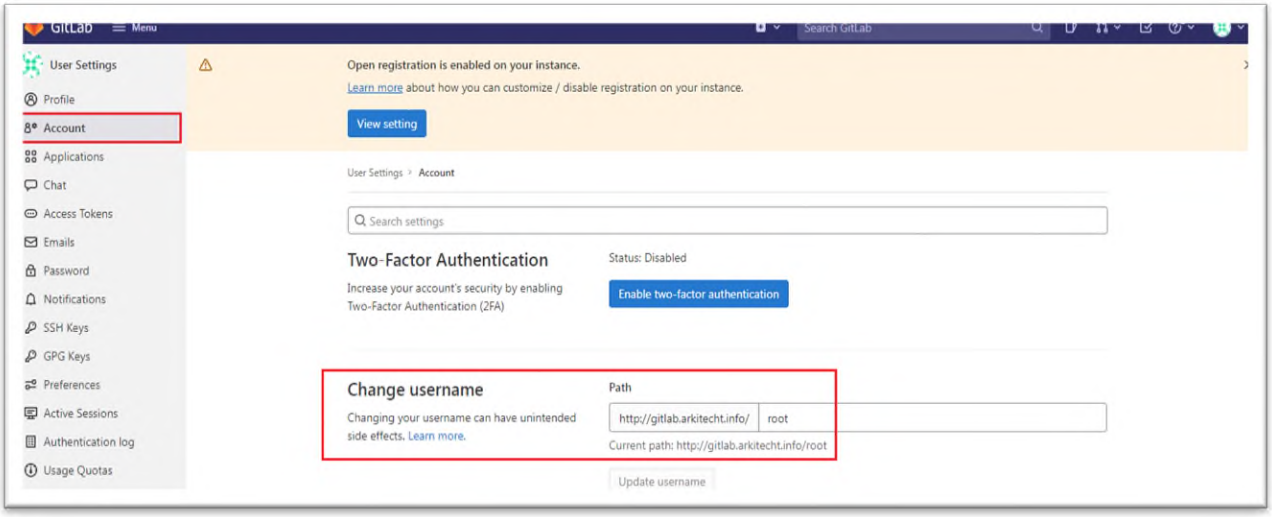
```
sudo gitlab-rake "gitlab:password:reset"
```

Запущенная задача попросит указать имя пользователя — указываем **root** — и задать новый пароль для него. После этого можно заходить в Gitlab через браузер.

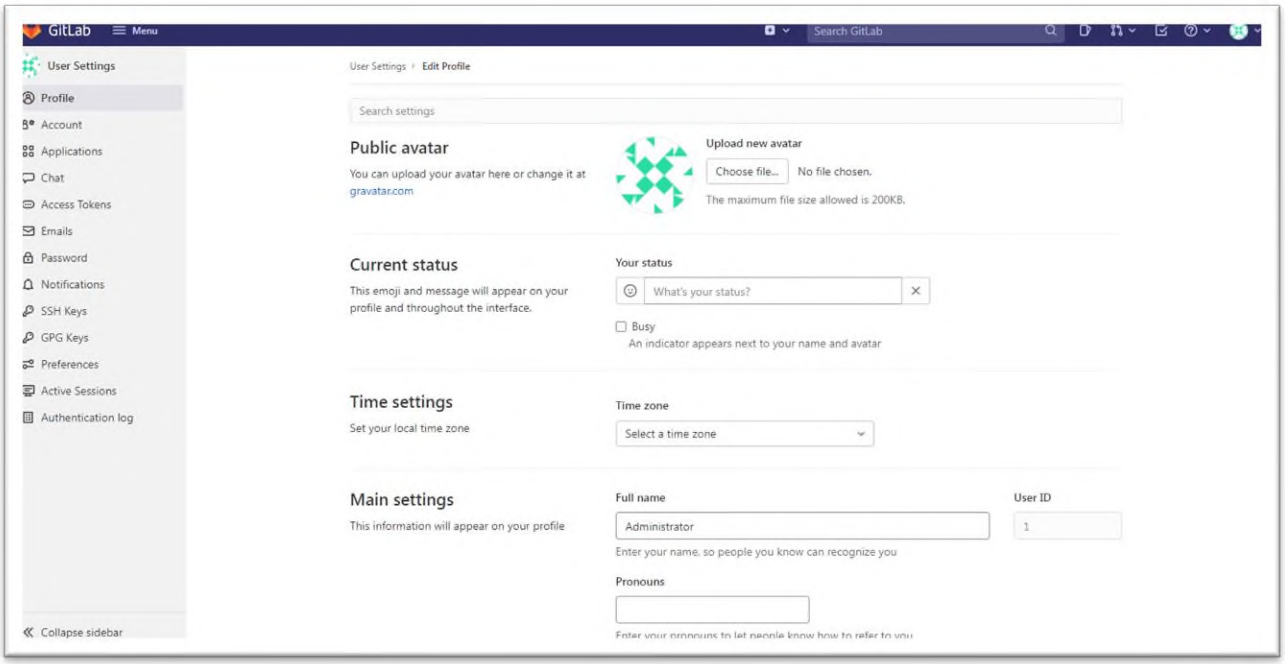
Первичная настройка

Заданное по умолчанию имя пользователя **root** редко меняется пользователями, поэтому является одним из потенциально уязвимых мест в безопасности нашего Gitlab. Сменим имя root-пользователя!

Для этого в разделе меню «User Settings» перейдём в раздел «Account» и в поле «Change Username» укажем новое имя. Важно помнить, что при этом меняется только имя root-пользователя - все права и привилегии его остаются неизменны.



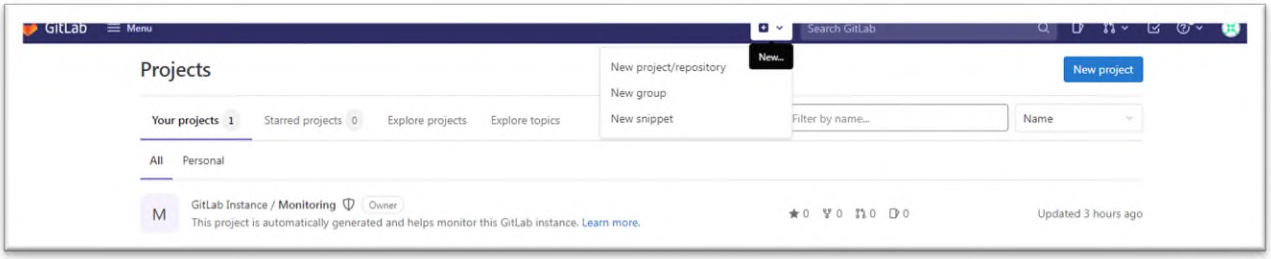
Основная настройка профиля (указание имени, почтового адреса для обращений и для коммитов, ссылки на социальные сети и пр.) производится в том же разделе меню «User Settings», но на вкладке «Profile».



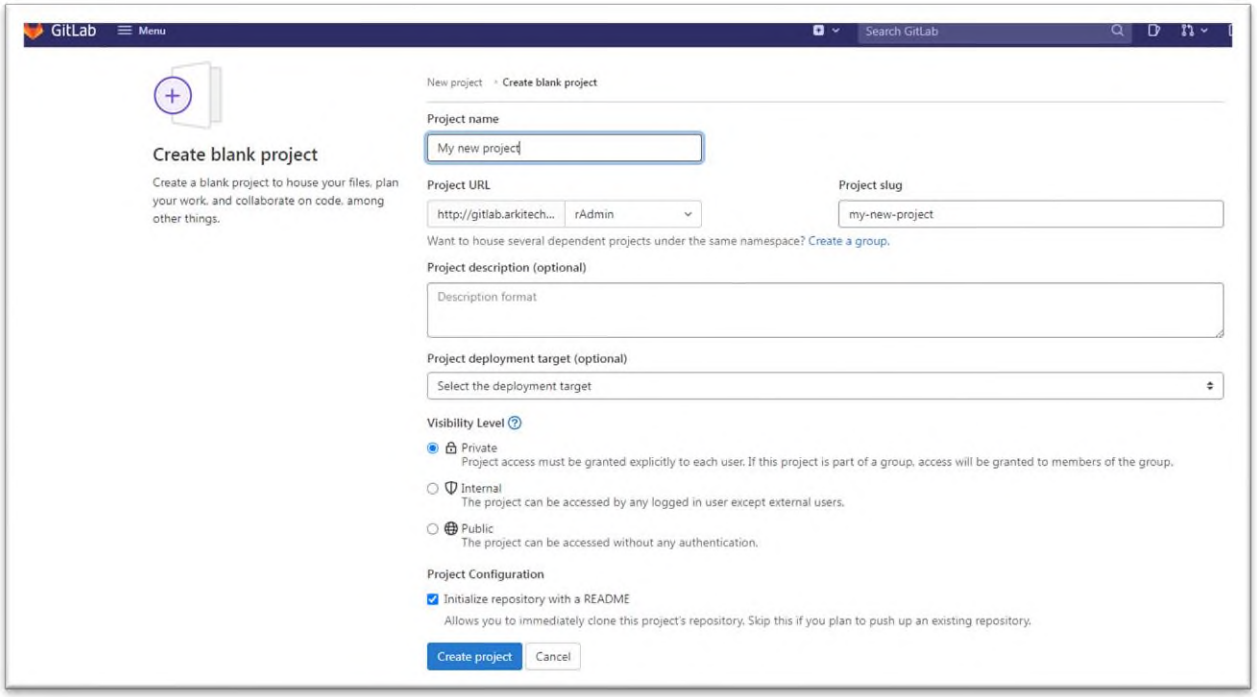
Создание и удаление проекта

Первым, что мы захотим сделать после настройки нашей учётной записи, будет, скорее всего, создание нового проекта.

Создать проект можно на главной странице Gitlab, нажав кнопку `New project` или на плюсик возле строки поиска



Здесь можно выбрать, создать новый проект, выбрать шаблон для проекта или загрузить уже имеющийся из репозитория. Создадим пустой проект.

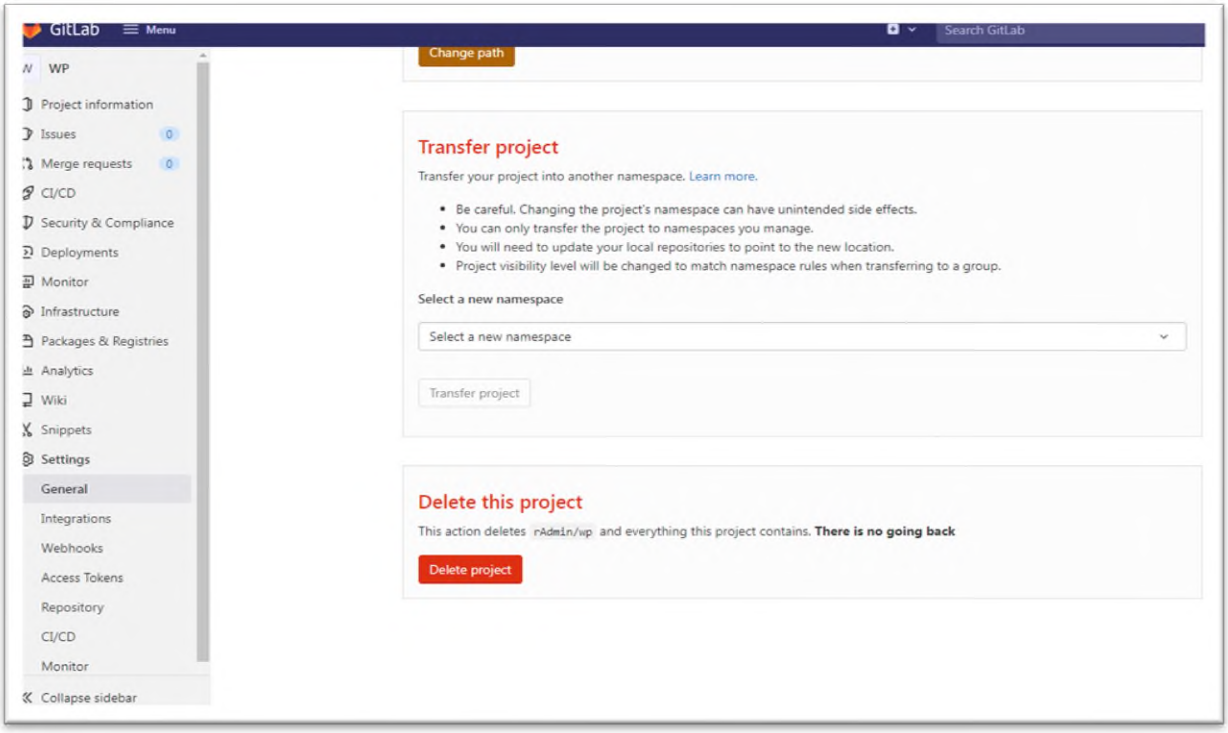


При создании проекта, помимо стандартных опций вроде задания имени, описания проекта и создания слага проекта, есть возможность выбрать уровень доступа к проекту:

- частный - доступ будет только у вас;
- внутренний - доступ только для тех, у кого есть доступ на сайт;
- публичный - открытый доступ для всех пользователей сети.

Также при создании проекта поставим галочку в пункте Initialize repository with a README — это позволит нам создать новый репозиторий, привязанный к нашему проекту. Первым файлом в репозитории станет README.txt, в котором обычно хранится полное описание проекта.

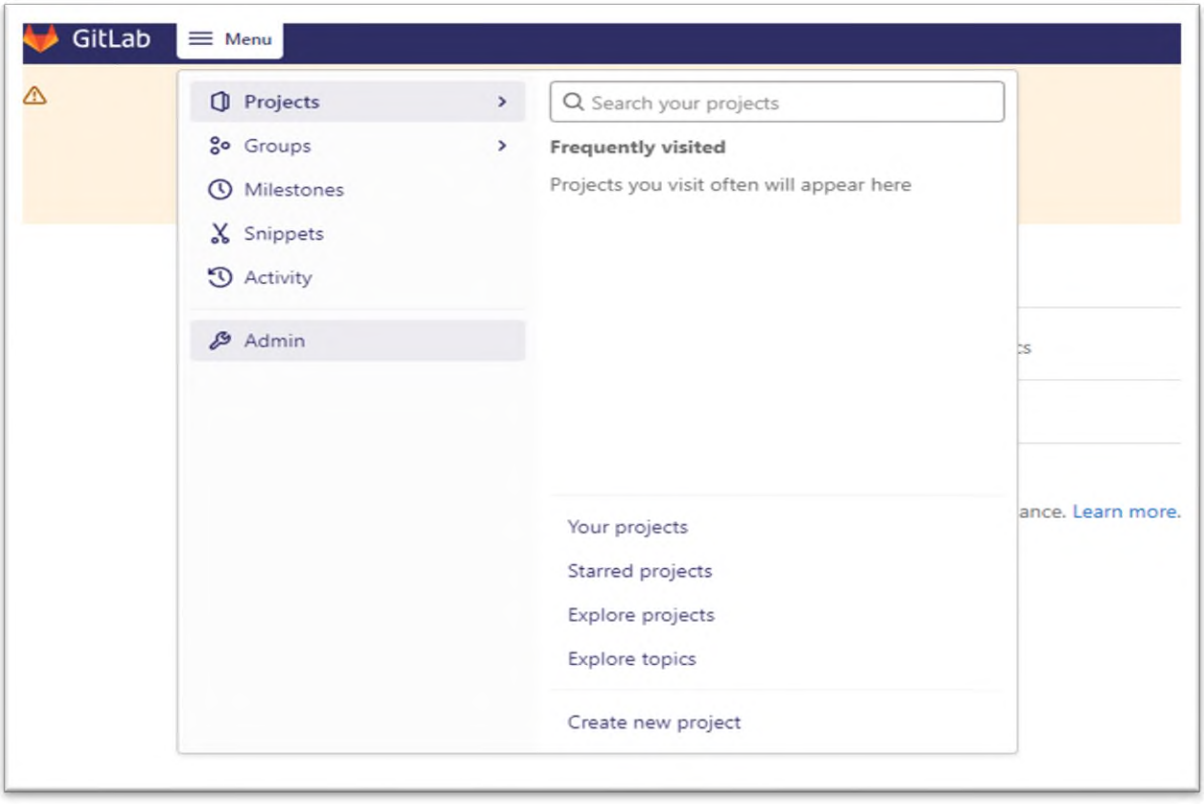
Удалить проект можно в разделе меню Settings — General. В нижней части раздела будет пункт Advanced при нажатии на кнопку Expand развернётся окно с расширенным набором настроек, в котором будет пункт Delete this project



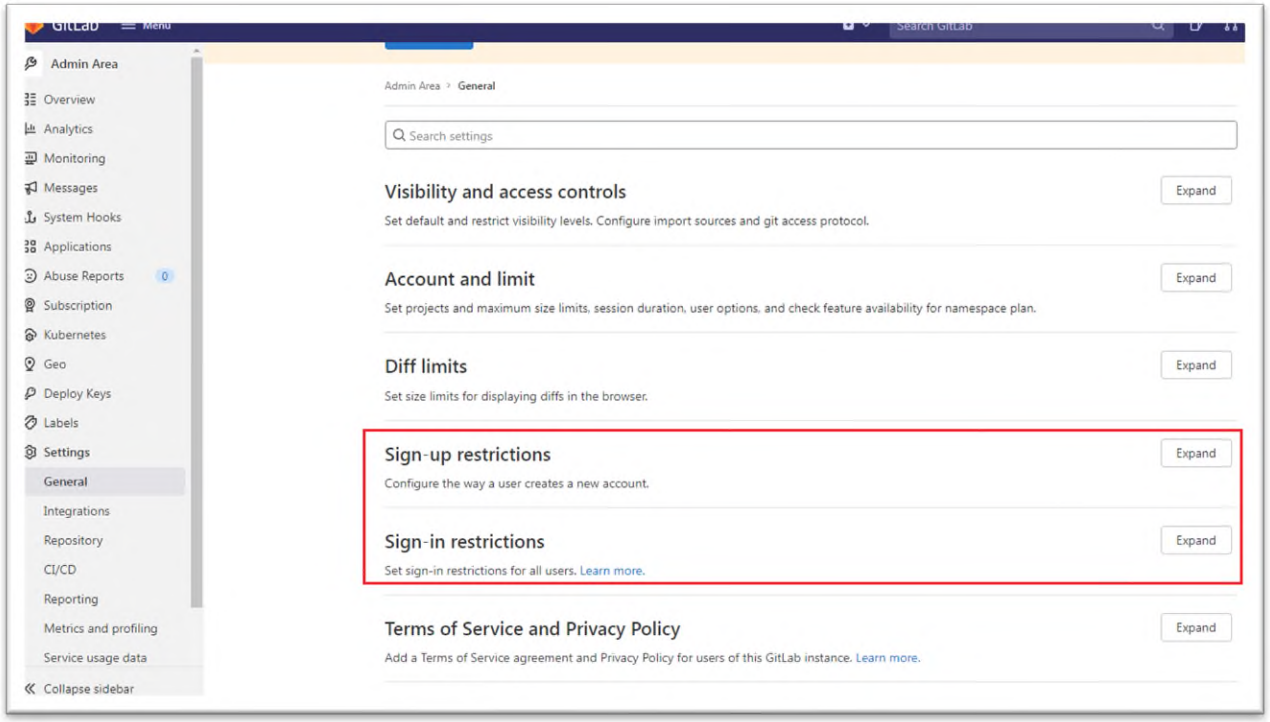
Настройки доступа

Административные права в Gitlab позволяют гибко настраивать политику доступа в отношении каждого пользователя и проекта.

Чтобы настроить ограничения доступа перейдите в административную область и в разделе Settings переключитесь на вкладку **General**.



Здесь в пунктах Sign-up restrictions и Sign-in restrictions можно настроить ограничения пользователей как на создание новых аккаунтов, так и на доступ к определённым проектам.



Эти разделы позволяют настроить ограничение доступа к проектам пользователям с определённых IP-адресов, задавать надёжность пароля для новых пользователей, а также создавать списки разрешённых и запрещённых доменных имён, чьи пользователи смогут или не смогут зарегистрироваться.

Добавить нового пользователя к уже существующему проекту можно из окна управления самим проектом.

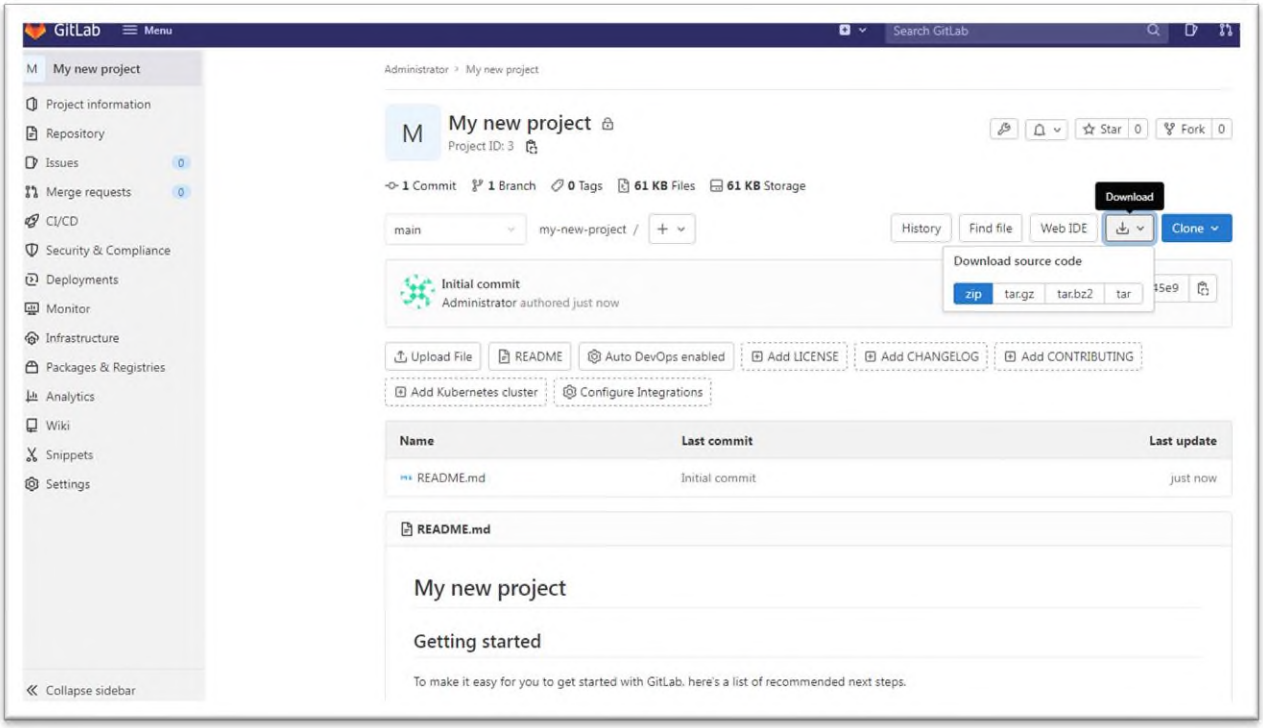
В разделе меню Project information выберите пункт Members и затем из уже созданных пользователей выберите тех, кому хотите открыть доступ к проекту.

Бэкап проекта

Создать резервную копию проекта можно двумя путями: через консоль SSH-подключения или через веб-интерфейс.

Веб-интерфейс

Резервную копию проекта можно в любой момент выгрузить с помощью веб-интерфейса Gitlab в одном из предложенных форматов: zip, tar.gz, tar.bz2, tar. Для этого нужно зайти в проект и нажать на кнопку Download на главной странице проекта



Полученную версию можно сохранить на локальной машине

Консоль

Чтобы создать бэкап своих проектов в консоли, сначала установите git:

```
# Centos 8
sudo dnf install git

#Ubuntu
sudo apt install git
```

После этого можно запустить менеджер команд Gitlab-rake:

```
sudo gitlab-rake gitlab:backup:create
```

После завершения работы команда выдаст предупреждение:

```
Warning: Your gitlab.rb and gitlab-secrets.json files contain sensitive data
and are not included in this backup. You will need these files to restore a backup.
Please back them up manually.
```

Указанные файлы gitlab.rb и gitlab-secrets.json нужно сохранить вручную.

Адрес, по которому будут располагаться резервные копии, указывается в конфигурационном файле:

```
sudo vi /etc/gitlab/gitlab.rb
```

В открывшемся файле находим блок

```
### Backup Settings
###! Docs: https://docs.gitlab.com/omnibus/settings/backups.html

# gitlab_rails['manage_backup_path'] = true
# gitlab_rails['backup_path'] = "/var/opt/gitlab/backups"
# gitlab_rails['backup_gitaly_backup_path'] = "/opt/gitlab/embedded/bin/gitaly-backup"
```

Как видим, по умолчанию резервные копии хранятся в /var/opt/gitlab/backups. Если нужно больше места для их хранения, выделите для этого отдельное дисковое пространство и перепишите в конфигурационном файле путь к ним.

Важно: после изменения конфигурационного файла не забудьте обновить конфигурацию Gitlab:

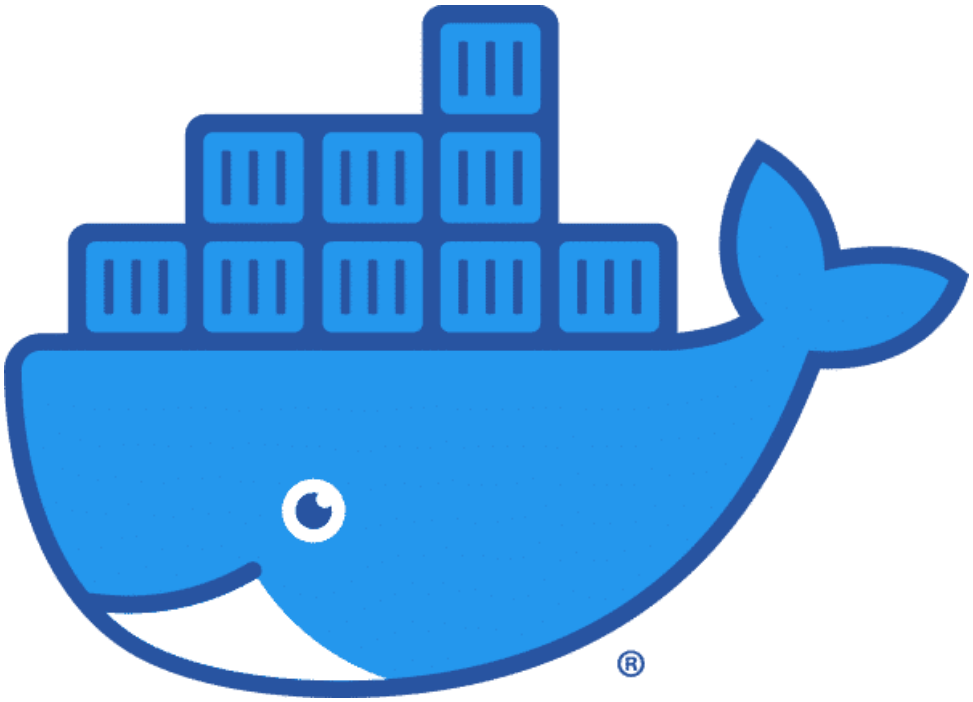
```
sudo gitlab-ctl reconfigure
```



МОДУЛЬ 15. Практические навыки работы с Docker

1. Знакомство с Docker
2. Создание собственного Image
3. Создание собственного контейнера
4. Знакомство с Registry
5. Размещение своего Image в публичный Registry

1. Знакомство с Docker

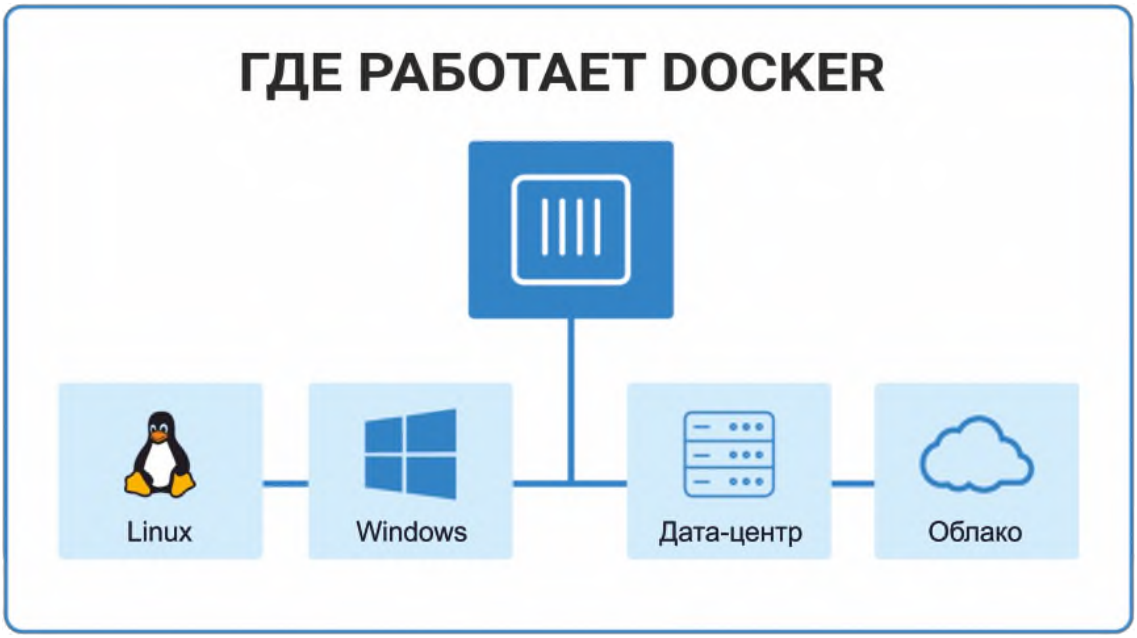


Docker ***(Докер)*** — программное обеспечение с открытым исходным кодом, применяемое для разработки, тестирования, доставки и запуска веб-приложений в средах с поддержкой контейнеризации. Он нужен для более эффективного использование системы и ресурсов, быстрого развертывания готовых программных продуктов, а также для их масштабирования и переноса в другие среды с гарантированным сохранением стабильной работы.

Разработка Docker была начата в 2008 году, а в 2013 году он был опубликован как свободно распространяемое ПО под лицензией Apache 2.0. В качестве тестового приложения Docker был включен в дистрибутив Red Hat Enterprise Linux 6.5. В 2017 году была выпущена коммерческая версия Docker с расширенными возможностями.

Docker работает в Linux, ядро которых поддерживает cgroups, а также изоляцию пространства имен. Для инсталляции и использования на платформах, отличных от Linux, существуют специальные утилиты Kitematic или Docker Machine.

Основной принцип работы Docker — контейнеризация приложений. Этот тип виртуализации позволяет упаковывать программное обеспечение по изолированным средам — контейнерам. Каждый из этих виртуальных блоков содержит все нужные элементы для работы приложения. Это дает возможность одновременного запуска большого количества контейнеров на одном хосте.



Docker-контейнеры работают в разных средах: локальном центре обработки информации, облаке, персональных компьютерах и т. д.

Преимущества использования Docker

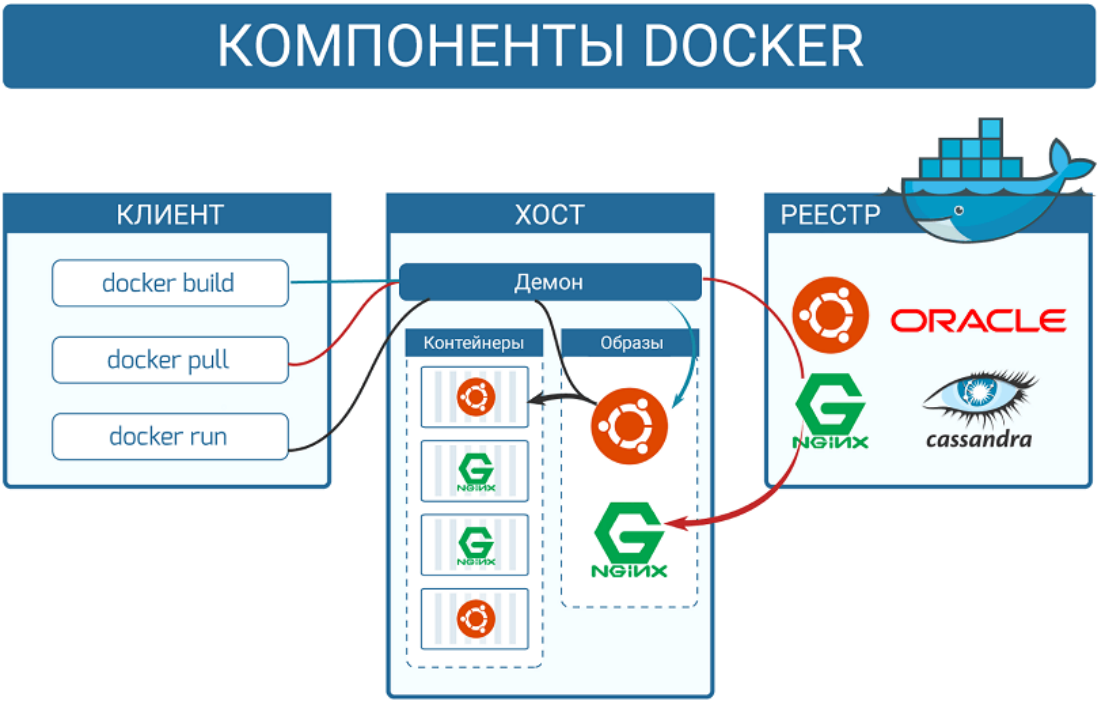
1. **Минимальное потребление** ресурсов — контейнеры не виртуализируют всю операционную систему (ОС), а используют ядро хоста и изолируют программу на уровне процесса. Последний потребляет намного меньше ресурсов локального компьютера, чем виртуальная машина.
2. **Скоростное развертывание** — вспомогательные компоненты можно не устанавливать, а использовать уже готовые docker-образы (шаблоны). Например, не имеет смысла постоянно устанавливать и настраивать Linux Ubuntu. Достаточно 1 раз ее инсталлировать, создать образ и постоянно использовать, лишь обновляя версию при необходимости.
3. **Удобное скрытие процессов** — для каждого контейнера можно использовать разные методы обработки данных, скрывая фоновые процессы.
4. Работа с **небезопасным кодом** — технология изоляции контейнеров позволяет запускать любой код без вреда для ОС.
5. **Простое масштабирование** — любой проект можно расширить, внедрив новые контейнеры.
6. **Удобный запуск** — приложение, находящееся внутри контейнера, можно запустить на любом docker-хосте.
7. **Оптимизация файловой системы** — образ состоит из слоев, которые позволяют очень эффективно использовать файловую систему.

Компоненты Docker

Для начинающих разработчиков необходимо знать как работает Docker, его основные компоненты и связь между ними.

Компоненты Docker

Для начинающих разработчиков необходимо знать, как работает Docker, его основные компоненты и связь между ними.



1. **Docker-демон** (Docker-daemon) — сервер контейнеров, входящий в состав программных средств Docker. Демон управляет Docker-объектами (сети, хранилища, образы и контейнеры). Демон также может связываться с другими демонами для управления сервисами Docker.
2. **Docker-клиент** (Docker-client / CLI) — интерфейс взаимодействия пользователя с Docker-демоном. Клиент и Демон — важнейшие компоненты «движка» Докера (Docker Engine). Клиент Docker может взаимодействовать с несколькими демонами.
3. **Docker-образ** (Docker-image) — файл, включающий зависимости, сведения, конфигурацию для дальнейшего развертывания и инициализации контейнера.
4. **Docker-файл** (Docker-file) — описание правил по сборке образа, в котором первая строка указывает на базовый образ. Последующие команды выполняют копирование файлов и установку программ для создания определенной среды для разработки.
5. **Docker-контейнер** (Docker-container) — это легкий, автономный исполняемый пакет программного обеспечения, который включает в себя все необходимое для запуска приложения: код, среду выполнения, системные инструменты, системные библиотеки и настройки.

6. **Том (Volume)** — эмуляция файловой системы для осуществления операций чтения и записи. Она создается автоматически с контейнером, поскольку некоторые приложения осуществляют сохранение данных.

7. **Реестр** (Docker-registry) — зарезервированный сервер, используемый для хранения docker-образов.

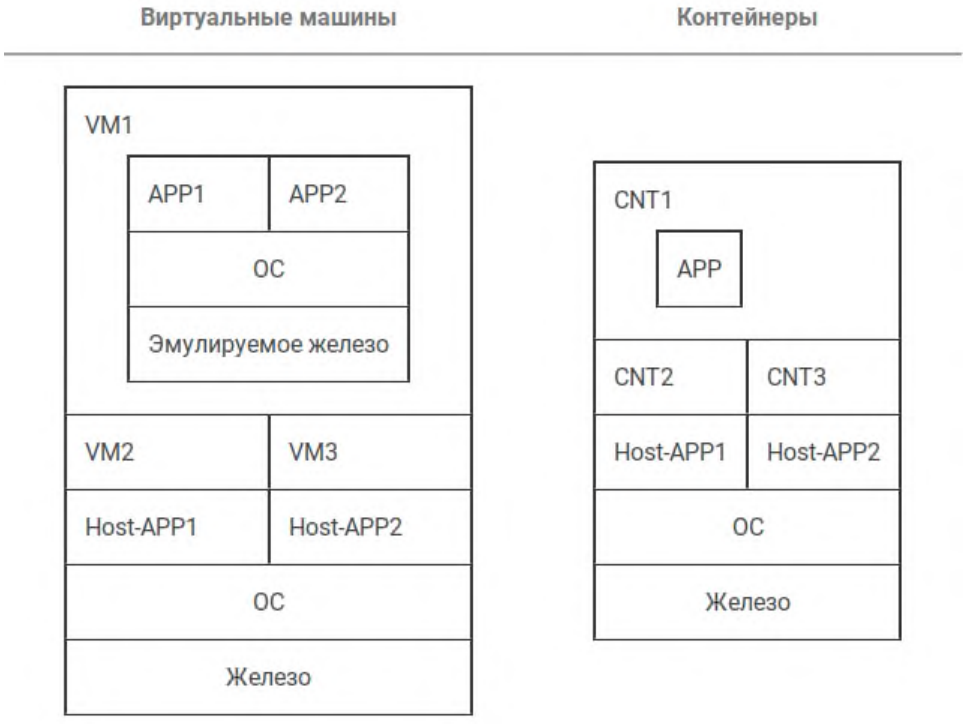
- Примеры реестров:
- Центр Docker — реестр, используемый для загрузки docker-image. Он обеспечивает их размещение и интеграцию с GitHub и Bitbucket.
- Контейнеры Azure — предназначен для работы с образами и их компонентами в директории Azure (Azure Active Directory).
- Доверенный реестр Docker или DTR — служба docker-реестра для инсталляции на локальном компьютере или сети компании.

8. **Docker-хаб** (Docker-hub) или хранилище данных — репозиторий, предназначенный для хранения образов с различным программным обеспечением. Наличие готовых элементов влияет на скорость разработки.

9. **Docker-хост** (Docker-host) — машинная среда для запуска контейнеров с программным обеспечением.

10. **Docker-сети** (Docker-networks) — применяются для организации сетевого интерфейса между приложениями, развернутыми в контейнерах.

Чем же отличаются контейнеры от виртуальных машин?



Как видно из данного изображения:

- Приложения в виртуальных машинах отделены от реального железа большим кол-вом слоёв логики, что влияет на расход ресурсов и может влиять на производительность.
- Код ОС дублируется в хостовой и виртуальной системах, из-за этого ВМ относительно долго загружаются, а также могут повторно применяться заплатки безопасности, которые в отдельных случаях понижают производительность.
- ВМ должны соревноваться за ресурсы с приложениями/ОС на хостовой системе (хотя этого и можно избежать настройкой планировщиков).
- ОС внутри ВМ тоже нужно администрировать!

Также стоит отметить, что в случае виртуальных машин ресурсы (например, диск и память) расходуются нерационально:

- Выделен диск размером 100Гб, но фактически используется ~40Гб.
- Выделено 16Гб памяти, но большую часть времени используется ~8Гб.

Обе эти проблемы тоже можно исправить, но возникает вопрос о необходимости и об удобстве использования такого решения. Несмотря на то, что виртуальные машины предоставляют большую гибкость и из-за этого в ряде случаев их выбор вполне обоснован, использовать их в повседневной работе не очень удобно.

Какие бывают контейнеры?

На иллюстрации выше изображена примерная модель для Docker контейнеров, следующая принципу один контейнер - одно приложение. Но это не является ограничением самих контейнеров, в них также можно запустить "init" процесс, который в свою очередь запустит множество приложений. Таким путём пошли LXC-контейнеры и в итоге они скорей являются легковесными аналогами виртуальных машин, со всеми недостатками такого решения:

- Их тоже нужно администрировать.
- Конфигурация задана жёстко и может варьироваться на разных инсталляциях.
- Сложно тиражировать/пересоздавать.

Docker же предлагает иной подход к контейнерам:

- Image контейнера предоставляет минимально необходимое окружение с приложением.

- Основные параметры (сетевые интерфейсы, DNS, IP-адреса и т. д.) конфигурируются извне, без необходимости производить настройки изнутри контейнера.
- Сборка контейнера воспроизводима.
- Изоляция средствами ОС - приложения в контейнере отделены от остальных (других контейнеров, хоста).
- Эфемерность - контейнеры временны, незначительные изменения не сохраняются.
- Значимые данные отделены от приложения и его окружения.

В итоге Docker предоставляет следующие возможности:

- Единообразие управления
- Автоматическая конфигурации сети и т. д.
- Создание окружения с помощью Images
- Вынесение изменяющихся данных во вне
- Изоляция отдельных приложений

Механизмы работы контейнеров

Linux-контейнеры (и docker в их числе) используют механизмы *cgroups* и *namespaces*. Первый позволяет ограничивать использование ресурсов, а второй - создавать отдельные пространства имён для различных аспектов процессов внутри ОС, среди которых есть:

- Пространство процессов (pid)
- Пространство сетевых интерфейсов (net)
- Пространство ФС (mnt)
- Пространство пользователей (user)

По-разному комбинируя эти пространства, можно получать интересные результаты, но одним из наиболее полезных вариантов является случай, когда все пространства - уникальны. В таком случае создаётся видимость обособленной системы, но при этом со стороны ОС сам процесс выполняется, как и все остальные.

Что такое платформа Docker?

Docker дает возможность упаковать и запустить приложение в слабо изолированном окружении, называемом контейнер. Изоляция и безопасность решения позволяют вам запускать множество контейнеров одновременно на нужном хосте. Благодаря легковесной природе контейнеров, без дополнительных затрат на гипервизор вы можете запускать больше контейнеров на доступном железе, чем если бы вы использовали виртуальные машины.

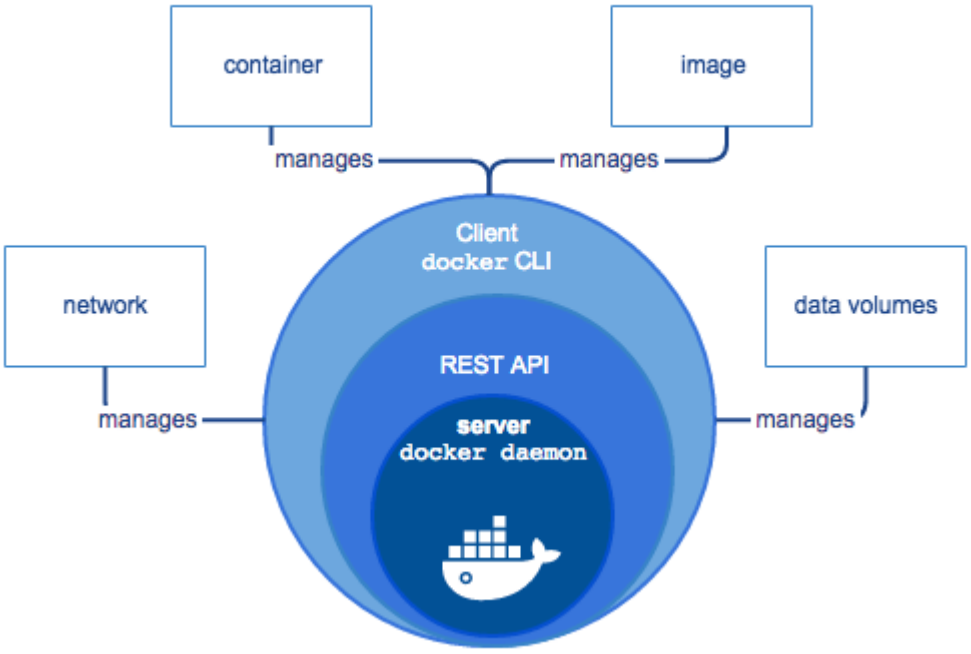
Docker предоставляет утилиты и платформу для управления жизненным циклом ваших контейнеров:

- Инкапсулируйте ваши приложения (и поддерживаемые компоненты) в контейнеры Docker
- Распространяйте и доставляйте эти контейнеры вашим командам для дальнейшей разработки и тестирования
- Разворачивайте эти приложения в вашем продуктивном окружении, что бы это ни было, локальный датацентр или облако

Что такое Docker Engine?

Docker Engine — это клиент-серверное приложение, содержащее следующие основные компоненты:

- Сервер, по сути являющийся долго работающим приложением, называемым демоном
- REST API, определяющее интерфейсы для взаимодействия, которые могут использовать другие программы.
- Консольный (CLI) клиент



Консольный клиент использует Docker REST API для управления или взаимодействия с демоном Docker при помощи скриптов или непосредственных консольных команд.

Демон создает и управляет объектами Docker, такими как образы (images), контейнеры, сети и тома данных (data volumes)

На заметку: Docker распространяется под открытой Apache 2.0 лицензией.

Элементы Docker Engine

Сервер выполняет инициализацию демона (фоновой программы), который применяется для управления и модификации контейнеров, образов и томов.

REST API — механизм, отвечающий за организацию взаимодействия Докер-клиента и Докер-демона.

Клиент — позволяет пользователю взаимодействовать с сервером при помощи команд, набираемых в интерфейсе (CLI).

Как работает Docker

Работа Docker основана на принципах клиент-серверной архитектуры, которая основана на взаимодействии клиента с веб-сервером (хостом). Первый отправляет запросы на получение данных, а второй их предоставляет.

Что такое Docker - архитектура

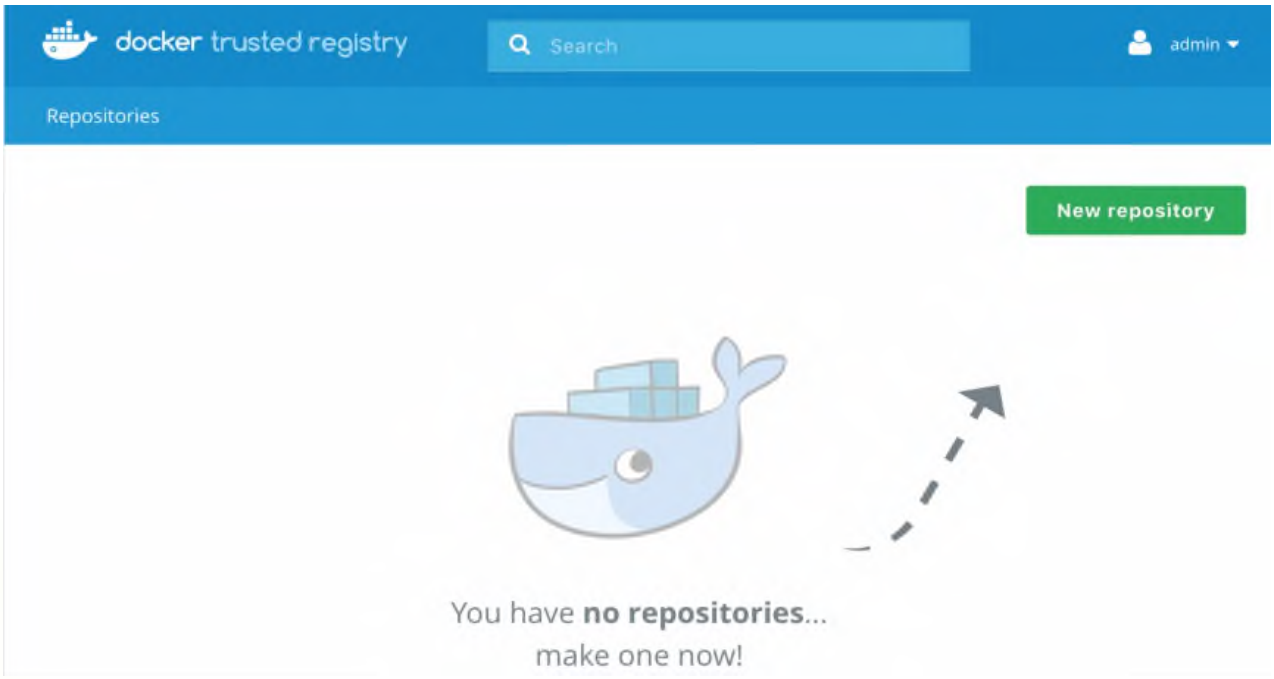
Схема работы

1. Пользователь отдает команду с помощью клиентского интерфейса Docker-демону, развернутому на Docker-хосте. Например, скачать готовый образ из реестра (хранилища Docker-образов) с помощью команды docker pull. Взаимодействие между клиентом и демоном обеспечивает REST API. Демон может использовать публичный (Docker Hub) или частный реестры.
2. Исходя из команды, заданной клиентом, демон выполняет различные операции с образами на основе инструкций, прописанных в файле Dockerfile. Например, производит их автоматическую сборку с помощью команды docker build.
3. Работа образа в контейнере. Например, запуск docker-image, посредством команды docker run или удаление контейнера через команду docker kill.

Как работают образы

Docker-image — шаблон только для чтения (read-only) с набором некоторых инструкций, предназначенных для создания контейнера. Он состоит из слоев, которые Docker комбинирует в один образ при помощи вспомогательной файловой системы UnionFS. Так решается проблема нерационального использования дисковой памяти. Параметры образа определяются в Docker-file.

Для многократного применения Docker-image следует пользоваться реестром образов или Докер-реестром (Docker-registry), позволяющим закачивать готовые образы с внешнего репозитория сервиса и хранить их в реестре Докер-хоста. Рекомендуемый вариант — официальный реестр компании Docker Trusted Registry (DTR).

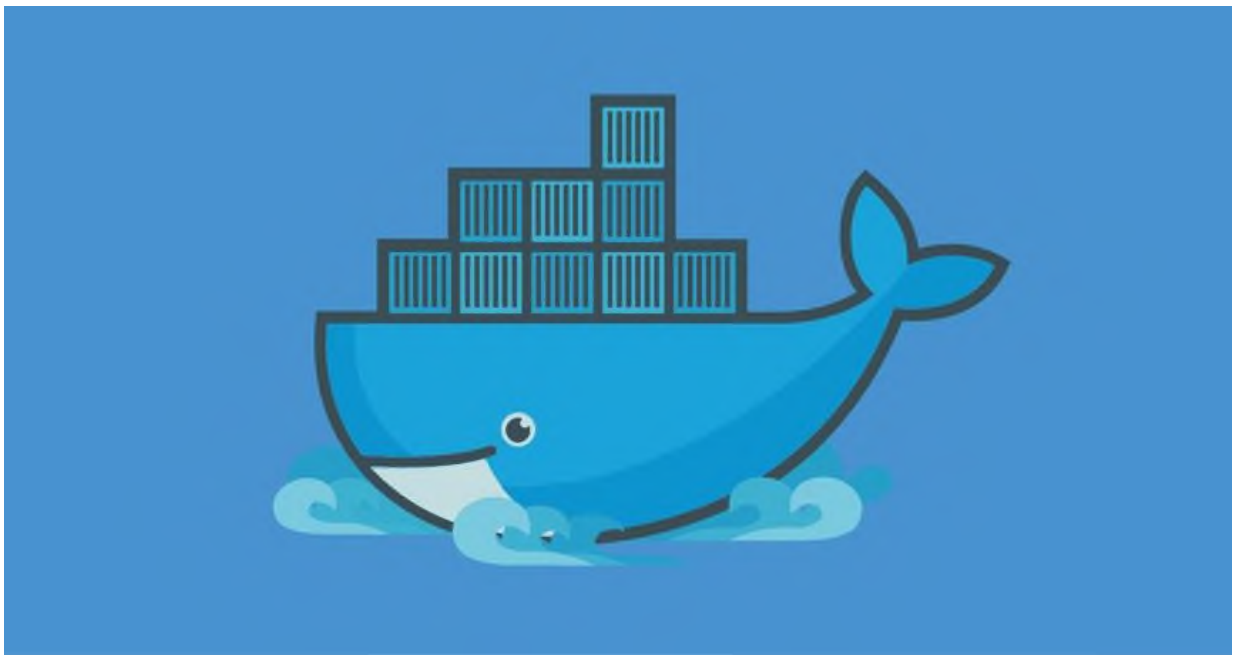


Если требуется файл, то скачиваться будут только нужные слои. Например, разработчик решил доработать программное обеспечение и модифицировать образ, изменив несколько файлов. После загрузки на сервер будут отправлены слои, содержащие только модифицированные данные.

Как работают контейнеры

Каждый контейнер строится на основе Docker-образов. Контейнеры запускаются напрямую из ядра операционной системы Linux. Благодаря этому они потребляют гораздо меньше ресурсов, чем при аппаратной виртуализации.

Изоляция рабочей среды осуществляется при помощи технологии namespace. Для каждого изолированного пространства (контейнера) создается уникальное пространство имен, которое и обеспечивает к нему доступ. Любой процесс, выполняемый внутри контейнера, ограничивается namespace.



В ОС Linux посредством Docker Engine используется немного другая технология — контрольные группы (cgroups). При этом приложение ограничивается некоторым набором ресурсов. Cgroups осуществляют обмен доступных аппаратных ресурсов с контейнерами, на которые дополнительно устанавливаются необходимые ограничения (использование памяти, прав доступа к другому ресурсу и т. д.).

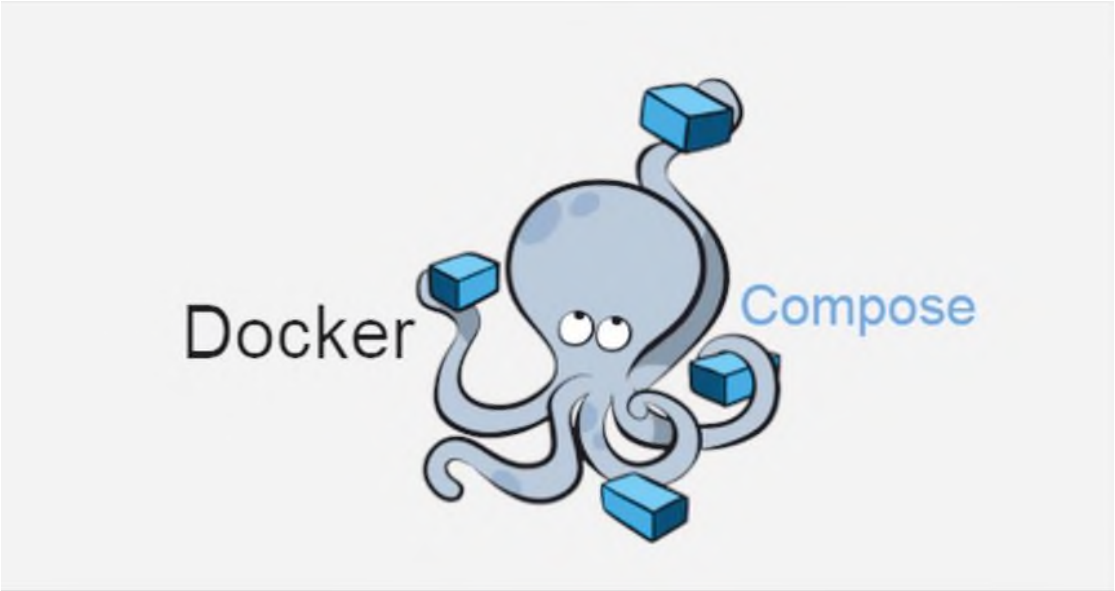
Движок Docker объединяет пространство имен (namespace), контрольные группы (cgroups) и файловую систему (UnionFS) в формат контейнера. В будущем планируется поддержка других форматов посредством интеграции технологий BSD Jails или Solaris Zones.

Что происходит при запуске контейнера

1. Происходит запуск образа (Docker-image). Docker Engine проверяет существование образа. Если образ уже существует локально, Docker использует его для нового контейнера. При его отсутствии выполняется скачивание с Docker Hub.
2. Создание контейнера из образа.
3. Разметка файловой системы и добавление слоя для записи.
4. Создание сетевого интерфейса.
5. Поиск и присвоение IP-адреса.
6. Запуск указанного процесса.
7. Захват ввода/вывода приложения.

Docker Compose

Для управления несколькими контейнерами, из которых состоит проект, используют пакетный менеджер — Docker Compose.

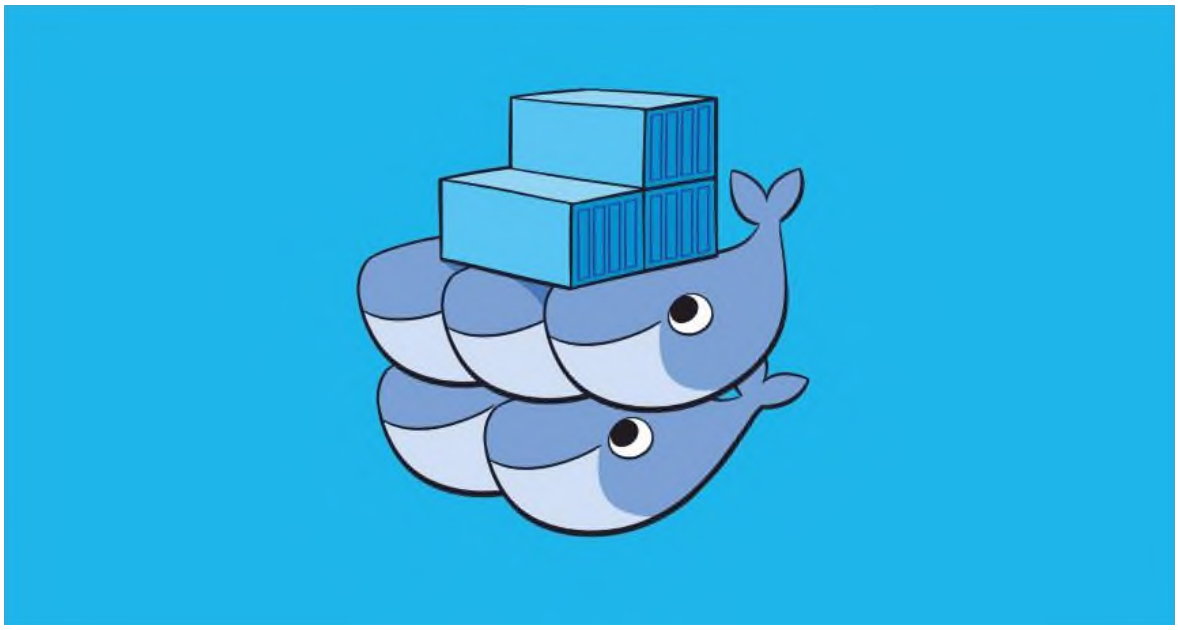


Он применяется не во всех случаях. Если проект является простым приложением, не требующим использования сторонних сервисов, то для его развертывания можно ограничиться только Docker. Docker Compose рекомендуется использовать при проектировании сложных программных продуктов, включающих в себя множество процессов и сервисов.

Подробнее о возможностях и практикической стороне работы с Docker Compose можно узнать здесь.

Docker Swarm

При преобразовании хостов в кластер нужно воспользоваться утилитой кластеризации Docker Swarm («Docker в режиме роя). Хост, находящийся в его составе, называется «узлом» (node), который бывает управляющим или рабочим. Один кластер содержит только один управляющий «узел».



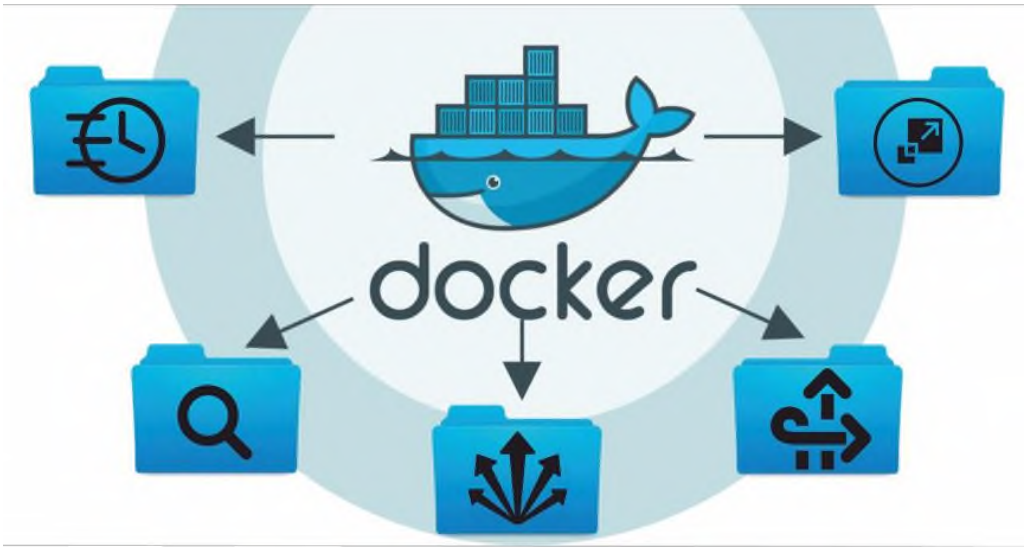
Некоторые возможности утилиты

1. Управление нагрузочными характеристиками — осуществляется оптимизация рассылки запросов между хостами, обеспечивая на них равномерную нагрузку.
2. Динамическое управление — допускается добавление элементов в swarm-кластер без дальнейшего его перезапуска.
3. Возможность масштабирования — позволяет добавлять или удалять docker-образ для автоматического создания контейнера.
4. Восстановление «узла» после сбоя — работоспособность каждого хоста постоянно контролируется управляющим «узлом». При сбое кластера происходит его восстановление и перезапуск.
5. Rolling-update — выполняет обновление контейнеров. Процедура может выполняться в определенной последовательности и с временной задержкой для запуска другого контейнера. Параметр указывается в настройках. Если произойдет сбой обновления, то Docker Swarm выдаст ошибку и процесс повторится заново.

Для оркестрации контейнеров Docker, наряду с Docker Swarm, применяют сторонний инструмент — опенсорсную платформу Kubernetes. Однако, в 2021 году в Kubernetes заявили об отказе от использования Doker в новых версиях. Поэтому связка Kubernetes и Docker будет постепенно терять актуальность.

Примеры применения

Окружение для разработки Docker применяется во множестве сфер — от обработки больших массивов данных, до работы с микросервисами, основанных на распределенной архитектуре.



Чтобы понять, как можно применять Докер на практике, разберем основные примеры использования для чайников.

1. Быстрая доставка приложений (команды docker pull и docker push) позволяет организовать коллективную работу над проектом. Разработчики могут работать удаленно на локальных компьютерах и выполнять пересылку фрагментов кода в контейнер для тестов.

2. Развертывание и масштабирование — контейнеры работоспособны на локальных компьютерах, серверах, в облачных онлайн-сервисах. Их можно загружать на хостинг для дальнейшего тестирования, создавать (docker run), останавливать (docker stop), запускать (docker start), приостанавливать и возобновлять (docker pause и docker unpause соответственно).

3. Множественные нагрузки — осуществление запуска большого количества контейнеров на одном и том же оборудовании, поскольку Docker занимает небольшой объем дисковой памяти.

4. Диспетчер процессов — возможность мониторинга процессов в Docker посредством команд docker ps и docker top, имеющими схожий синтаксис с Linux.

5. Удобный поиск — в реестрах Docker он осуществляется очень просто. Для этого следует использовать команду docker search.

Заключение

Docker является важным инструментом для каждого современного разработчика, как основа аппаратной виртуализации приложений. Эта технология обладает широким функционалом и возможностями для контроля процессов. Докер позволяет не только развертывать контейнеры, но и оперативно масштабировать их экземпляры, работать с многоконтейнерными приложениями (Docker Compose), а также объединять несколько Докер-хостов в единый кластер (Docker Swarm).

Докер характеризуется достаточно простым синтаксисом. Поэтому он довольно прост в освоении как для опытных IT-специалистов, так и для новичков. Программное обеспечение совместимо со всеми версиями операционных систем Linux и Windows, поэтому область применения Docker практически не ограничена.

2. Создание собственного Image

Что такое Docker-Image?

Docker Image — это файл, используемый для выполнения кода в контейнере Docker. Docker Images действуют как набор инструкций для создания контейнера Docker , например шаблона. Docker Images также служат отправной точкой при использовании Docker. Image сравним со снимком в среде виртуальной машины (ВМ).

Docker используется для создания, запуска и развертывания приложений в контейнерах. Docker Image содержит код приложения, библиотеки, инструменты, зависимости и другие файлы, необходимые для запуска приложения. Когда пользователь запускает Image, он может стать одним или несколькими экземплярами контейнера.

Docker Images имеют несколько слоев, каждый из которых происходит от предыдущего слоя, но отличается от него. Слои ускоряют сборку Docker , повышая возможность повторного использования и уменьшая использование диска. Слои изображений также являются файлами только для чтения. После создания контейнера поверх неизменяемых изображений добавляется доступный для записи слой, позволяющий пользователю вносить изменения.

Ссылки на дисковое пространство в Images и контейнерах Docker могут сбивать с толку. Важно различать размер и виртуальный размер. Размер относится к дисковому пространству, которое использует доступный для записи уровень контейнера, а виртуальный размер — это дисковое пространство, используемое для контейнера и доступного для записи уровня. Слои Image, доступные только для чтения, могут совместно использоваться любым контейнером, запущенным из одного и того же Image.

Варианты использования Docker Image

В Docker Image есть все необходимое для запуска контейнерного приложения, включая код, файлы конфигурации, переменные среды, библиотеки и среды выполнения. Когда Image развернут в среде Docker, его можно запустить как контейнер Docker. Команда `docker run` создает контейнер из определенного Image.

Docker Images — это ресурс многократного использования, который можно развернуть на любом хосте. Разработчики могут взять слои статического изображения из одного проекта и использовать их в другом. Это экономит время пользователя, поскольку ему не нужно воссоздавать изображение с нуля.

Контейнер Docker и Image Docker

Контейнер Docker — это виртуализированная среда выполнения, используемая при разработке приложений. Он используется для создания, запуска и развертывания приложений, изолированных от основного оборудования. Контейнер Docker может использовать одну машину, совместно использовать свое ядро и виртуализировать ОС для запуска более изолированных процессов. В результате контейнеры Docker легковесны.

Docker Image подобен моментальному снимку в других типах сред виртуальных машин. Это запись контейнера Docker в определенный момент времени. Docker Images также неизменны. Хотя их нельзя изменить, их можно дублировать, делиться ими или удалять. Эта функция полезна для тестирования нового программного обеспечения или конфигураций, потому что, что бы ни случилось, Image остается неизменным.

Для существования контейнеров требуется исполняемый Image. Контейнеры зависят от Images, поскольку они используются для создания среды выполнения и необходимы для запуска приложения.

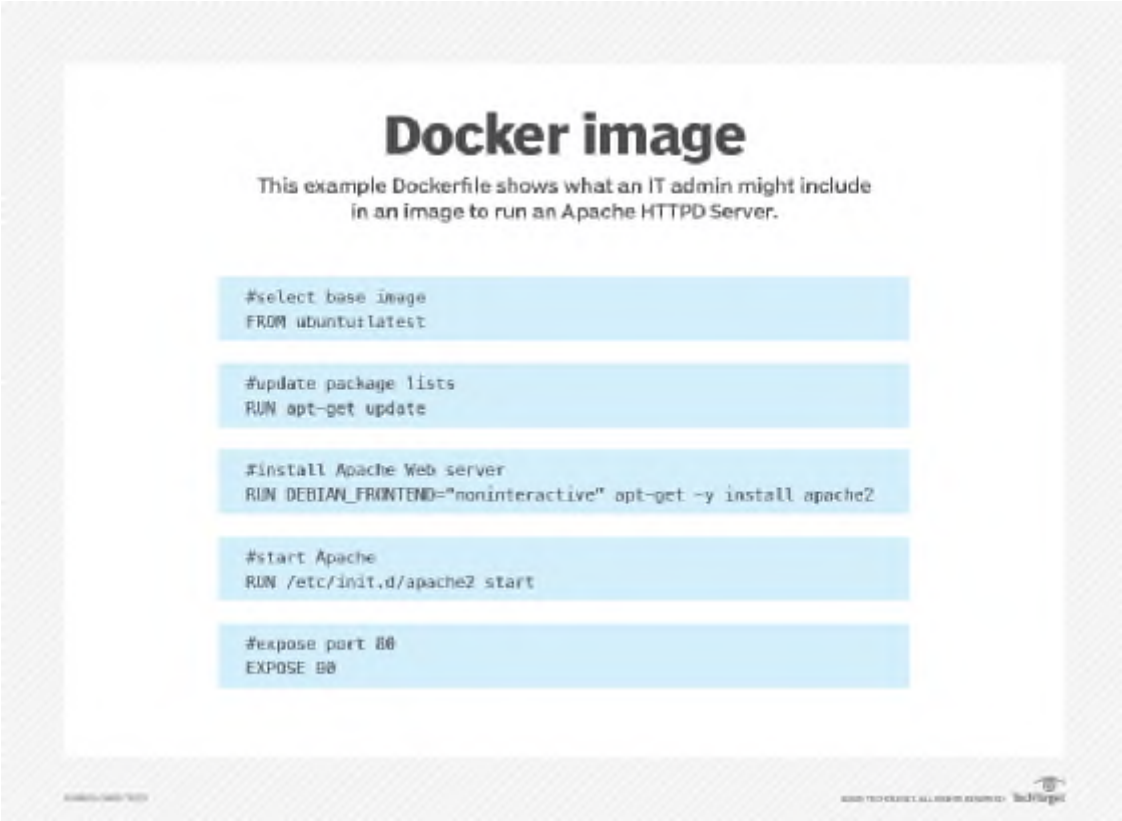
Анатомия Docker Image

Docker Image имеет много слоев, и каждый Image включает в себя все необходимое для настройки среды контейнера — системные библиотеки, инструменты, зависимости и другие файлы. Некоторые части изображения включают в себя:

- **Base image.** Пользователь может создать этот первый слой полностью с нуля с помощью команды сборки.
- **Parent image.** В качестве альтернативы базовому Image родительский образ может быть первым слоем в Docker Image. Это повторно используемое изображение, которое служит основой для всех остальных слоев.
- **Layers.** Слои добавляются к базовому Image с помощью кода, позволяющего запускать его в контейнере. Каждый уровень Docker Image можно просмотреть в `/var/lib/docker/aufs/diff` или с помощью команды `Docker history` в интерфейсе командной строки (CLI). По умолчанию Docker показывает все образы верхнего уровня, включая репозиторий , теги и размеры файлов. Промежуточные слои кэшируются , что облегчает просмотр верхних слоев. У Docker есть накопители, которые управляют содержимым слоя изображения.
- **Container layer.** Docker Image создает не только новый контейнер, но и доступный для записи слой или слой контейнера. На этом уровне размещаются изменения, внесенные в работающий контейнер, и хранятся

вновь записанные и удаленные файлы, а также изменения в существующих файлах. Этот слой также используется для настройки контейнеров.

- **Docker manifest.** Эта часть Docker Image является дополнительным файлом. Он использует формат JSON для описания изображения с использованием такой информации, как теги изображения и цифровая подпись.



Как может выглядеть Image Docker для запуска HTTPD-сервера Apache.

Репозитории Docker Images

Docker Images хранятся в частных или общедоступных репозиториях, например в облачной службе Registry Docker Hub , из которой пользователи могут развертывать контейнеры, а также тестировать образы и обмениваться ими. Docker Trusted Registry от Docker Hub также предоставляет возможности управления Images и контроля доступа.

Официальные образы создаются Docker, а *образы сообщества* — это образы, создаваемые пользователями Docker. Агент CoScale — это официальный Image Docker, который отслеживает приложения Docker . Datadog/docker-dd-agent, контейнер Docker для агентов в программе управления журналами Datadog , является примером Docker Image сообщества.

Пользователи также могут создавать новые образы из существующих и использовать команду docker push для загрузки пользовательских Images в Docker Hub. Чтобы обеспечить качество изображений сообщества, Docker предоставляет авторам обратную связь перед публикацией. После

публикации изображения автор несет ответственность за обновления. Авторы должны быть осторожны при получении изображения от другого лица, потому что злоумышленники могут получить доступ к системе через подражательные изображения, предназначенные для того, чтобы заставить пользователя думать, что они из надежного источника.

Концепция последнего изображения также может вызвать путаницу. Docker Images с тегом «:latest» не обязательно являются последними в обычном смысле. Последний тег не относится к самой последней версии изображения; это просто тег по умолчанию.

Как создать Image Docker

Docker Images можно создавать с помощью интерактивного метода или метода Dockerfile.

Интерактивный метод

С помощью этого метода пользователи запускают контейнер из существующего Docker Image и вручную вносят необходимые изменения в среду перед сохранением Image. Интерактивный метод — самый простой способ создания Images докеров. Первый шаг — запустить Docker и открыть сеанс терминала. Затем используйте команду Docker run image_name:tag_name. Это запускает сеанс оболочки с контейнером, который был запущен из Image. Если имя тега опущено, Docker использует самую последнюю версию Image. После этого изображение должно появиться в списке результатов.

Метод Dockerfile

Этот подход требует создания обычного текстового Dockerfile. Dockerfile содержит спецификации для создания Image. Этот процесс сложнее и требует больше времени, но он хорошо работает в средах с непрерывной доставкой (continuous delivery) . Метод включает в себя создание Dockerfile и добавление команд, необходимых для Image. После запуска Dockerfile пользователь настраивает файл .dockerignore, чтобы исключить любые файлы, которые не нужны для окончательной сборки. Файл .dockerignore находится в корневом каталоге. Затем команда Docker build используется для создания Docker Image, и задаются имя и тег Image. Наконец, команда Docker images используется для просмотра созданного Image.

Команды Docker Image

Согласно Docker, существуют наборы основных команд Docker Image, классифицируемых как дочерние команды; некоторые включают следующее:

- **docker image build.** Создает Image из Dockerfile.

- **docker image inspect.** Отображает информацию об одном или нескольких изображениях.
- **docker image load.** Загружает изображение из архива tar или потоков для приема или чтения ввода (STDIN).
- **docker image prune.** Удаляет неиспользуемые изображения.
- **docker image pull.** Извлекает Image или репозиторий из Registry.
- **docker image push.** Помещает Image или репозиторий в Registry.
- **docker image rm.** Удаляет одно или несколько изображений.
- **docker image save.** Сохраняет одно или несколько изображений в tar-архив (по умолчанию передается в STDOUT).
- **docker image tag.** Создает тег TARGET_IMAGE, ссылающийся на SOURCE_IMAGE.

Интерфейс командной строки Docker предоставляет команды, которые используются для настройки Docker Images. Примеры команд Docker Image включают следующее:

- **docker image history.** Показывает историю изображения, включая изменения, внесенные в него и его слои.
- **docker update.** Позволяет пользователю обновлять конфигурацию контейнеров.
- **docker tag.** Создает тег, например TARGET_IMAGE, который позволяет пользователям группировать и упорядочивать образы контейнеров.
- **docker search.** Ищет в Docker Hub все, что нужно пользователю.
- **docker save.** Позволяет пользователю сохранять изображения в архив.
- **docker compose.** Используется для обработки переменной среды.

Создайте базовое изображение

Большинство Dockerfiles начинаются с родительского образа. Если вам нужно полностью контролировать содержимое вашего образа, вам может потребоваться вместо этого создать базовый образ. Вот разница:

- Parent image — это изображение, на котором основано ваше изображение. Он относится к содержимому `FROM` директивы в Dockerfile. Каждое последующее объявление в Dockerfile изменяет этот родительский образ. Большинство файлов Dockerfile начинаются с родительского образа, а не с базового образа. Однако иногда эти термины используются взаимозаменяемо.
- Base image находится в файле Dockerfile `FROM scratch`.

В этом разделе показано несколько способов создания базового образа. Конкретный процесс будет сильно зависеть от дистрибутива Linux, который вы

хотите упаковать. У нас есть несколько примеров ниже, и вам рекомендуется отправлять запросы на вытягивание, чтобы добавлять новые.

Создайте полный образ с помощью tar

В общем, начните с работающей машины, на которой работает дистрибутив, который вы хотите упаковать в качестве родительского образа, хотя это не требуется для некоторых инструментов, таких как Debian Debootstrap , который вы также можете использовать для сборки образов Ubuntu.

Создать родительский образ Ubuntu может быть так же просто:

```
$ sudo debootstrap focal focal > /dev/null
$ sudo tar -C focal -c . | docker import - focal

sha256:81ec9a55a92a5618161f68ae691d092bf14d700129093158297b3d01593f4ee3

$ docker run focal cat /etc/lsb-release

DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=20.04
DISTRIB_CODENAME=focal
DISTRIB_DESCRIPTION="Ubuntu 20.04 LTS"
```

Еще примеры скриптов для создания родительских образов есть в репозитории Docker GitHub .

Создайте простой родительский образ с помощью нуля

Вы можете использовать зарезервированный минимальный образ Docker `scratch` в качестве отправной точки для создания контейнеров. Использование `scratch` «образа» сигнализирует процессу сборки, что вы хотите, чтобы следующая команда в Dockerfile файле была первым слоем файловой системы в вашем образе.

Пока `scratch` он появляется в репозитории Docker на концентраторе, вы не можете извлечь его, запустить или пометить любой образ именем `scratch`. Вместо этого вы можете ссылаться на него в своем Dockerfile. Например, чтобы создать минимальный контейнер, используя `scratch`:

```
# syntax=docker/dockerfile:1
FROM scratch
ADD hello /
CMD ["/hello"]
```

Предполагая, что вы создали исполняемый пример «hello», используя исходный код на https://github.com/docker-library/hello-world , и вы скомпилировали его с –static флагом, вы можете создать этот образ Docker с помощью этой docker build команды:

```
$ docker build --tag hello .
```

Не забудьте . символ в конце, который устанавливает контекст сборки в текущий каталог.

Примечание . Поскольку Docker Desktop для Mac и Docker Desktop для Windows используют виртуальную машину Linux, вам нужен двоичный файл Linux, а не двоичный файл Mac или Windows. Вы можете использовать контейнер Docker для его сборки:

```
$ docker run --rm -it -v $PWD:/build ubuntu:20.04

container# apt-get update && apt-get install build-essential
container# cd /build
container# gcc -o hello -static hello.c
```

Чтобы запустить новый образ, используйте docker run команду:

```
$ docker run --rm hello
```

3. Создание собственного контейнера

Подсистема Docker содержит средства, автоматизирующие создание образов контейнеров. Хотя образы контейнеров можно создавать вручную с помощью команды `docker commit`, внедрение процесса автоматического создания образа предоставляет множество преимуществ, в том числе:

- Сохранение образов контейнеров в виде кода.
- Быстрое и точное воссоздание образов контейнеров для обслуживания и обновления.
- Непрерывная интеграция между образами контейнеров и циклом разработки.

За такую автоматизацию отвечают два компонента Docker — файл `Dockerfile` и команда `docker build`.

`Dockerfile` — это текстовый файл с инструкциями, необходимыми для создания образа контейнера. Эти инструкции включают идентификацию существующего образа, используемого в качестве основы, команды, выполняемые в процессе создания образа, и команду, которая будет выполняться при развертывании новых экземпляров этого образа контейнера.

`Docker build` — команда подсистемы Docker, использующая файл Dockerfile и запускающая процесс создания образа.

В этом разделе рассказывается о том, как использовать файлы Dockerfile с контейнерами Windows, а также объясняются наиболее распространенные инструкции и базовый синтаксис таких файлов.

Базовый синтаксис

В исходной форме файл Dockerfile может быть очень простым. Следующий пример создает образ, включающий IIS и сайт "hello world". Этот пример включает комментарии (обозначенные с помощью `#`), поясняющие каждый шаг. В последующих разделах этой главы мы более подробно рассматрим правила синтаксиса Dockerfile и инструкции Dockerfile.

Примечание

Файл Dockerfile необходимо создавать без расширения. Чтобы сделать это в Windows, создайте файл с помощью удобного для вас редактора, а затем сохраните его, используя нотацию "Dockerfile" (вместе с кавычками).

```
# Sample Dockerfile

# Indicates that the windows server core image will be used as the base image.
FROM mcr.microsoft.com/windows/servercore:ltsc2019
```

```
# Metadata indicating an image maintainer.
LABEL maintainer="jshelton@contoso.com"

# Uses dism.exe to install the IIS role.
RUN dism.exe /online /enable-feature /all /feature name:iis-webserver /NoRestart

# Creates an HTML file and adds content to this file.
RUN echo "Hello World - Dockerfile" > c:\inetpub\wwwroot\index.html

# Sets a command or process that will run each time a container is run from the new image.
CMD [ "cmd" ]
```

Инструкции

Инструкции Dockerfile дают подсистеме Docker необходимые указания для создания образа контейнера. Эти инструкции выполняются по очереди, одна за другой. Ниже приведены примеры наиболее часто используемых инструкций в файлах Dockerfile. Полный список инструкций Dockerfile см. в справочнике по файлам Dockerfile.

FROM

Инструкция FROM задает образ контейнера, который будет применяться при создании нового образа. Например, при использовании инструкции FROM mcr.microsoft.com/windows/servercore полученный образ является производным и зависимым от базового образа ОС Windows Server Core. Если указанный образ отсутствует в системе, где выполняется процесс сборки Docker, подсистема Docker попытается скачать его из общедоступного или частного реестра образов.

Формат инструкции FROM выглядит следующим образом:

```
FROM <image>
```

Ниже приведен пример команды FROM.

Чтобы загрузить Windows Server Core версии ltsc2019 из Реестра контейнеров (Майкрософт):

```
Копировать
FROM mcr.microsoft.com/windows/servercore:ltsc2019
```

Дополнительные сведения см. в справочнике по инструкции FROM.

RUN

Инструкция RUN задает команды, которые следует выполнить и поместить в новый образ контейнера. Эти команды могут включать такие элементы, как установка программного обеспечения, создание файлов и папок, а также создание конфигурации среды.

Инструкция RUN выглядит следующим образом:

```
# exec form

RUN ["<executable>", "<param 1>", "<param 2>"]

# shell form

RUN <command>
```

Различие между формой исполняемого файла (exec form) и формой оболочки (shell form) заключается в способе выполнения инструкции RUN. При использовании формы исполняемого файла указанная программа запускается явным образом.

Ниже приведен пример формы исполняемого файла.

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019

RUN ["powershell", "New-Item", "c:/test"]
```

Полученный образ выполняет команду powershell New-Item c:/test:

```
docker history doc-exe-method

IMAGE          CREATED         CREATED BY                      SIZE
COMMENT
b3452b13e472   2 minutes ago   powershell New-Item c:/test     30.76
MB
```

В отличие от предыдущего примера здесь та же операция выполняется с использованием формы оболочки:

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019

RUN powershell New-Item c:\test
```

Полученный образ содержит инструкцию RUN cmd /S /C powershell New-Item c:\test.

```
docker history doc-shell-method
```

```
IMAGE          CREATED          CREATED BY                                      SIZE
COMMENT
062a543374fc   19 seconds ago   cmd /S /C powershell New-Item c:\test           30.76 MB
```

Рекомендации по использованию инструкции RUN с Windows

В Windows при использовании инструкции RUN в формате исполняемого файла необходимо экранировать инструкции символы обратной косой черты.

```
RUN ["powershell", "New-Item", "c:\\test"]
```

Если целевая программа является установщиком Windows, чтобы запустить фактическую процедуру (автоматической) установки, необходимо извлечь программу установки, используя флаг /x:<directory>. Прежде чем выполнять какие-либо другие действия, необходимо дождаться завершения выполнения команды. В противном случае процесс будет завершен преждевременно и без установки. Дополнительные сведения см. в приведенном ниже примере.

Примеры использования инструкции RUN с Windows

В следующем примере для установки служб IIS в образе контейнера используется система DISM.

```
RUN dism.exe /online /enable-feature /all /featurename:iis-webserver /NoRestart
```

Этот пример устанавливает распространяемый пакет Visual Studio. Start-Process и параметр -Wait используются для запуска программы установки. Это гарантирует, что установка будет завершена до перехода к следующей инструкции в Dockerfile.

```
RUN powershell.exe -Command Start-Process c:\vcredist_x86.exe -ArgumentList '/quiet' -Wait
```

Подробные сведения об инструкции RUN см. в справочнике по RUN на странице .

COPY

Инструкция COPY копирует файлы и каталоги в файловую систему контейнера. Эти файлы и каталоги должны иметь путь, являющийся относительным для Dockerfile.

Формат инструкции COPY выглядит следующим образом:

```
COPY <source> <destination>
```

Если источник или назначение содержит пробел, заключите путь в квадратные скобки и двойные кавычки, как показано в следующем примере:

```
COPY ["<source>", "<destination>"]
```

Рекомендации по использованию инструкции COPY с Windows

В Windows для путей назначения необходимо использовать символы косой черты. Например, здесь показаны допустимые инструкции `COPY`.

```
COPY test1.txt /temp/
COPY test1.txt c:/temp/
```

При этом следующий формат с обратными косыми чертами работать не будет:

```
COPY test1.txt c:\temp\
```

Примеры использования инструкции COPY с Windows

В следующем примере содержимое исходного каталога добавляется в каталог с именем `sqllite` в образе контейнера.

```
COPY source /sqlite/
```

В следующем примере все файлы, начинающиеся с config, добавляют в каталог `c:\temp` образа контейнера.

```
COPY config* c:/temp/
```

Дополнительные сведения об инструкции COPY см. в справочнике по инструкции COPY.

ADD

Инструкция ADD похожа на инструкцию COPY, но с дополнительными возможностями. Кроме копирования файлов с узла в образ контейнера, инструкция `ADD` также позволяет скопировать файлы из удаленного расположения с помощью задания URL-адреса.

Формат инструкции ADD выглядит следующим образом:

```
ADD <source> <destination>
```

Если источник или назначение содержит пробел, заключите путь в квадратные скобки и двойные кавычки.

```
ADD ["<source>", "<destination>"]
```

Рекомендации по выполнению инструкции ADD с Windows

В Windows для путей назначения необходимо использовать символы косой черты. Например, здесь показаны допустимые инструкции `ADD`.

```
ADD test1.txt /temp/
ADD test1.txt c:/temp/
```

При этом следующий формат с обратными косыми чертами работать не будет:

```
ADD test1.txt c:\temp\
```

Кроме того, в системе Linux во время копирования инструкция ADD распаковывает сжатые пакеты. В Windows эта функция недоступна.

Примеры использования инструкции ADD с Windows

В следующем примере содержимое исходного каталога добавляется в каталог с именем sqllite в образе контейнера.

```
ADD source /sqlite/
```

В следующем примере все файлы, начинающиеся с "config", добавляют в каталог c:\temp образа контейнера.

```
ADD config* c:/temp/
```

В этом примере Python для Windows скачивается в каталог c:\temp образа контейнера.

```
ADD https://www.python.org/ftp/python/3.5.1/python-3.5.1.exe /temp/python-3.5.1.exe
```

Дополнительные сведения об инструкции ADD см. в справочнике по инструкции ADD.

WORKDIR

Инструкция WORKDIR задает рабочий каталог для других инструкций Dockerfile, например, RUN и CMD, а также рабочий каталог для запущенных экземпляров образа контейнера.

Формат инструкции WORKDIR выглядит следующим образом:

```
WORKDIR <path to working directory>
```

Рекомендации по использованию инструкции WORKDIR с Windows

Если в Windows рабочий каталог содержит обратную косую черту, ее следует экранировать.

```
WORKDIR c:\\windows
```

Примеры

```
WORKDIR c:\\Apache24\\bin
```

Подробные сведения об инструкции WORKDIR см. в справочнике по WORKDIR.

CMD

Инструкция CMD задает команду по умолчанию, выполняемую при развертывании экземпляра образа контейнера. Например, если в контейнере будет размещен веб-сервер NGINX, CMD может включать инструкции для запуска этого веб-сервера, например, с помощью команды nginx.exe. Если в файле Dockerfile указано несколько инструкций CMD, вычисляется только последняя из них.

Формат инструкции CMD выглядит следующим образом:

```
# exec form

CMD ["<executable", "<param>"]

# shell form

CMD <command>
```

Рекомендации по использованию инструкции CMD с Windows

В Windows для путей к файлам, указанным в инструкции CMD, следует использовать символы косой черты или экранировать символы обратной косой черты \\. Допустимы следующие инструкции CMD:

```
# exec form

CMD ["c:\\Apache24\\bin\\httpd.exe", "-w"]

# shell form

CMD c:\\Apache24\\bin\\httpd.exe -w
```

Однако следующий формат без соответствующих косых черт работать не будет:

```
CMD c:\Apache24\bin\httpd.exe -w
```

Escape-символ

Во многих случаях инструкция Dockerfile должна занимать несколько строк. Для этого можно использовать escape-символ. Escape-символ Dockerfile по умолчанию — обратная косая черта (\). Однако, поскольку обратная косая черта также является разделителем пути к файлу в Windows, использование его для разделения нескольких строк может вызвать проблемы. Чтобы их избежать, можно изменить escape-символ по умолчанию с помощью директивы анализатора.

В следующем примере показана инструкция RUN, которая занимает несколько строк и использует escape-символ по умолчанию.

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019

RUN powershell.exe -Command \
  $ErrorActionPreference = 'Stop'; \
```

```
        wget https://www.python.org/ftp/python/3.5.1/python-3.5.1.exe -OutFile
c:\python-3.5.1.exe ; \
        Start-Process c:\python-3.5.1.exe -ArgumentList '/quiet InstallAllUsers=1
PrependPath=1' -Wait ; \
        Remove-Item c:\python-3.5.1.exe -Force
```

Чтобы изменить escape-символ, поместите директиву Parser для escape-символа на первую строку Dockerfile. Это показано в следующем примере.

Примечание

В качестве escape-символов можно использовать только символы: \ и `.

```
# escape=`

FROM mcr.microsoft.com/windows/servercore:ltsc2019

RUN powershell.exe -Command `
    $ErrorActionPreference = 'Stop'; `
    wget https://www.python.org/ftp/python/3.5.1/python-3.5.1.exe -OutFile
c:\python-3.5.1.exe ; `
    Start-Process c:\python-3.5.1.exe -ArgumentList '/quiet InstallAllUsers=1
PrependPath=1' -Wait ; `
    Remove-Item c:\python-3.5.1.exe -Force
```

4. Знакомство с Registry

Что это

Registry — это высоко масштабируемое серверное приложение без сохранения состояния, которое хранит и позволяет вам распространять образы Docker. Registry с открытым исходным кодом под разрешающей лицензией Apache .

Зачем это использовать

Вам следует использовать Registry, если вы хотите:

- жестко контролировать, где хранятся ваши изображения
- полностью владеть конвейером распространения изображений
- тесно интегрируйте хранение и распространение изображений в ваш внутренний рабочий процесс разработки

Альтернативы

Пользователям, которые ищут готовое решение, не требующее обслуживания, рекомендуется перейти на Docker Hub , который предоставляет бесплатный для использования размещенный Registry, а также дополнительные функции (учетные записи организации, автоматические сборки и т. д.).

Основные команды

Запустите свой Registry

```
docker run -d -p 5000:5000 --name registry registry:2
```

Вытащите (или создайте) какой-нибудь образ из хаба

```
docker pull ubuntu
```

Пометьте образ так, чтобы он указывал на ваш Registry

```
docker image tag ubuntu localhost:5000/myfirstimage
```

Отправить это в Registry

```
docker push localhost:5000/myfirstimage
```

Вытянуть его обратно

```
docker pull localhost:5000/myfirstimage
```

Теперь остановите свой Registry и удалите все данные

```
docker container stop registry && docker container rm -v registry
```

5. Размещение своего Image в публичный Registry

Docker Hub публичный реестр поддерживаемый Docker, Inc. Он содержит образы, которые вы можете скачать и использовать для создания контейнеров. А также обеспечивает проверку подлинности, структуру рабочей группы, инструменты рабочего процесса, такие как webhooks, триггеры и инструменты приватности, как частных хранилищ для хранения образов, которые вы не хотите делать публичными. Docker команды и Docker Hub. Docker сам обеспечивает доступ к сервисам Docker Hub через команды docker search, pull, login и push. В этой главе мы покажем, как эти команды работают. Создание аккаунта и авторизация.

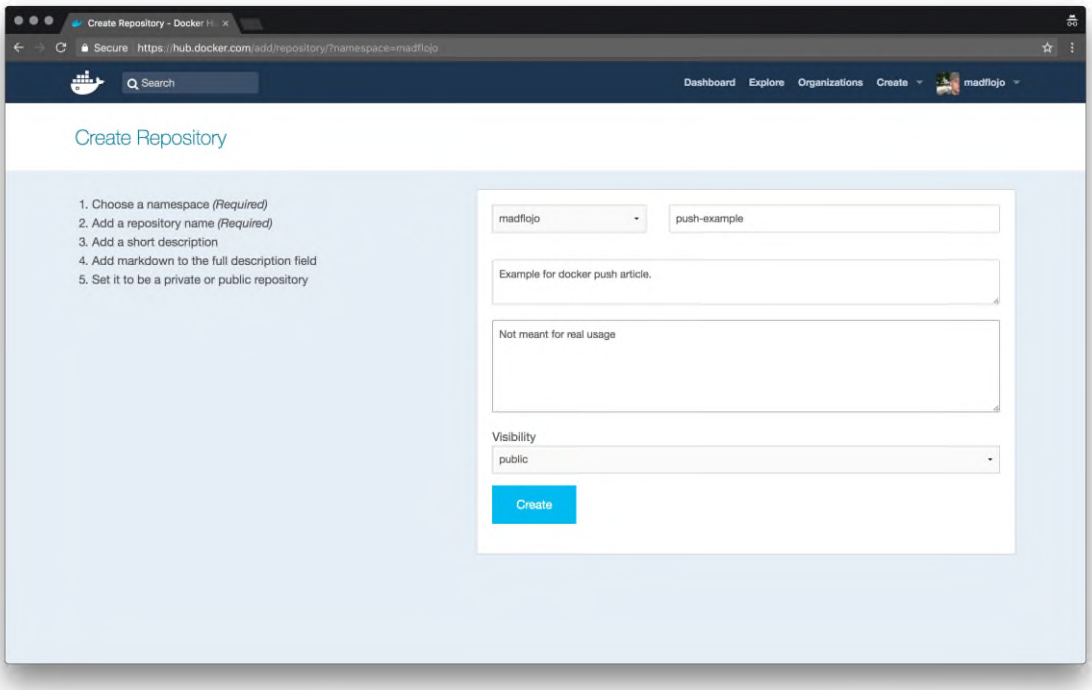
Возможность совместного использования изображений контейнеров в Docker Hub (публичный / частный реестр Docker) позволяет пользователям быстро делиться и опираться на уже существующие изображения.

В данной главе мы собираемся использовать эту функцию при изучении docker pushкоманды и использовать ее для загрузки нашего изображения контейнера Docker в Docker Hub.

Создание репозитория в Docker Hub

Прежде чем мы сможем отправить изображение в Docker Hub, нам сначала понадобится учетная запись в Docker Hub. Процесс регистрации прост и доступен на главной странице Docker Hub.

Как только у нас будет учетная запись, нашим следующим шагом будет создание нового репозитория. Этот процесс довольно понятен, однако на следующем скриншоте показан тип информации, которую мы должны предоставить.



После создания репозитория мы можем начать создавать образ Docker и загрузить его в Docker Hub.

Создание нового образа

Мы создадим образ с именем push-example. Как правило, если бы мы создавали этот образ для локального развертывания, мы бы просто выполнили следующую docker buildкоманду.

```
$ docker build -t push-пример.
```

Приведенная выше команда создаст новое изображение с тегом as push-example. В локальных развертываниях мы просто выдадим a docker run, указав этот образ по имени тега.

Хотя это работает для локального развертывания, это соглашение об именах не работает при создании образа для Docker Hub.

Использование имени Docker Hub при создании образа

При отправке изображения в Docker Hub мы должны указать имя пользователя Docker Hub как часть имени изображения. Для примера выше мне нужно будет создать изображение с именем тега madflojo/push-example.

Madflojo Часть имени – это имя пользователя для Docker Hub. Для того, чтобы мое изображение было загружено в репозиторий push-example под моим именем пользователем, мы должны включить это имя пользователя в имя изображения с форматом (username/image_name) .

Причина этого в том, что Docker Hub организует репозитории по имени пользователя. Любой репозиторий, созданный под учетной записью, включает имя пользователя в имя образа Docker. Например, мы создали redis-tlsконтейнер Docker, который также был доступен в Docker Hub. Если бы мы хотели использовать тот же образ, нам нужно было бы ссылаться на этот образ как madflojo/redis-tls.

Изображения зданий для организаций

Этот же подход используется при создании образов для репозитория организации. Например, Codeship имеет alpine-dockerрепозиторий под своей учетной записью организации. Если бы мы хотели использовать этот образ, мы бы ссылались на этот образ как codeship/alpine-docker. Организация-имена учетных записей заменяют имя пользователя в имени тега изображения Docker.

Теперь, когда мы поняли, как назвать наш контейнер, давайте продолжим и снова построим контейнер с соответствующим именем тега.

```
$ docker build -t madflojo / push-пример.
```

Вход в Docker Hub

Прежде чем мы запустим и загрузим наш контейнер в Docker Hub, нам сначала нужно будет войти в Docker Hub из нашего интерфейса командной строки. Для этого мы будем использовать docker loginкоманду.

```
$ docker войти

Имя пользователя: madflojo

Пароль:

Войти удалось
```

После того, как мы вошли в систему, теперь мы можем переместить наш контейнер в Docker Hub.

Толкаем наш контейнер

Поскольку самые сложные шаги уже выполнены, наш следующий шаг довольно прост. Все, что нам нужно сделать, это выдать docker pushкоманду с указанием имени тега изображения.

```
$ docker push madflojo/push-пример

Push ссылается на репозиторий [docker.io/madflojo/push-example ]

4393194860cb: Нажал

0011f6346dc8: Нажал

340dc52ed535: Нажал

72073bf3dbb2: Нажал

627a5019997b: Нажал

62924cac48de: Нажал

33f1a94ed7fc: нажал

b27287a6dbce: нажал

47c2386f248c: Нажал

2be95f0d8a0c: нажал

2df9b8def18a: Нажал

последние: дайджест: sha256:
```

```
 f483e14a1c6b7a13bb7ec0ab1c69f4588da2c253e87652329e615d2f8c439abe
размер: 2606
```

После выполнения вышеприведенной команды наш образ теперь доступен в Docker Hub. Чтобы увидеть его, мы можем просто перейти к нашему недавно созданному репозиторию.

Подпишитесь на бесплатную учетную запись Codeship

Перемещение в реестр, не являющийся Docker-Hub

Мы узнали, как создать новое изображение с нашим именем пользователя, включенным в имя тега изображения Docker. Мы также узнали, как войти в систему и отправить этот образ в Docker Hub.

Хотя это полезно для тех, кто использует службу реестра Docker Hub, шаги немного меняются при попытке перейти к внутреннему реестру Docker. К счастью, процесс не сильно меняется.

Указание адреса реестра в имени тега

Точно так же, как мы должны были указать наше имя пользователя при создании образа для Docker Hub, мы должны сначала включить это имя в имя тега для образа Docker, когда мы хотим создать образ для реестра, не связанного с Docker Hub.

Команда ниже показывает создание образа для реестра, работающего на registry.example.com.

$ docker build -t registry.example.com/madflojo/push-example.

Имя следует тому же формату, что и выше, с той разницей, что в него включен адрес реестра Docker.

Войдите в реестр, не являющийся Docker-Hub

docker loginКоманда также немного изменится. Вместо простого запуска docker loginнам нужно будет указать адрес реестра, чтобы сообщить Docker, в какой реестр войти.

$ docker войти registry.example.com

После входа в систему наша docker pushкоманда также меняется, однако не очень сильно.

Используйте полное имя тега с помощью push

Ранее, когда мы выполняли этот docker buildшаг, мы указывали имя изображения registry.example.com/madflojo/push-example. Когда мы

выполним docker pushкоманду, нам также нужно будет указать это имя изображения.

При docker pushвыполнении Docker посмотрит на указанное имя изображения, чтобы определить, в какой реестр загрузить изображение. Когда мы указываемregistry.example.com/madflojo/push-example, он знает, что мы помещаем контейнер с именем push-examplев репозиторий пользователя (madflojo) в реестре registry.example.com. Мы видим это при выполнении docker pushкоманды.

$ docker push registry.example.com/madflojo/push-example

Push ссылается на репозиторий [registry.example.com/madflojo/push-example]

Теперь мы знаем, как отправлять изображения не только в Docker Hub, но и в любой другой реестр, независимо от того, является ли этот реестр общедоступным или частным.



МОДУЛЬ 16. Микросервисы и микросервисная архитектура

1. Введение в микросервисы
2. Микросервисы: принципы
3. Микросервисы: подходы
4. Микросервисы: масштабирование

1. Введение в микросервисы

Микросервисы позволяют разработчикам разделить проект разработки программного обеспечения на набор модульных сервисов.

Замкнутая природа микросервисной архитектуры может упростить создание проектов по разработке программного обеспечения и управление ими. Многие крупные компании, такие как PayPal, Netflix и даже Amazon, используют микросервисы, демонстрируя, что микросервисы могут быть полезным инструментом для разработки масштабируемого программного продукта.

Что такое микросервисы?

Микросервисы — это средства организации приложения с помощью нескольких независимых однофункциональных протоколов или сервисов. Стандартным архитектурным стилем для создания программных приложений является шаблон монолитной архитектуры.

Хотя для многих или большинства приложений монолитная архитектура является адекватным выбором, она имеет свои ограничения. А именно, монолитное приложение поощряет тесную связь, когда многие компоненты программной системы взаимосвязаны. Такая зависимость иногда может быть полезной, но часто это может означать увеличение сложности с течением времени. И изменения в монолитном приложении также обычно негативно влияют на время выполнения и скорость обработки.

В результате масштабируемость становится серьезной проблемой. Взаимозависимость и сложный информационный поток монолитных архитектур в конечном итоге могут стать препятствием для жизненного цикла разработки программного обеспечения и дальнейшего обслуживания вашего программного приложения.

С другой стороны, микросервисная архитектура может развертываться независимо . При правильном применении этот платформенный подход может стимулировать рост бизнеса за счет улучшения процесса разработки программного обеспечения. С микросервисами программные компоненты в значительной степени разъединены, что допускает слабую связь. Манипуляции с одной функцией не влияют на связанную с ней функцию и наоборот.

Формальное определение микрослужб варьируется, поскольку многие характеристики определяют их природу. Микросервисы, как правило, представляют собой процессы, взаимодействующие по сети, автономные, небольшие по размеру и т. д. Точно так же известный инженер-программист

Мартин Фаулер определяет несколько основных характеристик, описывающих микросервисы и/или их использование при разработке программного обеспечения:

- деловые возможности
- автоматическое развертывание
- интеллект конечных точек
- децентрализованный контроль над языками и данными.

Короче говоря, многие из этих атрибутов соответствуют тому факту, что микросервисы обеспечивают непрерывную доставку (CD). В гибкой разработке программного обеспечения компакт-диск гарантирует перестройку и повторное развертывание программного обеспечения для оптимизации конечного продукта. Кроме того, архитектурный шаблон микросервиса основан на сильных сервисно-ориентированных принципах, таких как проектирование, ориентированное на предметную область (DDD). DDD гарантирует, что программное обеспечение соответствует его бизнес-целям.

Как работают микросервисы?

Микросервисы работают, изолируя функциональные возможности программного обеспечения в несколько независимых модулей, которые несут отдельные обязанности. Сотни микросервисов объединяются, чтобы предоставить пользователям более крупный макросервис или набор макросервисов в виде программного приложения.

В связи с постоянно растущим спросом на крупномасштабные приложения микросервисы удовлетворяют потребность в более целенаправленном подходе к разработке программного обеспечения, что способствует росту бизнеса.

Но как работают микросервисы? Хотя фактическая разработка приложения с использованием микросервисной архитектуры довольно сложна, вот краткий обзор концептуальных истоков микросервисов.

Монолитные приложения

Чтобы лучше понять, как работает микросервисная архитектура, вам следует вернуться к концепции монолитной архитектуры. Базовая конструкция монолитного программного приложения состоит из трех уровней — элемента представления или пользовательского интерфейса (UI), бизнес-логики и базы данных.

В то время как фронтенд-разработка включает в себя создание пользовательского интерфейса, бэкэнд-разработчики пишут бизнес-логику и управляют базой данных. Теперь этот стек довольно линейный и ничуть не

кажется сложным. Но в некотором смысле верно и обратное. Существует много точек отказа, которые могут возникнуть, когда только одна центральная система содержит все наиболее важные части вашего программного приложения. Аппаратный сбой или ошибка могут серьезно повредить систему с такой архитектурой, и восстановить ее будет непросто.

Что касается операций на стороне сервера, масштабирование монолитных приложений также влечет за собой использование новых серверов для обработки требуемой дополнительной полосы пропускания и емкости пользователей. Естественно, это приводит к неравномерному балансу ресурсов.

Сервис-ориентированная архитектура (SOA)

Сервис-ориентированная архитектура (SOA) возникла из-за проблем, описанных выше. Контекстуализация функциональности с точки зрения конкретных сервисов, к которым могут обращаться модульные фрагменты кода, приводит к более интуитивно понятным и автономным структурам программного обеспечения. SOA распределяет службы между серверами, объединяя их по сети и полагаясь на интерфейсы прикладного программирования (API) для разделения служб. Балансировка нагрузки, в свою очередь, становится более стабильной и равномерной. SOA позволяет сократить циклы разработки, поскольку небольшие специализированные группы могут сосредоточиться только на одном сервисе за раз.

Микросервисы

Микросервисы — это особый тип сервис-ориентированной архитектуры. Они еще больше разрушают концепцию сервис-ориентированной архитектуры, рассматривая проблемы в гипериндивидуалистической манере. Сотни микросервисов объединяются, чтобы предоставить пользователям более крупный макросервис или набор макросервисов в виде программного приложения. Нетфликс, например. хорошо известна в отрасли переходом от монолитной архитектуры к микросервисной архитектуре с более чем 1000 микросервисов. Большая часть микросервисной архитектуры основана на облаке и использует контейнерные технологии, что упрощает децентрализацию серверов. Терминология контейнеров происходит от упаковочных контейнеров для грузовых перевозок. Предыдущие версии SOA использовали виртуальные машины. Вместо этого контейнеры упаковывают код и его зависимости в отдельную единицу, которую можно, так сказать, отправить в облако.

Вы также можете спроектировать микросервисы так, чтобы они были самовосстанавливающимися, где вы можете отчуждать, устранять неполадки

и устранять проблемы с помощью автоматизации. Конечно, монолитное приложение потребует слишком большого вмешательства человека, чтобы оно могло работать в массовом масштабе.

Каковы основные области применения микросервисов?

Вы можете применить микросервисы к своей программной архитектуре по ряду причин. Ниже приведены несколько примеров, когда микросервисы могут подойти.

Небольшие команды

Микросервисная архитектура служит подходящим дополнением для небольших групп разработчиков программного обеспечения.

Поскольку разработка программного обеспечения сводится к созданию нескольких самодостаточных сервисов, вы можете выбрать нескольких опытных разработчиков и назначить их для создания определенного сервиса, который необходимо сделать. В целом, это снижает утечку ресурсов, которая неизбежна, когда внутренний персонал становится затруднительным или если вы работаете с ограниченным бюджетом.

Тестирование и развертывание

Развертывание и тестирование программного обеспечения, использующего микросервисную архитектуру, откровенно менее сложно, чем то же самое с монолитным приложением. Конечно, тестирование и отладка монолитного приложения требует одновременного запуска всего приложения, поиска ошибок вручную и их устранения таким образом, чтобы исправления не оказывали негативного влияния на другие части вашего кода. Разработчики должны следить за последним обстоятельством, поскольку монолитные приложения по определению взаимозависимы, поэтому каждое изменение может создать эффект бабочки. Выполнение обновлений и обновлений вашего приложения может быть таким же разочаровывающим по тем же причинам. Эти ограничения также усложняют масштабирование, что должно быть серьезной проблемой практически для любого бизнеса.

Разнообразные технологические стеки

Используя микросервисы, вы сможете использовать более одного стека технологий. Например, одна небольшая команда может работать над сервисом Python, а другая может использовать Ruby. Это дает вам больше свободы действий при найме разработчиков и принятии решения о том, какие языки программирования и инструменты разработки лучше всего использовать. Универсальный подход удобен для небольших приложений, но более крупные проекты обычно требуют большей универсальности. В разработке программного обеспечения это позволяет исследовать и

использовать различные языки программирования, фреймворки, базы данных и так далее.

6 характеристик микросервисов

Микросервисы и их непосредственные преимущества можно разделить на несколько ключевых характеристик.

1. Несколько компонентов

Опять же, основополагающее качество микросервисной архитектуры заключается в том, что команда разработчиков структурирует программное обеспечение в сервисы-компоненты. Поэтому сервисы могут функционировать независимо друг от друга.

2. Создан для бизнеса

Повторяя Мартина Фаулера, бизнес-возможности центра микросервисов . По сравнению с монолитными приложениями микросервисы повышают эффективность за счет повышения эффективности небольших групп.

3. Простая маршрутизация

Микросервисы получают запросы и отвечают на них. Просто, верно? Что ж, монолитное приложение активирует череду зависимых уровней приложения, и функциональные возможности могут сталкиваться друг с другом для достижения одной и той же конечной цели.

4. Децентрализованный

Управление данными децентрализовано. Для поклонников микросервисов они считают это более безопасным. Повторное использование также является побочным продуктом децентрализации, поскольку автономные сервисы решают определенные проблемы.

5. Устойчивость к сбоям

Откровенно неправдоподобно, чтобы все сервисы в программной системе микросервисов вышли из строя одновременно. Конечно, одна служба может выйти из строя, но это никак не повлияет на соседнюю службу.

6. Эволюционный

Микросервисы могут расти вместе с вашим бизнесом. Добавление новых функций происходит за пределами монолитной системы, что полностью упрощает масштабируемость.

Каковы плюсы и минусы микросервисов?

Плюсы и минусы микросервисов важно учитывать, прежде чем принимать какое-либо важное решение о том, как приступить к созданию вашего следующего программного проекта. Микросервисы предлагают относительно новую методологию разработки программного обеспечения.

Хотя абстракция создания приложений захватывающая, вам все равно нужно критически относиться к новым способам разработки, какими бы популярными они ни были.

Плюсы микросервисов

Плюсы микросервисов должны быть очевидны, учитывая основные области применения архитектуры микросервисов, но, возможно, стоит более подробно рассмотреть, на что способны микросервисы.

Простота

Многие аспекты микросервисной архитектуры способствуют ее простоте. Во-первых, преднамеренное распределение функций, лежащих в основе основ микросервисов, означает более компактное развертывание кода.

Следовательно, написание кода для каждой функции или службы намного проще и практически безболезненно, когда разработчикам не нужно учитывать зависимости функций.

Объем проекта меньше, чем в противном случае, поэтому разработчики могут использовать непрерывное развертывание благодаря повышению производительности.

Как упоминалось выше, тестирование также использует преимущества микросервисной архитектуры, поскольку разработчики могут быстрее изолировать ошибки. И, конечно же, качественный программный продукт лучше делать раньше, чем позже.

Масштабируемость

Категориальная группировка функций, которую позволяет микросервисная архитектура, означает, что вы можете масштабировать одну или несколько служб, когда придет время, и сэкономить на масштабировании всего приложения. С помощью этой процедуры вы можете адекватно планировать ресурсы и соответственно экономить затраты.

Минусы микросервисов

Минусы микросервисов сложнее расшифровать, тем более что у архитектуры так много преимуществ. Но есть еще некоторые моменты, на которые следует обратить внимание, прежде чем принять решение об использовании микросервисной архитектуры для своего бизнеса.

Требовательность

Микросервисы требуют нескольких баз данных и серверов, а также большого количества операций по управлению транзакциями. Действительно, внедрение микросервисной архитектуры потребует значительных ресурсов . Как следствие, крупные компании довольно хорошо

адаптируются к микросервисам, а небольшие компании с трудом осваивают инфраструктуру. Правильная организация микросервисной архитектуры имеет первостепенное значение, но, учитывая ее исчерпывающие потребности, это более напряженная задача, чем готовы некоторые предприятия.

Комплекс

Может показаться странным, что сложное и простое описывают одно и то же понятие, но вы скоро поймете. Хотя микросервисная архитектура может упростить общую разработку вашего приложения, службы, которые вы создаете независимо друг от друга, в какой-то момент должны встретиться. Связь между каждой службой может быть сложно настроить, и вам, возможно, придется написать дополнительный код, чтобы гарантировать, что каждая модель может подключаться друг к другу без ошибок.

Микросервисы и API: в чем разница?

Основное различие между микросервисами и API заключается в их соответствующих функциях и классификациях.

Например, API определяет протоколы для интеграции двух или более программных систем . Вот несколько примеров API:

Порталы входа, которые полагаются на различные социальные сети, чтобы вы могли создавать или передавать информацию об учетной записи, как при входе с использованием вашей учетной записи Google в несвязанное приложение или веб-сайт.

Google извлекает информацию из местных онлайн-ресурсов, когда вы ищете ближайший ресторан или мероприятие.

Видео Youtube, встроенные на веб-страницу, отличную от самой Youtube

И наоборот, микросервисная архитектура описывает конкретный метод разработки программных приложений, разбивая функции на автономные сервисы.

Микросервисы зависят от API для подключения этих сервисов, но сами по себе они не являются API.

Многие крупные компании разрабатывают и поддерживают свое программное обеспечение с помощью микросервисной архитектуры, в том числе:

eBay

Саундклауд

Убер

Групон

Непонятно, почему микросервисы и API кажутся похожими или эквивалентными, но у них совершенно разные роли в мире разработки программного обеспечения.

Заключение

Микросервисы или микросервисная архитектура могут усовершенствовать ваш процесс разработки программного обеспечения в лучшую сторону. Благодаря микросервисной архитектуре разработчики могут мысленно и буквально создавать гибкие, функционально-ориентированные сервисы. Сервис-ориентированная архитектура, как правило, выгодна для разработки программного обеспечения, хотя бы концептуально. В то время как сборка монолитных приложений в некоторых случаях не оправдывает ожиданий. Тем не менее, у микросервисов есть свои плюсы и минусы, и очень важно знать о них.

2. Микросервисы: принципы

Принципы складываются из утверждений о том, как все должно делаться, и из разъяснения причин, по которым все должно делаться именно таким образом. Они помогают нам выстроить различные решения, которые приходится принимать при создании наших систем. Конечно же, вы должны определить собственные принципы, но есть смысл изложить видение того, как выглядят ключевые принципы микросервисных архитектур, обобщенный вид которых представлен на рис. Эти принципы призваны помочь нам в создании небольших автономных сервисов, успешно выполняющих совместную работу.

Вы можете принять за основу сразу все эти принципы или же подстроить их под то, что имеет смысл принять именно в вашей организации. Но обратите внимание на ценность их использования в сочетании друг с другом: целостность может быть гораздо ценнее суммы частей. Итак, если вы решили отвергнуть один из них, убедитесь в том, что понимаете, что теряете.



Принципы микросервисов

Модель, построенная вокруг бизнес-концепций

Опыт подсказывает, что интерфейсы, структурированные вокруг контекстов, связанных с бизнес-задачами, отличаются большей стабильностью, чем те, которые структурированы вокруг технических

концепций. Моделированием областей, в которых работает наша система, мы не только пытаемся создать более стабильные интерфейсы, но и обеспечиваем повышение возможности более свободно отображать изменения в бизнес-процессах. Для определения потенциальных границ областей следует использовать ограниченный контекст.

Внедрение культуры автоматизации

Микросервисы усложняют системы главным образом из-за необходимости работать с огромным количеством активных компонентов. Применение культуры автоматизации является одним из основных способов справиться с этой проблемой, и в этом плане имеет смысл как можно раньше приложить максимум усилий к созданию инструментария для поддержки микросервисов. Поскольку удостовериться в работоспособности сервисов по сравнению с монолитными системами намного сложнее, важную роль в этом процессе играет автоматизированное тестирование.

Также может помочь использование одинакового вызова командной строки для повсеместного единообразного развертывания, который может стать основной частью внедрения непрерывной поставки с целью обеспечения быстрой оценки качества работы в производственном режиме при каждой проверке.

Подумайте о возможности использования определений среды в качестве вспомогательного средства конкретизации отличий одной среды от другой без ущерба для возможности использования единообразного метода развертывания. Также подумайте о создании настраиваемых образов, позволяющих ускорить развертывание и включающих в себя создание полностью автоматизированных неизменяемых серверов, упрощающих понимание устройства ваших систем.

Скрытые подробности внутренней реализации

Чтобы максимально увеличить возможность одного сервиса совершенствоваться независимо от всех других сервисов, крайне важно скрыть подробности его реализации. Содействие в этом могут оказать ограниченные контексты, поскольку они помогают сконцентрировать усилия на моделях, предназначенных для совместного использования и при этом обладающих скрытностью. Сервисы также должны скрывать свои базы данных во избежание попадания в одну из наиболее распространенных разновидностей связывания, которая может проявляться в традиционных сервис-ориентированных архитектурах и использовать для объединения данных от нескольких сервисов с целью создания отчетов перекачку данных или перекачку данных о событиях.

При любой возможности отдавайте предпочтение применению технологически нейтральных API-интерфейсов, чтобы получить свободу использования различных технологических стеков. Подумайте о применении технологии передачи репрезентативного состояния REST, оформляющей разделение подробностей внутренней и внешней реализации which. Те же самые идеи можно принять за основу и при использовании удаленных вызовов процедур (RPC).

Всесторонняя децентрализация

Для достижения максимальной автономности, допускаемой микросервисами, нужно постоянно выискивать возможности для передачи полномочий по принятию решений и управлению тем командам, которые владеют микросервисом. Этот процесс начинается с внедрения самообслуживания везде, где только можно, позволяя людям развертывать программные средства по мере надобности, делая как можно проще разработку и тестирование и исключая необходимость для отдельных команд выполнять эти действия.

Обеспечение того, что команды владеют своими собственными сервисами, является важным шагом на данном пути, возлагающим на команды ответственность за вносимые изменения и в идеале даже за принятие решения о том, когда именно выпустить такие изменения. Чтобы обеспечить возможность внесения изменений в сервисы, находящиеся во владении других команд, воспользуйтесь семейственным открытым кодом, но при этом помните, что для его реализации нужно будет поработать. Подстраивайте команды под организацию, чтобы гарантировать, что закон Конвея работает на вас и помогает вашей команде приобрести специализацию в создаваемых ею сервисах, сконцентрированных на решении конкретных бизнес-задач. Если требуется какая-либо всеобъемлющая ориентация, попробуйте воспользоваться моделью общего руководства, при которой представители разных команд несут коллективную ответственность за развитие технической концепции системы.

Этот принцип применим и для архитектуры. Избегайте подходов вроде сервисных шин предприятия или оркестровых систем, которые могут привести к централизации бизнес-логики и появлению безмолвных сервисов. Чтобы гарантировать наличие соответствующей логики и данных на границах сервисов, которые помогают достигать нужного сцепления, отдавайте предпочтение хореографическому, а не оркестровому принципу и безмолвным промежуточным программным средствам с интеллектуальными конечными точками.

Независимое развертывание

Нужно всегда стремиться к тому, чтобы микросервисы могли развертываться и развертывались самостоятельно. Даже когда требуются изменения, нарушающие общий режим работы, нужно добиваться совместного существования конечных точек разных версий, позволяя потребителям со временем внести изменения.

Тем самым мы сможем оптимизировать скорость выпуска новых свойств, а также повысить автономность работы команд, владеющих этими микросервисами, снимая с них обязанность постоянного согласования развертываний своих сервисов.

Если используется интеграция на основе RPC, избегайте при создании заглушек тесной связанности между клиентом и сервером вроде той, что поддерживается технологией Java RMI.

Принимая модель «по одному сервису на каждом хосте», вы уменьшаете число побочных эффектов, которые могут при развертывании одного сервиса влиять на другой, не связанный с ним сервис. Чтобы отделить развертывание от выпуска, сократив тем самым риск неудачного выпуска, присмотритесь к использованию технологий сине-зеленого развертывания или канареечного выпуска. Чтобы отловить изменения, нарушающие работу системы, еще до того, как они на нее смогут повлиять, воспользуйтесь контрактами на основе запросов потребителей.

Запомните, что возможность внесения изменений в отдельно взятый сервис и его выпуска для работы в производственном режиме без необходимости развертывания других сервисов с приостановкой работы системы должна стать нормой, а не исключением. Ваши потребители должны сами решать, когда им нужно обновляться, а вам следует к этому приспособиться.

Изолирование сбоев

Архитектура микросервисов может быть более устойчивой, чем монолитная система, но только если мы разбираемся в ситуации и имеем план на случай сбоев в части нашей системы. Если возможность и неизбежность сбоев вызова нижестоящего сервиса в расчет не принимаются, наша система может испытать катастрофический каскадный сбой, и мы останемся с системой, еще более хрупкой, чем прежняя.

При использовании сетевых вызовов не уподобляйте удаленные вызовы локальным, так как из-за этого будут скрываться разные виды отказов.

Поэтому убедитесь, что при использовании клиентских библиотек не происходит чрезмерного применения удаленных вызовов.

Не нужно постоянно думать о том, что система должна быть устойчивой к сбоям, и при этом всегда и всюду ожидать сбоев. Обеспечьте соответствующую настройку лимитов времени. Разберитесь, когда и как для ограничения выхода из строя компонентов использовать перегородки и предохранители. Разберитесь в том, какое влияние окажет на клиента неправильная работа всего одной из частей системы. Узнайте, какими окажутся последствия разделения сети и будет ли правильно требовать в этой ситуации принести в жертву доступность или согласованность данных.

Всестороннее наблюдение

При оценке правильности функционирования системы мы не можем полагаться на наблюдение за поведением одного-единственного экземпляра сервиса или состоянием одной машины. Нам требуется более широкое представление о происходящем. Чтобы убедиться в правильном поведении системы, нужно использовать семантический мониторинг путем внедрения в вашу систему искусственных транзакций с целью имитации поведения реального пользователя. Следует объединить свои журналы и объединить статистические данные, чтобы при обнаружении проблемы можно было докопаться до ее источника. А когда дело дойдет до воспроизведения неприятных моментов в поведении системы или наблюдения за тем, как в вашей системе происходит взаимодействие компонентов в производственном режиме работы, используйте идентификаторы взаимосвязанности, позволяющие проводить трассировку вызовов внутри системы.

А когда не следует применять микросервисы?

Первая часть совета будет такой: чем хуже вы разбираетесь в заданной области, тем сложнее вам будет правильно определить ограниченные контексты для сервисов. Как уже говорилось, неправильное определение границ сервисов может привести к необходимости внесения множества изменений в совместную работу сервисов, что является весьма затратным делом. Поэтому, если вы взялись за монолитную систему, область применения которой вам непонятна, сначала уделите время изучению того, чем эта система занимается, а затем, прежде чем разбивать ее на сервисы, присмотритесь к определению четких границ модулей.

Разработка с чистого листа также дается весьма непросто. И дело даже не в том, что область, вероятно, также будет новой, а в том, что разбить на части что-либо уже существующее гораздо проще, чем что-то, чего у вас еще

нет! Итак, повторим: присмотритесь сначала к созданию монолитной системы, а ее разбиением займитесь, как только добьетесь от нее стабильной работы.

Сложности, с которыми вы столкнетесь при работе с микросервисами, при масштабировании существенно усугубятся.

А когда не следует применять микросервисы?

При наличии одного-двух сервисов у вас все может получиться, ну а если сервисов будет пять или десять? Если придерживаться старых положений по мониторингу, где просто изучается статистика загруженности центрального процессора и памяти, этого может хватить только для работы с весьма небольшим количеством сервисов, но чем шире используется совместная работа сервисов, тем обременительнее станет процесс. Вас это начнет беспокоить по мере добавления сервисов, приведенные советы помогут вам заметить возникновение ряда подобных проблем и подскажут их конкретные решения. Компании REA и Gilt потратили время на создание инструментария и инструкций по управлению микросервисами еще до появления возможности их использования в большом количестве. Эти истории только подтвердили важность постепенного расширения, чтобы разобраться в потребностях организации и возможностях внесения изменений, которые помогут вам внедрить микросервисы надлежащим образом.

Архитектуры микросервисов предоставляют вам более широкий выбор и возможность принятия большего количества решений. Принимать решение в этой области приходится намного чаще, чем в области монолитных систем. Абсолютно все решения правильными не бывают, это я заявляю со всей ответственностью.

Следовательно, каков должен быть наш выбор, зная, что мы что-то поймем неправильно? Посоветуем найти способы принятия каждого решения в самых скромных масштабах. Тогда, если оно окажется неверным, это повлияет лишь на небольшую часть вашей системы. Научитесь следовать концепции развивающейся архитектуры, при которой ваша система будет способна гибко подстраиваться под ситуацию и изменяться по мере обнаружения новых обстоятельств. Не замышляйте грандиозной перезаписи кода, лучше вместо этого сохраняйте гибкость системы путем серии изменений, вносимых в вашу систему в течение длительного времени.

3. Микросервисы: подходы

Клиент-серверная архитектура

По мере роста объема работ, возникают проблемы с синхронизацией работ, например при работе над интерфейсом и логикой проекта. Обычно, это зоны ответственности разных специалистов, и при интенсивной работе неизбежно возникает сильная зависимость от результата работ друг друга, а также конфликты при работе с одной и той же функциональной частью системы. Следствие этих конфликтов — необходимость более объемного тестирования даже после небольших изменений, снижение частоты релизов.

Эти и другие проблемы — следствие «монолитной» архитектуры системы. Монолитная архитектура — архитектура, реализованная внутри одного программного модуля (одной программы).

Монолитная архитектура удобнее для одного разработчика или микрокоманд из 2–4 человек, т. к. требует меньше издержек и рутинных действий. Когда команда растет, а как следствие растет число параллельных потоков разработки, монолитная архитектура становится узким местом и не позволяет эффективно организовать процесс.

Чтобы развиваться дальше, мы делаем шаг в сторону распределенной архитектуры, где различные компоненты информационной системы функционируют физически раздельно, используя для взаимодействия согласованный набор команд, называемый API (Application Programming Interface).

До этого момента у нас была единая система, являющаяся одним программным модулем, использующим единые логические компоненты. С этого момента она будет разделена на отдельные подсистемы, но для начала только на две:

- «back-end» — серверная часть, реализующая основную ценную логику системы
- «front-end» — клиентская часть, предоставляющая пользователю интерфейс для работы с системой.
-



Система с разделенным front-end и back-end.

Данный подход позволяет выделить в одну программную систему все, что касается пользовательского интерфейса, а в другую — полезную логику, расчеты, работу с данными. Кроме этого, единый механизм взаимодействия с back-end частью дает возможность делать дополнительные front-end подсистемы, например мобильные приложения.

Плюсы

- Больше нет связности пользовательского интерфейса и логики. Разнопрофильные разработчики теперь могут работать независимо;
- Рост качества, т. к. каждый ответственен за свою подсистему и более погружен в нее;
- API подходит как для веб, так и для мобильных приложений'
- Удобнее масштабировать под высокие нагрузки. За счет изоляции бизнес-логики в отдельный физический модуль, его проще горизонтально/вертикально масштабировать.

Минусы

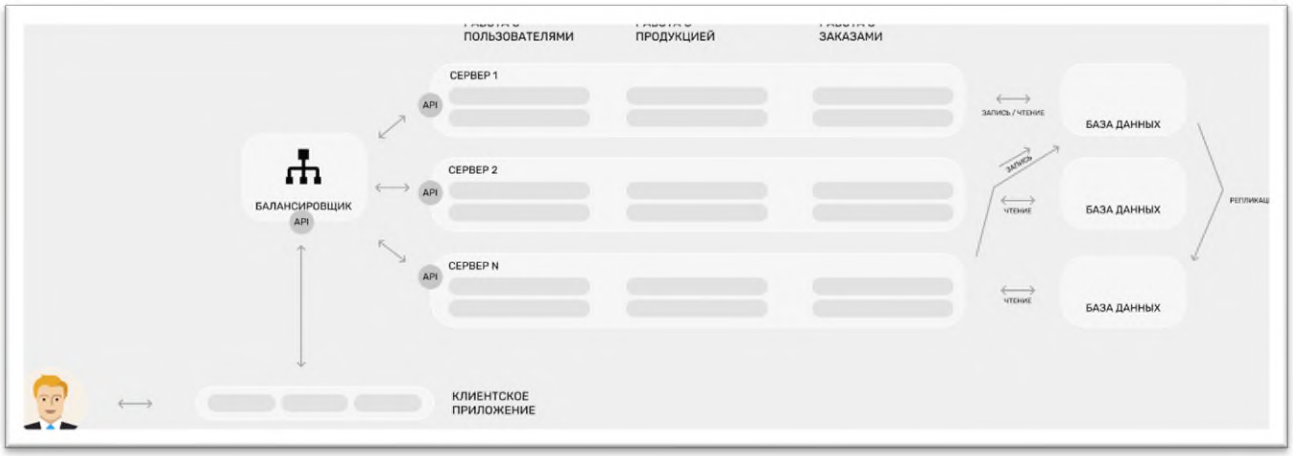
- Рост команды. Технологии разработки фронт-енд и бек-енд систем различны и при разделении обычно требуется наличие отдельного специалиста или команды на каждую подсистему;
- Дороже за счет роста команды. Для проектов с постоянной интенсивной работой и релизами, этот подход будет эффективнее, но для редко обновляемых проектов затраты будут выше.

Рост нагрузок и горизонтальное масштабирование

После того, как мы разделили проект на отдельные логические единицы, уровни абстракции и реализовали клиент-серверную архитектуру, нам стало намного удобнее вносить правки и развивать продукт.

По мере роста нагрузки встает необходимость оптимизировать проект под них. Основной принцип оптимизации под высокие нагрузки — горизонтальное масштабирование. Подробно как это работает мы опишем в отдельной статье. В контексте данной статьи надо иметь в виду, что горизонтальное масштабирование технически возможно при любой архитектуре, но наиболее эффективно это реализуется при распределенной архитектуре, начиная от клиент-серверной архитектуры.

При текущей архитектуре мы масштабируем бекенд на несколько серверов. Простыми словами, разврачиваем копию серверного приложения на нескольких серверах и учим его работать параллельно. Физически это могут быть как отдельные арендованные сервера, так и облачные мощности, автоматически развертывающие дополнительные параллельные экземпляры при росте нагрузок.



Упрощенный пример горизонтального масштабирования при клиент-серверной архитектуре

С помощью горизонтального масштабирования мы можем справиться с большими нагрузками, но это усложняет обслуживание системы, выкатку обновлений. Общая логика системы усложняется, взаимодействие с инфраструктурными системами становится более изощренной. Эффективность снижается.

Важно понимать, что слово «эффективность» в разработке ПО не является аналогом «дешевой» или «экономичной» и в первую очередь относится к организации процесса разработки, т. е. к людям. Эффективность оценивается в контексте развития системы при затрачиваемых ресурсах. Основное — скорость и прогнозируемость разработки программного обеспечения на длинной дистанции.

В целом, для 90–95% проектов имеет смысл остановиться на этом этапе. Но есть и более сложный вариант развития архитектуры, дающий высокую эффективность на проектах с десятками и сотнями людей в команде и большими нагрузками. Как? Все то же самое — делим и изолируем.

Микросервисная архитектура

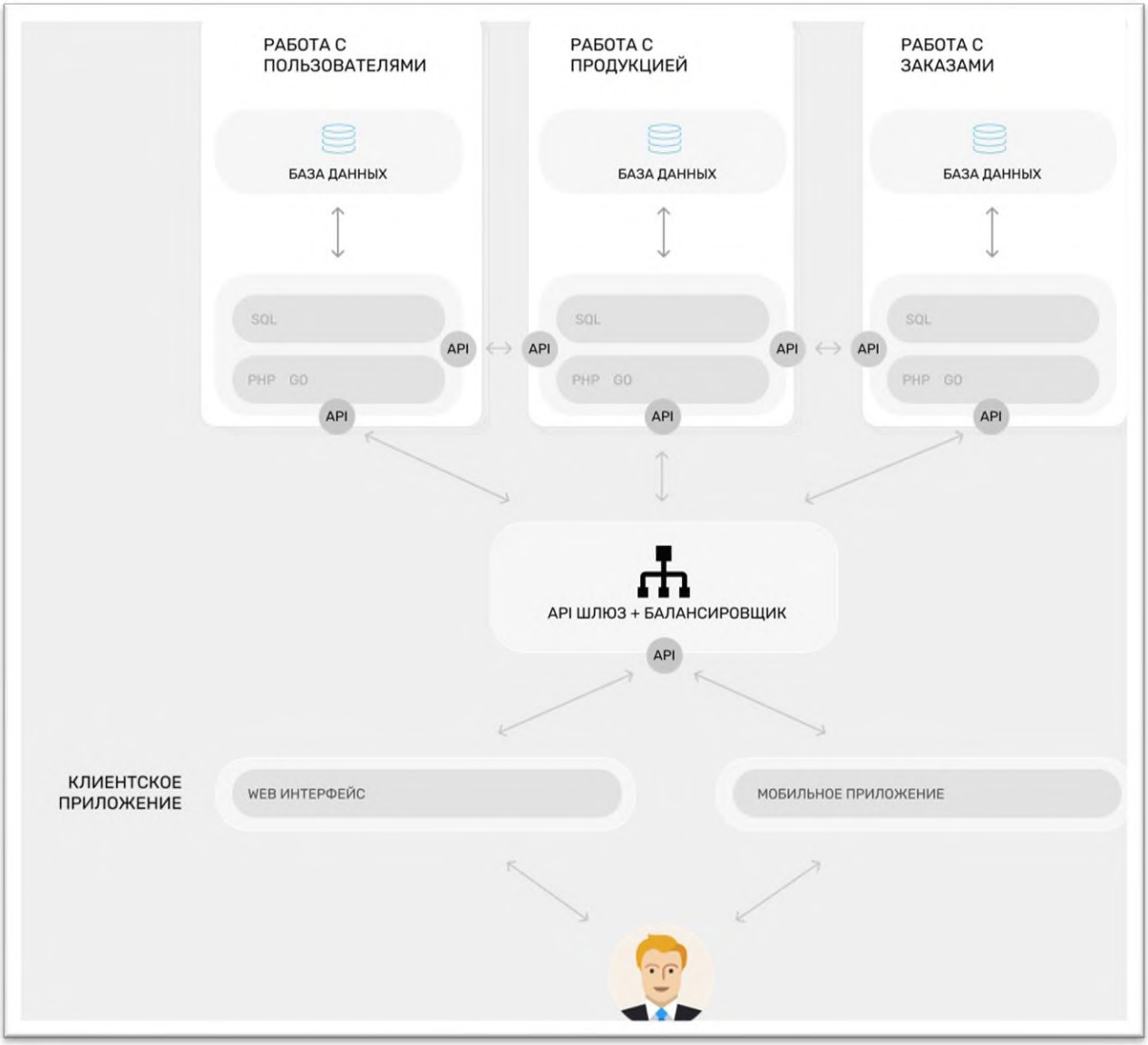
Следующей стадией развития является физическое разделение серверного звена сисиемы (бекенда) на независимые приложения, называемые сервисами или микросервисами.

Ранее, при вводе клиент-серверной архитектуры мы уже разделили клиентскую (пользовательский интерфейс) часть и серверную, получив так называемую «трехзвенную архитектуру» (клиент-логика-данные). Теперь мы делим серверную на отдельные подсистемы, каждая из которых обслуживает свою функциональную область или даже отдельные функции.

Принцип деления на сервисы может быть различным и зависит от особенностей проекта, распределения нагрузки и других вводных. В некоторых случаях, даже состав команды и возможности по найму имеют некоторое значение при выборе принципа разделения на сервисы.

Разделение, может быть, на основную ведущую систему и вспомогательные сервисы (например, основная логика остается в одном бекенд-приложении, а вспомогательные функции выносится в отдельные сервисы). Также можно разделить систему на более-менее равнозначные сервисы, каждый из которых отвечает за свою функциональную область.

Одно из важных преимуществ микросервисной архитектуры заключается в том, что взаимодействие между сервисами осуществляется с помощью API, которое согласовывается отдельно и меняется редко. Этот принцип чем-то похож на длительные договорные отношения: вы один раз заключили общий договор, описывающий принцип получения каких-нибудь услуг на длительный срок, и Вам не важно, что и как делает подрядчик, чтобы выполнять свои обязательства по условиям договора. Важно, что взаимодействие будет предсказуемо.



Упрощенный пример микросервисной архитектуры

Микросервисную архитектуру хорошо применять там, где она действительно повысит эффективность разработки. Делать проект командой из 2–4 человек на микросервисной архитектуре — ошибочное и неэффективное решение, т. к. значительный ресурс будет уходить на инфраструктурные издержки, а не на основную бизнес-логику.

Разделять на микросервисы функционирующий проект сложно, но это наиболее правильный путь, т. к. к этому моменту обычно уже есть понимание и об узких местах, и о профиле нагрузки, и о планах развития.

Микросервисная архитектура позволяет эффективно организовать работу большой команды и дает возможность выделить отдельные небольшие команды для каждого сервиса, которые будут заниматься только им. Такой подход существенно упрощает и повышает эффективность

управления разработкой крупных интернет-проектов и корпоративных систем.

Благодаря взаимодействию сервисов через API, команды могут спокойно заниматься развитием своего микросервиса, выкатывая релизы без необходимости согласования и синхронизации с другими участниками (API не меняется, следовательно, для других сервисов тоже ничего не изменится).

Кроме того, микросервисная архитектура позволяет эффективнее использовать облачные вычислительные мощности. Микросервис, осблуживающий большую нагрузку, удобно масштабировать с помощью облачных контейнеров. Это позволяет по запросу выделять необходимое количество вычислительной мощности для нужд микросервиса.

Резюме

Для каждой ситуации специфичен свой вариант построения архитектуры информационной системы. Для новых проектов и небольших команд больше подойдет монолитная или клиент-серверная архитектура с единым серверным приложением. Далее, по мере развития проекта и роста нагрузок следует рассматривать переход на микросервисную архитектуру, с целью разделения как потоков разработки, так и выделения наиболее нагруженных функций для дальнейшего масштабирования.

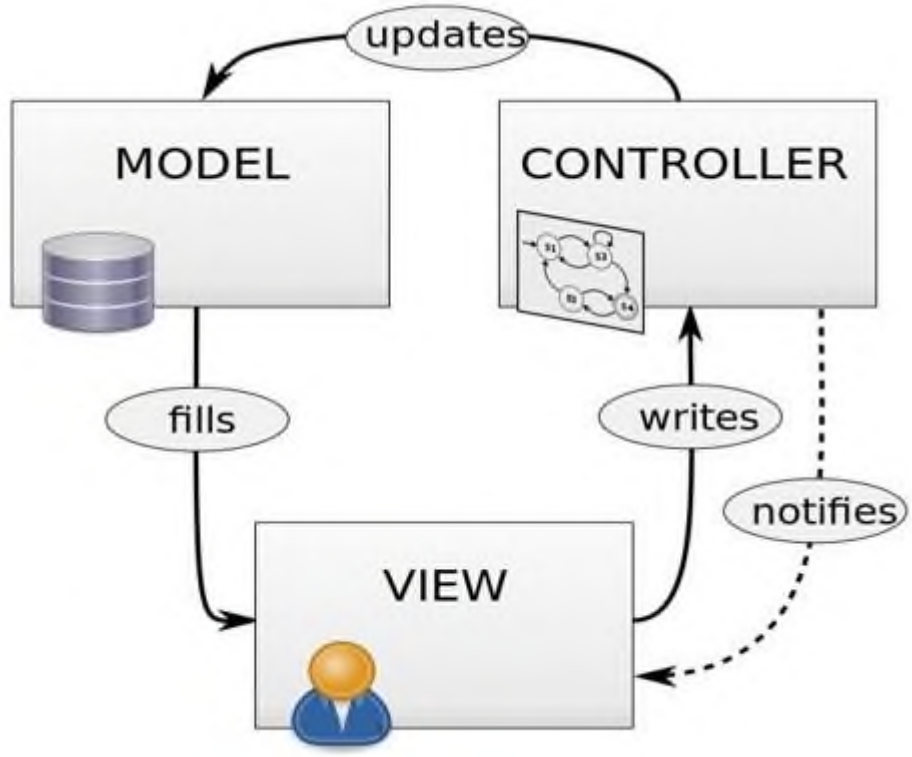
Делать проект командой из 2–4 человек изначально на микросервисной архитектуре — неправильное и малоэффективное решение, т. к. больше ресурсов будет уходить на инфраструктурные издержки. Для старта проекта, особенно на этапе MVP версии, подойдет монолитная MVC (или другой паттерн) или клиент-серверная архитектура в зависимости от специфики. В случае развития проекта можно будет рассматривать переход на микросервисную архитектуру. Для функционирующего проекта это сложно, но это лучшее решение, т. к. к этому времени есть понимание и об узких местах, и в целом о функциональности и планах развития.

4. Микросервисы: масштабирование

Масштабирование — это процесс разбиения программного обеспечения на разные единицы. Масштабирование также определяет масштабируемость. Масштабируемость — это потенциал для реализации более продвинутых функций приложения. Это помогает повысить безопасность, долговечность и ремонтопригодность приложения. У нас есть три типа процедур масштабирования, которым следуют в отраслях. Ниже приведены различные методологии масштабирования вместе с соответствующими примерами из реальной жизни.

Масштабирование по оси X

Масштабирование по оси X также называется горизонтальным масштабированием. В этой процедуре все приложение подразделяется на различные горизонтальные части. Обычно любой веб-сервер приложение может иметь этот тип масштабирования. Рассмотрим обычную архитектуру MVC, которая следует горизонтальному масштабированию, как показано на следующем рисунке.



В качестве примера можно рассмотреть любое приложение-сервлет JSP. В этом приложении контроллер управляет каждым запросом и при необходимости генерирует представление, связываясь с моделью. Обычно монолитные приложения следуют этому методу масштабирования.

Масштабирование по оси X очень простое по своей природе и занимает гораздо меньше времени. В этой методологии одно программное обеспечение будет масштабироваться в зависимости от его различных задач, за которые отвечает подразделение. Например, контроллер отвечает за управление входящим и исходящим запросом, представление отвечает за представление бизнес-функциональности пользователям в браузере, а модель отвечает за хранение наших данных и работает как база данных.

Масштабирование по оси Y

Масштабирование по оси Y также называется вертикальным масштабированием, которое включает в себя любое масштабирование уровня ресурсов. Любая система DBaaS или Hadoop может считаться масштабируемой по оси Y. При этом типе масштабирования запрос пользователя перенаправляется и ограничивается реализацией некоторой логики.

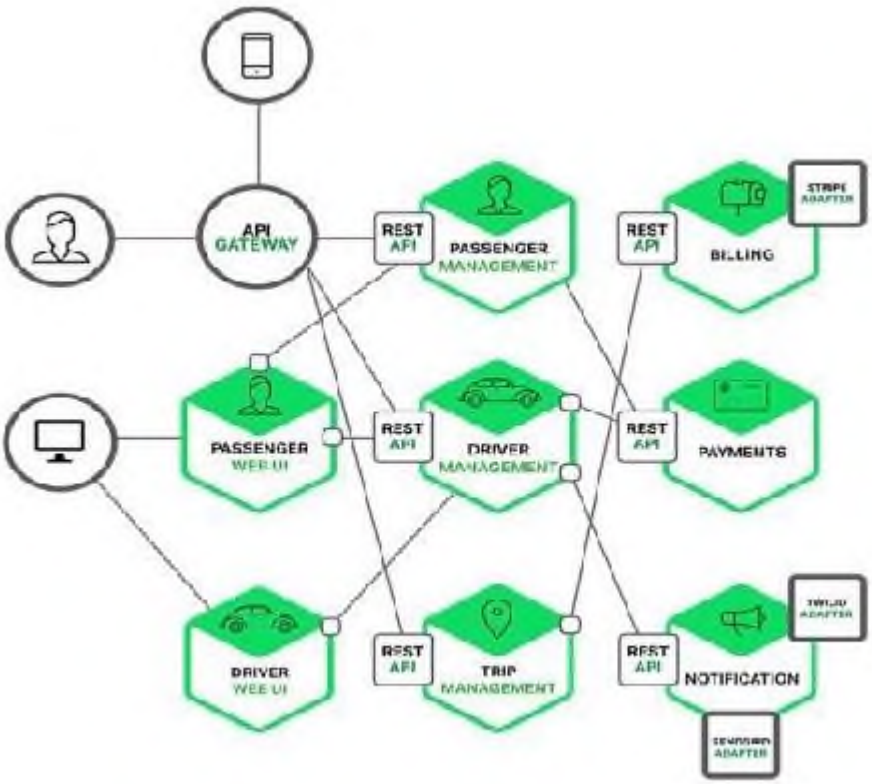
Рассмотрим VK в качестве примера. VK должен обрабатывать 1,79 миллиона пользователей каждую секунду; следовательно, контроль трафика - огромная ответственность сетевых инженеров VK. Чтобы избежать любой опасности, они следуют масштабированию по оси Y, которое включает одновременный запуск нескольких серверов с одним и тем же приложением. Теперь, чтобы контролировать этот огромный уровень трафика, VK перенаправляет весь трафик из одного региона на определенный сервер, как показано на рисунке. Эта передача трафика на основе региона называется балансировкой нагрузки на архитектурном языке.



Этот метод разбивки ресурсов на небольшие независимые бизнес-единицы известен как масштабирование по оси Y.

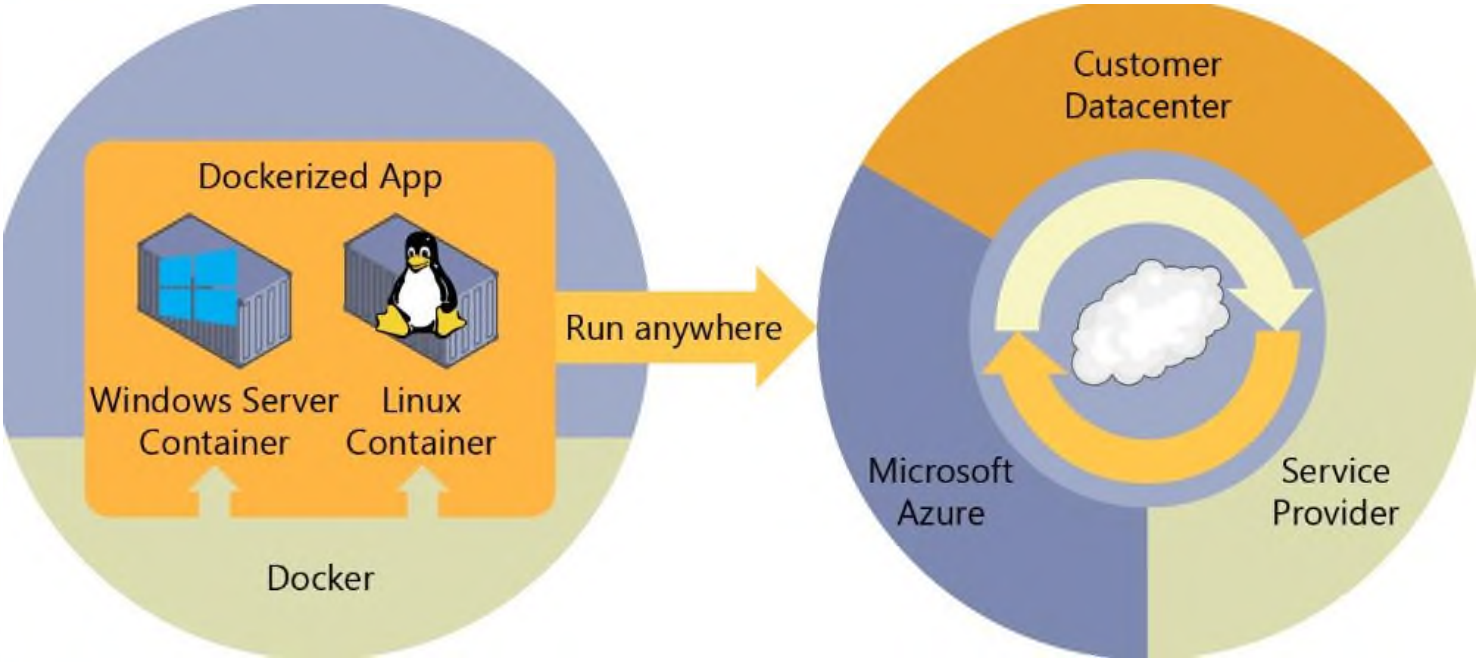
Масштабирование по оси Z

Масштабирование по осям X и Y гораздо проще для понимания. Однако одно приложение также может быть масштабировано на бизнес-уровне, что называется масштабированием по оси Z. Ниже приведен пример масштабирования приложения cab service в различных вертикалях бизнес-единиц.



Преимущества масштабирования

- **Стоимость** − Правильное масштабирование программного обеспечения снижает затраты на обслуживание.
- **Производительность** − Из-за слабой связи производительность правильно масштабируемого программного обеспечения всегда лучше, чем не масштабируемого программного обеспечения.
- **Распределение нагрузки** − Используя различные технологии, мы можем легко поддерживать нагрузку на наш сервер.
- **Повторное** использование − масштабируемость программного обеспечения также повышает удобство использования программного обеспечения.



Модуль 17. Оркестровка контейнеров, кластеры Kubernetes.

- Компоненты Kubernetes: api-server, scheduler, kubelet, etcd
- Команды для работы с Kubernetes
- Развертывание кластера на собственных серверах
- Сетевые решения CNI
- Service mesh. Знакомство с Istio и Envoy
- Контейнеры, поды, deployment, statefulset, services, endpoints Kubernetes

1. Компоненты Kubernetes: api-server, scheduler, kubelet, etcd

Что такое Kubernetes

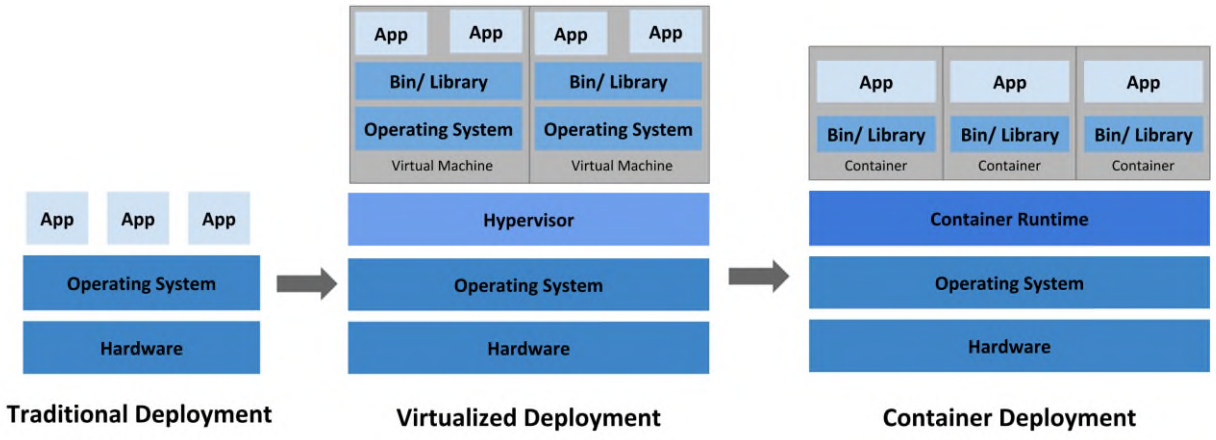
Эта страница посвящена краткому обзору Kubernetes.

Kubernetes — это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию. У платформы есть большая, быстро растущая экосистема. Сервисы, поддержка и инструменты Kubernetes широко доступны.

Название Kubernetes происходит от греческого, что означает рулевой или штурман. Google открыл исходный код Kubernetes в 2014 году. Kubernetes основывается на десятилетнем опыте работы Google с масштабными рабочими нагрузками, в сочетании с лучшими в своем классе идеями и практиками сообщества.

История

Давайте вернемся назад и посмотрим, почему Kubernetes так полезен.



Традиционная эра развертывания: Ранее организации запускали приложения на физических серверах. Не было никакого способа определить границы ресурсов для приложений на физическом сервере, и это вызвало проблемы с распределением ресурсов. Например, если несколько приложений выполняются на физическом сервере, могут быть случаи, когда одно приложение будет занимать большую часть ресурсов, и в результате чего другие приложения будут работать хуже. Решением этого было запустить каждое приложение на другом физическом сервере. Но это не масштабировалось, поскольку ресурсы использовались не полностью, из-за чего организациям было накладно поддерживать множество физических серверов.

Эра виртуального развертывания: В качестве решения была представлена виртуализация. Она позволила запускать несколько

виртуальных машин (ВМ) на одном физическом сервере. Виртуализация изолирует приложения между виртуальными машинами и обеспечивает определенный уровень безопасности, поскольку информация одного приложения не может быть свободно доступна другому приложению.

Виртуализация позволяет лучше использовать ресурсы на физическом сервере и обеспечивает лучшую масштабируемость, поскольку приложение можно легко добавить или обновить, кроме этого снижаются затраты на оборудование и многое другое. С помощью виртуализации можно превратить набор физических ресурсов в кластер одноразовых виртуальных машин.

Каждая виртуальная машина представляет собой полноценную машину, на которой выполняются все компоненты, включая собственную операционную систему, поверх виртуализированного оборудования.

Эра контейнеров: Контейнеры похожи на виртуальные машины, но у них есть свойства изоляции для совместного использования операционной системы (ОС) между приложениями. Поэтому контейнеры считаются легкими. Подобно виртуальной машине, контейнер имеет свою собственную файловую систему, процессор, память, пространство процесса и многое другое. Поскольку они не связаны с базовой инфраструктурой, они переносимы между облаками и дистрибутивами ОС.

Контейнеры стали популярными из-за таких дополнительных преимуществ как:

- Гибкое создание и развертывание приложений: простота и эффективность создания образа контейнера по сравнению с использованием образа виртуальной машины.
- Непрерывная разработка, интеграция и развертывание: обеспечивает надежную и частую сборку и развертывание образа контейнера с быстрым и простым откатом (благодаря неизменности образа).
- Разделение задач между Dev и Ops: создавайте образы контейнеров приложений во время сборки/релиза, а не во время развертывания, тем самым отделяя приложения от инфраструктуры.
- Наблюдаемость охватывает не только информацию и метрики на уровне ОС, но также информацию о работоспособности приложений и другие сигналы.
- Идентичная окружающая среда при разработке, тестировании и релизе: на ноутбуке работает так же, как и в облаке.
- Переносимость облачных и операционных систем: работает на Ubuntu, RHEL, CoreOS, on-prem, Google Kubernetes Engine и в любом другом месте.

- Управление, ориентированное на приложения: повышает уровень абстракции от запуска ОС на виртуальном оборудовании до запуска приложения в ОС с использованием логических ресурсов.
- Слабосвязанные, распределенные, гибкие, выделенные микросервисы: вместо монолитного стека на одной большой выделенной машине, приложения разбиты на более мелкие независимые части, которые можно динамически развертывать и управлять.
- Изоляция ресурсов: предсказуемая производительность приложения.
- Грамотное использование ресурсов: высокая эффективность и компактность.

Зачем вам Kubernetes и что он может сделать?

Контейнеры — отличный способ связать и запустить ваши приложения. В производственной среде необходимо управлять контейнерами, которые запускают приложения, и гарантировать отсутствие простоев. Например, если контейнер выходит из строя, необходимо запустить другой контейнер. Не было бы проще, если бы такое поведение обрабатывалось системой?

Вот тут Kubernetes приходит на помощь! Kubernetes дает вам фреймворк для гибкой работы распределенных систем. Он занимается масштабированием и обработкой ошибок в приложении, предоставляет шаблоны развертывания и многое другое. Например, Kubernetes может легко управлять канареечным развертыванием вашей системы.

Kubernetes предоставляет вам:

- **Мониторинг сервисов и распределение нагрузки** Kubernetes может обнаружить контейнер, используя имя DNS или собственный IP-адрес. Если трафик в контейнере высокий, Kubernetes может сбалансировать нагрузку и распределить сетевой трафик, чтобы развертывание было стабильным.
- **Оркестрация хранилища** Kubernetes позволяет вам автоматически смонтировать систему хранения по вашему выбору, такую как локальное хранилище, провайдеры общедоступного облака и многое другое.
- **Автоматическое развертывание и откаты** Используя Kubernetes можно описать желаемое состояние развернутых контейнеров и изменить фактическое состояние на желаемое. Например, вы можете автоматизировать Kubernetes на создание новых контейнеров для развертывания, удаления существующих контейнеров и распределения всех их ресурсов в новый контейнер.
- **Автоматическое распределение нагрузки** Вы предоставляете Kubernetes кластер узлов, который он может использовать для запуска

контейнерных задач. Вы указываете Kubernetes, сколько ЦП и памяти (ОЗУ) требуется каждому контейнеру. Kubernetes может разместить контейнеры на ваших узлах так, чтобы наиболее эффективно использовать ресурсы.

- **Самоконтроль** Kubernetes перезапускает отказавшие контейнеры, заменяет и завершает работу контейнеров, которые не проходят определенную пользователем проверку работоспособности, и не показывает их клиентам, пока они не будут готовы к обслуживанию.
- **Управление конфиденциальной информацией и конфигурацией** Kubernetes может хранить и управлять конфиденциальной информацией, такой как пароли, OAuth-токены и ключи SSH. Вы можете развертывать и обновлять конфиденциальную информацию и конфигурацию приложения без изменений образов контейнеров и не раскрывая конфиденциальную информацию в конфигурации стека.

Чем Kubernetes не является

Kubernetes — это не традиционная комплексная система PaaS (платформа как услуга). Поскольку Kubernetes работает на уровне контейнеров, а не на уровне оборудования, у него имеется определенные общепримениимые возможности, характерные для PaaS, такие как развертывание, масштабирование, балансировка нагрузки, ведение журналов и мониторинг. Тем не менее, Kubernetes это не монолитное решение, поэтому указанные возможности по умолчанию являются дополнительными и подключаемыми. У Kubernetes есть компоненты для создания платформы разработчика, но он сохраняет право выбора за пользователем и гибкость там, где это важно.

Kubernetes:

- Не ограничивает типы поддерживаемых приложений. Kubernetes стремится поддерживать широкий спектр рабочих нагрузок, включая те, у которых есть или отсутствует состояние, а также связанные с обработкой данных. Если приложение может работать в контейнере, оно должно отлично работать и в Kubernetes.
- Не развертывает исходный код и не собирает приложение. Рабочие процессы непрерывной интеграции, доставки и развертывания (CI/CD) определяются культурой и предпочтениями организации, а также техническими требованиями.
- Не предоставляет сервисы для приложения, такие как промежуточное программное обеспечение (например, очереди сообщений), платформы обработки данных (например, Spark), базы данных (например, MySQL), кеши или кластерные системы хранения (например, Ceph), как

встроенные сервисы. Такие компоненты могут работать в Kubernetes и/или могут быть доступны для приложений, работающих в Kubernetes, через переносные механизмы, такие как Open Service Broker.

- Не включает решения для ведения журнала, мониторинга или оповещения. Он обеспечивает некоторые интеграции в качестве доказательства концепции и механизмы для сбора и экспорта метрик.
- Не указывает и не требует настройки языка/системы (например, Jsonnet). Он предоставляет декларативный API, который может быть нацелен на произвольные формы декларативных спецификаций.
- Не предоставляет и не принимает никаких комплексных систем конфигурации, технического обслуживания, управления или самовосстановления.
- Кроме того, Kubernetes — это не просто система оркестрации. Фактически, Kubernetes устраняет необходимость в этом. Техническое определение оркестрации — это выполнение определенного рабочего процесса: сначала сделай A, затем B, затем C. Напротив, Kubernetes содержит набор независимых, компонуемых процессов управления, которые непрерывно переводит текущее состояние к предполагаемому состоянию. Неважно, как добраться от A до C. Не требуется также централизованный контроль. Это делает систему более простой в использовании, более мощной, надежной, устойчивой и расширяемой.

2. Команды для работы с Kubernetes

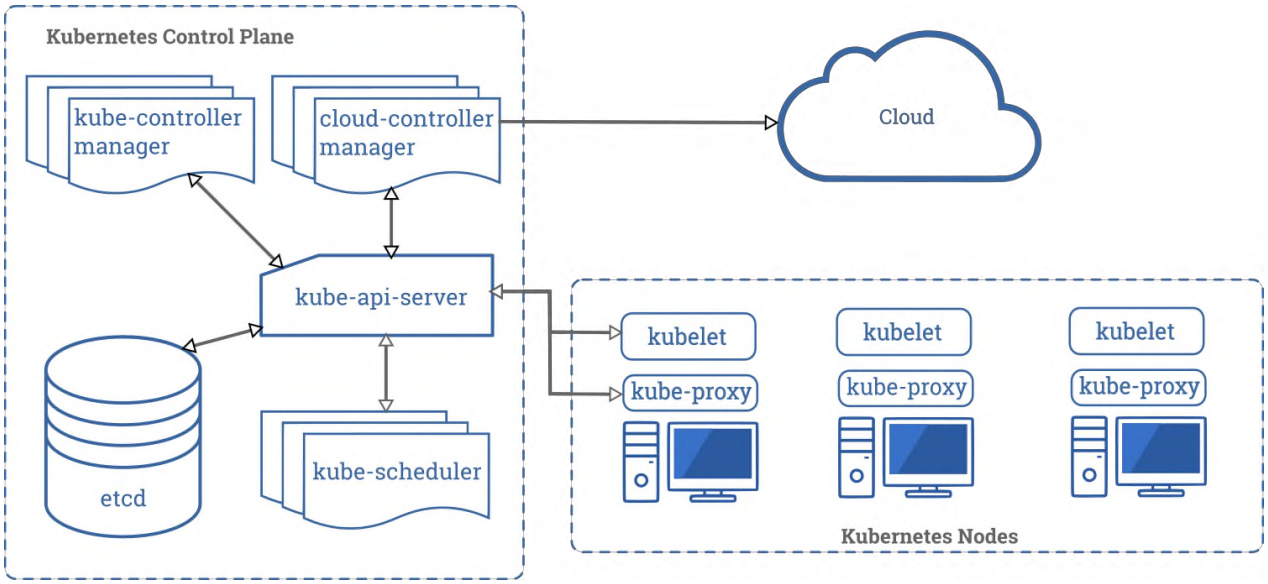
При развёртывании Kubernetes вы имеете дело с кластером.

Кластер Kubernetes cluster состоит из набора машин, так называемые узлы, которые запускают контейнеризированные приложения. Кластер имеет как минимум один рабочий узел.

В рабочих узлах размещены поды, являющиеся компонентами приложения. Плоскость управления управляет рабочими узлами и подами в кластере. В промышленных средах плоскость управления обычно запускается на нескольких компьютерах, а кластер, как правило, развёртывается на нескольких узлах, гарантируя отказоустойчивость и высокую надёжность.

На этой странице в общих чертах описывается различные компоненты, необходимые для работы кластера Kubernetes.

Ниже показана диаграмма кластера Kubernetes со всеми связанными компонентами.



Плоскость управления компонентами

Компоненты панели управления отвечают за основные операции кластера (например, планирование), а также обрабатывают события кластера (например, запускают новый под, когда поле replicas развертывания не соответствует требуемому количеству реплик).

Компоненты панели управления могут быть запущены на любой машине в кластере. Однако для простоты сценарии настройки обычно запускают все компоненты панели управления на одном компьютере и в то же время не позволяют запускать пользовательские контейнеры на этом компьютере. Смотрите страницу Создание высоконадёжных кластеров для примера настройки нескольких ведущих виртуальных машин.

kube-apiserver

Сервер API — компонент Kubernetes панели управления, который представляет API Kubernetes. API-сервер — это клиентская часть панели управления Kubernetes

Основной реализацией API-сервера Kubernetes является kube-apiserver. kube-apiserver предназначен для горизонтального масштабирования, то есть развёртывание на несколько экземпляров. Вы можете запустить несколько экземпляров kube-apiserver и сбалансировать трафик между этими экземплярами.

etcd

Распределённое и высоконадёжное хранилище данных в формате "ключ-значение", которое используется как основное хранилище всех данных кластера в Kubernetes.

Если ваш кластер Kubernetes использует etcd в качестве основного хранилища, убедитесь, что у вас настроено резервное копирование данных.

Вы можете найти подробную информацию о etcd в официальной документации.

kube-scheduler

Компонент плоскости управления, который отслеживает созданные поды без привязанного узла и выбирает узел, на котором они должны работать.

При планировании развёртывания подов на узлах учитываются множество факторов, включая требования к ресурсам, ограничения, связанные с аппаратными/программными политиками, принадлежности (affinity) и непринадлежности (anti-affinity) узлов/подов, местонахождения данных, предельных сроков.

kube-controller-manager

Компонент Control Plane запускает процессы контроллера.

Вполне логично, что каждый контроллер в свою очередь представляет собой отдельный процесс, и для упрощения все такие процессы скомпилированы в один двоичный файл и выполняются в одном процессе.

Эти контроллеры включают:

- Контроллер узла (Node Controller): уведомляет и реагирует на сбои узла.
- Контроллер репликации (Replication Controller): поддерживает правильное количество подов для каждого объекта контроллера репликации в системе.
- Контроллер конечных точек (Endpoints Controller): заполняет объект конечных точек (Endpoints), то есть связывает сервисы (Services) и поды (Pods).

• Контроллеры учетных записей и токенов (Account & Token Controllers): создают стандартные учетные записи и токены доступа API для новых пространств имен.

cloud-controller-manager

cloud-controller-manager запускает контроллеры, которые взаимодействуют с основными облачными провайдерами. Двоичный файл cloud-controller-manager — это альфа-функциональность, появившиеся в Kubernetes 1.6.

cloud-controller-manager запускает только циклы контроллера, относящиеся к облачному провайдеру. Вам нужно отключить эти циклы контроллера в kube-controller-manager. Вы можете отключить циклы контроллера, установив флаг --cloud-provider со значением external при запуске kube-controller-manager.

С помощью cloud-controller-manager код как облачных провайдеров, так и самого Kubernetes может разрабатываться независимо друг от друга. В предыдущих версиях код ядра Kubernetes зависел от кода, предназначенного для функциональности облачных провайдеров. В будущих выпусках код, специфичный для облачных провайдеров, должен поддерживаться самим облачным провайдером и компоноваться с cloud-controller-manager во время запуска Kubernetes.

Следующие контроллеры зависят от облачных провайдеров:

• Контроллер узла (Node Controller): проверяет облачный провайдер, чтобы определить, был ли удален узел в облаке после того, как он перестал работать

• Контроллер маршрутов (Route Controller): настраивает маршруты в основной инфраструктуре облака

• Контроллер сервисов (Service Controller): создаёт, обновляет и удаляет балансировщики нагрузки облачного провайдера.

• Контроллер тома (Volume Controller): создаёт, присоединяет и монтирует тома, а также взаимодействует с облачным провайдером для оркестрации томов.

Компоненты узла

Компоненты узла работают на каждом узле, поддерживая работу подов и среды выполнения Kubernetes.

kubelet

Агент, работающий на каждом узле в кластере. Он следит за тем, чтобы контейнеры были запущены в поде.

Утилита kubelet принимает набор PodSpecs, и гарантирует работоспособность и исправность определённых в них контейнеров. Агент kubelet не отвечает за контейнеры, не созданные Kubernetes.

kube-proxy

kube-proxy — сетевой прокси, работающий на каждом узле в кластере, и реализующий часть концепции сервис.

kube-proxy конфигурирует правила сети на узлах. При помощи них разрешаются сетевые подключения к вашими подам изнутри и снаружи кластера.

kube-proxy использует уровень фильтрации пакетов в операционной системы, если он доступен. В противном случае, kube-proxy сам обрабатывает передачу сетевого трафика.

Среда выполнения контейнера

Среда выполнения контейнера — это программа, предназначенная для выполнения контейнеров.

Kubernetes поддерживает несколько сред для запуска контейнеров: Docker, containerd, CRI-O, и любая реализация Kubernetes CRI (Container Runtime Interface).

Дополнения

Дополнения используют ресурсы Kubernetes (DaemonSet, Deployment и т.д.) для расширения функциональности кластера. Поскольку дополнения охватывают весь кластер, ресурсы относятся к пространству имен kube-system.

Некоторые из дополнений описаны ниже; более подробный список доступных расширений вы можете найти на странице Дополнения.

DNS

Хотя прочие дополнения не являются строго обязательными, однако при этом у всех Kubernetes-кластеров должен быть кластерный DNS, так как многие примеры предполагают его наличие.

Кластерный DNS — это DNS-сервер наряду с другими DNS-серверами в вашем окружении, который обновляет DNS-записи для сервисов Kubernetes.

Контейнеры, запущенные посредством Kubernetes, автоматически включают этот DNS-сервер в свои DNS.

Веб-интерфейс (Dashboard)

Dashboard — это универсальный веб-интерфейс для кластеров Kubernetes. С помощью этой панели, пользователи могут управлять и устранять неполадки кластера и приложений, работающих в кластере.

Мониторинг ресурсов контейнера

Мониторинг ресурсов контейнера записывает общие метрики о контейнерах в виде временных рядов в центральной базе данных и предлагает пользовательский интерфейс для просмотра этих данных.

Логирование кластера

Механизм логирования кластера отвечает за сохранение логов контейнера в централизованном хранилище логов с возможностью их поиска/просмотра.

3. Развертывание кластера на собственных серверах

Идея самостоятельно развернуть кластер Kubernetes на собственных серверах или в облаке выглядит привлекательной: кажется, что это дешевле, чем платить за Managed-решение от провайдера. На самом деле все не так однозначно: на практике можно обнаружить скрытые расходы и подводные камни.

При этом для крупных компаний Self-Hosted может быть вариантом, так как у них есть условно бесплатные ресурсы и штат специалистов для поддержки технологии, а иногда еще горячее желание построить и развивать свою платформу во что бы то ни стало. А вот с малым и средним бизнесом ситуация немного другая, решение нужно взвесить со всех сторон.

Для старта понадобятся время, деньги и администраторы, разбирающиеся в Kubernetes.

Первая статья расходов — на специалистов, которые умеют работать с этой системой и смогут обслуживать кластер. Это дорогие ребята, на рынке их немного, и нанять трудно.

Почему Kubernetes сильно увеличивает расходы на специалистов? Вроде бы развернуть кластер несложно, для этого есть официальная документация и инсталляторы, например, Kubespray или Kubeadm. Однако если в компании есть инженер, который может прочитать строчку документации и разобраться, как поставить Kubernetes на серверы с помощью одной команды, это еще не все, этим его работа не ограничится.

В реальности развернуть кластер только половина дела. В таком виде он будет работать до первой проблемы, которая неизбежно возникнет через неделю или месяц. Например, перестанут создаваться поды из-за неверной конфигурации ресурсов на controller-manager. Или кластер начнет работать нестабильно из-за проблем с дисками у etcd. Или запущенные CronJob из-за ошибок controller-manager начнут бесконечно плодить новые поды. Или в кластере будут возникать сетевые ошибки из-за неправильного выбора конфигурации DNS.

В общем, проблем может быть много, поэтому нужен отдельный человек, знающий, как развернуть кластер, как дебажить, как запускать приложения в производственной среде.

Кроме того, вместе с Kubernetes в компании появляются новые потребности, например, мониторинг для выявления ошибок, система хранения данных, сбор логов. Кластер нужно развивать, чтобы получить от

технологии ожидаемый профит. Это требует времени, поэтому даже опытному администратору не получится выделить неделю для настройки кластера и какие-то часы для администрирования.

Скорее всего, понадобится человек на фултайм, который будет заниматься только Kubernetes, поддержкой и развитием кластера. В большой компании может родиться отдел для поддержки инфраструктуры.

Конечно, если запускать Kubernetes только ради деплоя контейнеров, то можно не разбираться и не развивать кластер. Но тогда возникает вопрос: зачем вам Kubernetes? Можно взять более простой в настройке и поддержке инструмент, тот же Docker Swarm. Если вы хотите от Kubernetes что-то простое, просто его не используйте. Нет смысла тратить время на развертывание кластера лишь ради запуска простого кода. Эта технология предназначена для проектов, где постоянно идет разработка, часто запускаются новые релизы и нужно выдерживать требования HighLoad.

По этой причине Self-Hosted Kubernetes в большинстве случаев могут успешно запустить только крупные компании, где есть возможность выделить сотрудников для обслуживания кластера и нет потребности экономить ресурсы.

Кроме того, самостоятельное развертывание кластера — дело небыстрое. Если понадобится запустить кластер в короткие сроки для проекта или тестовых сред, то на Self-Hosted это не выйдет: развертывание займет несколько часов, а то и недель. К этому стоит быть готовыми. Для сравнения: в облаке вы запустите кластер KaaS за 10 минут и сможете сразу его использовать, но это получается потому, что над инфраструктурной частью уже заранее поработали специалисты провайдера.

Kubernetes требует прокачки: он не работает сам по себе

Kubernetes — отдельная экосистема, которой нужно заниматься и подключать к ней дополнительные инструменты. Если брать Self-Hosted, то все это придется делать самостоятельно.

Все инструменты, дополняющие Kubernetes, — Open Source-решения, которые нужно настраивать. В кластер потребуется установить систему мониторинга, реализовать балансировку нагрузки, сбор и хранение логов, настройки безопасности и авторизации пользователей, сети и многое другое.

Например, понадобится мониторить и сам кластер, и приложения в нем. Причем стандартного мониторинга через Zabbix вам не хватит, потребуется специфический — Prometheus или Telegraph.

С логами аналогичная ситуация: из коробки вы получите только историю логов для уже запущенных приложений, при передеплое она исчезнет. Вручную собирать логи с Kubernetes не получится, нужно подключать сборщики логов вроде Fluentd и систему хранения, например Elasticsearch или Loki. Отдельно придется заниматься балансировкой нагрузки: понадобится отказоустойчивый балансер вроде MetalLB.

Системы хранения для Self-Hosted Kubernetes — еще одна головная боль

Kubernetes изначально разработан для Stateless-приложений — они ничего не хранят внутри контейнеров. При работе со Stateful-приложениями, хранящими данные, встает вопрос подключения внешних хранилищ.

Самый простой вариант, к которому часто прибегают, — поднять один NFS-сервер, но это решение для бедных: оно не обеспечит высокую доступность и сохранность данных. Если в медленный и ненадежный NFS будут ходить продакшен-сервисы с важными данными, могут возникнуть большие проблемы.

Для нормальной работы приложения без изменения его логики понадобятся Persistent Volumes — хранилища, связанные с подами. Они подключаются внутрь контейнеров как локальные директории, позволяя приложению хранить данные «под собой». Среди рабочих вариантов — CephFS, Glusterfs, FC (Fiber Channel), полный список СХД можно посмотреть в официальной документации.

Интеграция Kubernetes с Persistent Volumes — нетривиальная задача. Чтобы развернуть тот же Ceph, недостаточно взять мануал с Хабра и выполнить ряд команд. Плюс в дальнейшем СХД должен кто-то заниматься — опять нужен отдельный инженер, а то и несколько.

Если же Self-Hosted-кластер развернут не на железе, а на виртуальных машинах в облаке, то все немного проще — собственный кластер Ceph поднимать не нужно. Можно взять кластер хранилища у провайдера и научить его работать с кластером K8s, если провайдер готов предоставить вам API к своей системе хранения данных, что есть не везде. Писать интеграцию при этом придется самостоятельно.

Правда, у провайдеров, предоставляющих IaaS, можно арендовать объектное хранилище или облачную СУБД, но только если логика приложения позволяет их использовать. А в Managed-решениях Kubernetes уже «из коробки» есть интегрированные Persistent Volumes.

Отказоустойчивость кластера — отдельная проблема

С Kubernetes проще обеспечить отказоустойчивость приложений, однако потребуется еще и реализовать отказоустойчивость кластера.

В Kubernetes есть мастер-нода, непосредственно управляющая кластером и содержащая его конфигурацию, метаданные и статусы объектов Kubernetes. Отказоустойчивый кластер включает три мастер-ноды, отдельные от самого кластера и дублирующие друг друга. Каждая нода — отдельный сервер или виртуальная машина, их не могут использовать бизнес-приложения. То есть их нужно отдельно подключать и обслуживать либо оплачивать аренду в облаке.

Это создает сложности для малого бизнеса: раньше для всех приложений требовалось всего два сервера, а с Kubernetes только ради отказоустойчивости нужно три дополнительных сервера.

Также в кластере Kubernetes есть прекрасная фича — встроенный механизм самовосстановления. Если одна из нод выходит из строя, то все процессы, ранее работающие на этой ноде, автоматически перезапускаются на других нодах кластера. Вот только чтобы это произошло, на остальных нодах нужен резерв по ресурсам. И его нельзя ничем занимать, иначе приложения не смогут переехать в случае проблем.

Резерв зависит от того, какое количество вышедших из строя нод вероятно в вашем случае:

Если у вас одна стойка с серверами в одном дата-центре, то одномоментно, скорее всего, выйдет из строя максимум одна нода на одном сервере, например, из-за ошибок ОС. Значит, нужен резерв на одну ноду. Конечно, может сломаться стойка, но тут уже нужно резервирование не средствами Kubernetes.

Если у вас несколько стоек с серверами, то есть вероятность потери одной стойки, например, из-за проблем со свичем, когда все серверы в ней станут недоступны. Значит, нужен резерв в размере количества серверов в одной стойке.

Если у вас несколько дата-центров, то в каждом нужно держать резерв по размеру другого дата-центра, чтобы приложения работали в случае его выхода из строя.

Если проще, то это выглядит так: когда в кластере 10 нод и вы хотите без проблем пережить потерю одной ноды, то вам потребуется 10-процентный

запас ресурсов. Если же приложения должны работать даже при потере 50% кластера, значит, на всех нодах нужен запас в 50%.

При этом лучше, если ноды в кластере небольшие по объему, но их много. Допустим, у вас есть пул ресурсов — 100 ГБ оперативной памяти и 100 ядер CPU. Такой объем позволяет запустить 10 виртуалок и 10 нод кластера Kubernetes. И в случае выхода из строя одной ноды вы теряете только 10% кластера.

На железных серверах такую конфигурацию не создашь. Например, используя 300 ГБ оперативной памяти и 50 ядер CPU, вы развернете всего 2–3 ноды кластера. И в случае выхода из строя одной ноды рискуете сразу потерять 30–50% кластера.

Получается, что риск того, что кластер «ляжет» вследствие сбоя или непредсказуемой нагрузки, на традиционной инфраструктуре выше. Кроме того, может быть так: специалисты без достаточного опыта не всегда могут заранее предусмотреть проблемы, понять, в чем их причина, и быстро устранить.

4. Сетевые решения CNI

Предыстория CNI

Проект Container Network Interface (CNI) зародился в компании **CoreOS**, известной по движку для контейнеров rkt (недавно был тоже передан в CNCF, одновременно с containerd), NoSQL-хранилищу etcd, активной работе над Kubernetes и другими Open Source-проектами, связанными с контейнерами и DevOps. Первая подробная демонстрация CNI состоялась ещё в конце 2015 года (см. видео в октябре, презентацию в ноябре). С самого начала проект позиционировался в качестве «предлагаемого стандарта для конфигурации сетевых интерфейсов для Linux-контейнеров», а использовать его первым делом начали, конечно же, в rkt. Первыми «сторонними пользователями» CNI стали Project Calico, Weaveworks, а в скором времени к ним примкнул Kubernetes.

Адаптация CNI в платформе Kubernetes стоит отдельного внимания. Поскольку потребность в стандартизации сетевой конфигурации Linux-контейнеров была очевидной и всё более актуальной на тот момент *(~2015 год)*, CNI оказался не единственным таким проектом. Конкурирующий продукт — **Container Network Model (CNM)** от Docker, представленный в том же 2015-м, — решал те же задачи и получил свою эталонную реализацию в виде libnetwork — библиотеки, которая выросла из сетевого кода в libcontainer и Docker Engine. И важным этапом в противостоянии CNI и CNM стал выбор, сделанный Kubernetes. В статье «Почему Kubernetes не использует libnetwork?» *(январь 2016 года)* разработчики подробно описывают все технические и иные причины, объясняющие их решение. Основная же суть (помимо ряда технических моментов) сводится к следующему:

CNI ближе к Kubernetes с философской точки зрения. Он гораздо проще CNM, не требует демонов и его кроссплатформенность по меньшей правдоподобна (исполняемая среда контейнеров CoreOS rkt поддерживает его) *[… а Kubernetes стремится поддерживать разные реализации контейнеров — **прим. перев.**]*. Быть кроссплатформенным означает возможность использования сетевых конфигураций, которые будут одинаково работать в разных исполняемых средах (т.е. Docker, Rocket, Hyper). Этот подход следует философии UNIX делать одну вещь хорошо.

По всей видимости, именно это (не только выбор Kubernetes, но и сторонний взгляд технических специалистов на существующие реализации) предопределило будущее CNI и его принятие в CNCF. Для сравнения, другие конкурирующие продукты CoreOS и Docker: среды исполнения контейнеров

rkt и conatinerd — были приняты в фонд вместе и одновременно, а вот с CNI/CNM этого не произошло.

Контейнерный сетевой интерфейс (CNI) — это платформа для динамической настройки сетевых ресурсов. Он использует группу библиотек и спецификаций, написанных на Go. Спецификация подключаемого модуля определяет интерфейс для настройки сети, предоставления IP-адресов и поддержания связи с несколькими хостами.

При использовании с Kubernetes CNI может легко интегрироваться с kubelet, позволяя использовать наложенную или базовую сеть для автоматической настройки сети между модулями. Оверлейные сети инкапсулируют сетевой трафик с помощью виртуального интерфейса, такого как виртуальная расширяемая локальная сеть (VXLAN). Базовые сети работают на физическом уровне и состоят из коммутаторов и маршрутизаторов.

После того как вы указали тип конфигурации сети, среда выполнения контейнера определяет сеть, к которой присоединяются контейнеры. Среда выполнения добавляет интерфейс в пространство имен контейнера посредством вызова подключаемого модуля CNI и выделяет маршруты подключенной подсети посредством вызовов подключаемого модуля управления IP-адресами (IPAM).

CNI поддерживает сеть Kubernetes , а также может использоваться с другими платформами оркестрации контейнеров на основе Kubernetes, такими как OpenShift. CNI использует программно-определяемую сеть (SDN) для унификации взаимодействия контейнеров во всех кластерах.

Kubernetes — это платформа оркестрации контейнеров с открытым исходным кодом, разработанная Google. Он используется для управления и автоматизации развертываний контейнеров приложений на нескольких кластерах компьютеров. Kubernetes позволяет управлять, планировать, отслеживать и поддерживать контейнерные рабочие нагрузки.

Kubernetes также можно использовать для работы в сети, что позволяет администраторам перемещать рабочие нагрузки между различными облачными инфраструктурами, включая общедоступные, частные и гибридные облака. Kubernetes позволяет разработчикам быстро упаковывать и развертывать приложения, используя предпочтительную инфраструктуру, что полезно для разработки новых версий.

Благодаря сети Kubernetes компоненты Kubernetes могут взаимодействовать с различными приложениями и друг с другом. Kubernetes отличается от других сетевых платформ тем, что имеет плоскую сетевую структуру, а это означает, что порты хоста не нужно сопоставлять с портами

контейнера. Это позволяет вам запускать распределенную систему с машинами, совместно используемыми приложениями без необходимости динамического распределения портов.

Изначально контейнеры (поды) не имеют сетевого интерфейса. Чтобы создать сетевой интерфейс для контейнера, среда выполнения контейнера отправляет команду ADD подключаемому модулю CNI (среда выполнения может вызвать подключаемый модуль с помощью таких команд, как ADD, DEL и CHECK). После создания нового сетевого интерфейса полезная нагрузка JSON передает сведения о том, что необходимо добавить в CNI.

Почему Kubernetes CNI?

Как контейнер Linux, так и сетевые технологии контейнеров продолжают развиваться, чтобы удовлетворить потребности приложений, работающих в различных средах. CNI — это инициатива Cloud-Native Computing Foundation (CNCF), которая определяет конфигурацию сетевых интерфейсов контейнеров Linux.

CNI был создан, чтобы сделать сетевые решения интегрируемыми с рядом систем оркестрации контейнеров и сред выполнения. Вместо того, чтобы делать сетевые решения подключаемыми, он определяет общий стандарт интерфейса как для сетевого уровня, так и для уровня исполнения контейнера.

CNI фокусируется на соединении сетей контейнеров и удалении выделенных ресурсов при завершении работы контейнеров. Этот фокус делает спецификации CNI простыми и позволяет им широко применяться. Проект CNI GitHub предоставляет дополнительную информацию о спецификациях CNI, в том числе о сторонних подключаемых модулях и средах выполнения, которые его используют.

Необходимость в различных сетевых плагинах Kubernetes

CNI имеет множество поддерживаемых плагинов, которые реализованы в основных фреймворках оркестрации контейнеров, таких как Kubernetes. Плагины обращаются к различным сетевым функциям контейнеров и должны соответствовать стандартам CNI, определенным спецификацией CNI.

CNI предлагает спецификации для нескольких подключаемых модулей, поскольку сеть сложна, а потребности пользователей могут различаться. Очень важно выбрать правильные плагины для вашего проекта и варианта использования.

Какие сетевые модели используются в CNI?

Сети CNI могут быть реализованы с использованием модели инкапсулированной или неинкапсулированной сети. XLAN является примером инкапсулированной модели, а протокол пограничного шлюза (BGP) — примером неинкапсулированной модели.

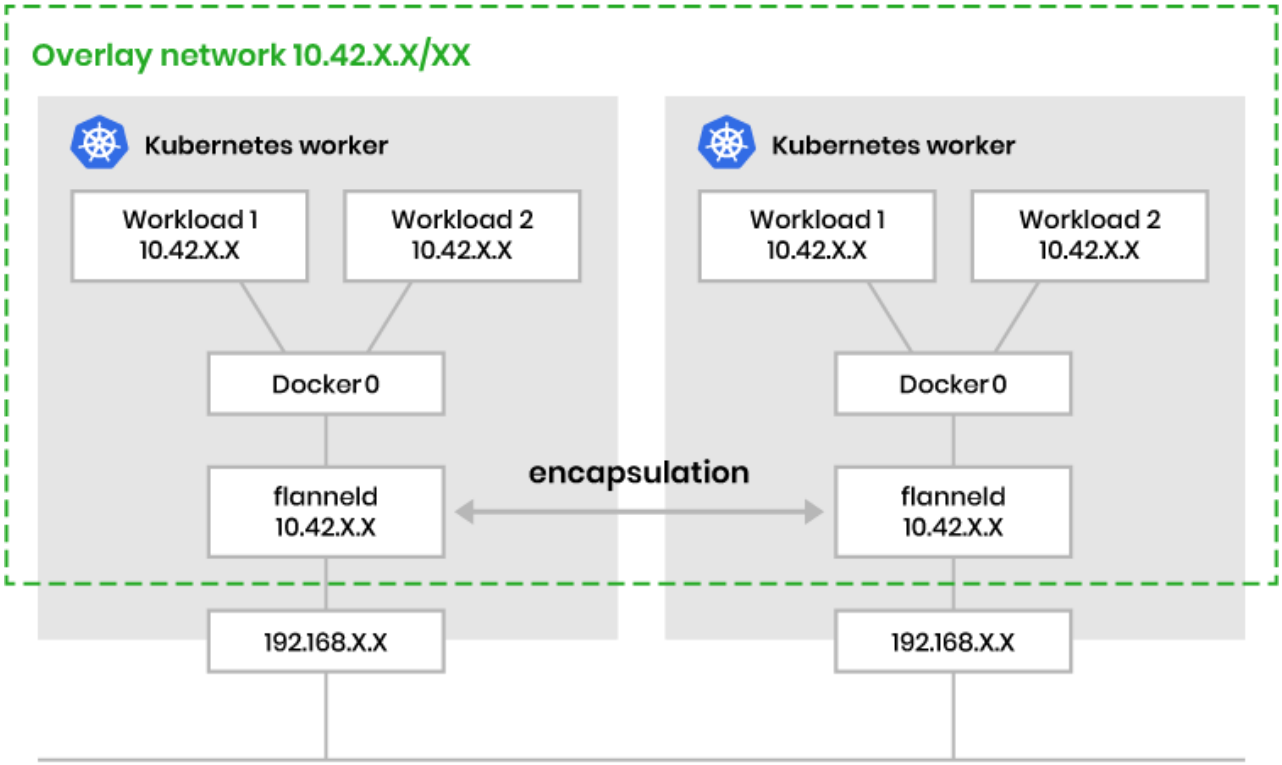
Инкапсулированные сети

Эта модель инкапсулирует логическую сеть уровня 2 поверх существующей топологии сети уровня 3, которая охватывает несколько узлов Kubernetes. Сеть уровня 2 изолирована, поэтому нет необходимости в распределении маршрутизации. Накладные расходы минимальны, при этом обеспечивается улучшенная обработка и более крупные IP-пакеты — инкапсуляция с наложением генерирует IP-заголовок, который предоставляет IP-пакет.

Порты UDP распределяют данные инкапсуляции между рабочими процессами в Kubernetes, транслируя информацию из плоскости управления сетью для достижения MAC-адресов. Примеры распространенных моделей сети с инкапсуляцией включают VXLAN и безопасность интернет-протокола (IPsec).

Проще говоря, эта модель обеспечивает мост, который соединяет рабочие процессы и модули Kubernetes. В модулях элементом, управляющим коммуникацией, является Docker или другой контейнерный движок. Он применяется в случаях, когда предпочтение отдается мосту уровня 2, поскольку он чувствителен к задержкам рабочих процессов Kubernetes на уровне 3. Для центров обработки данных, расположенных в разных географических точках, важно минимизировать задержки между ними, чтобы предотвратить сегментацию сети.

Примеры сетевых провайдеров CNI, которые следуют этой сетевой модели, включают Canal, Flannel и Weave.

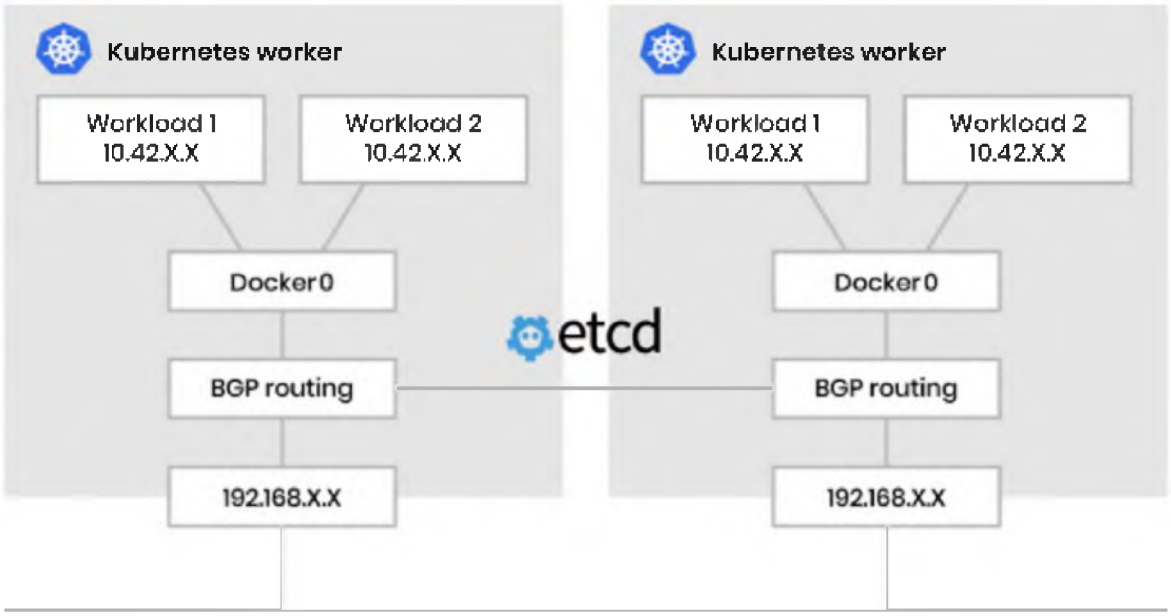


Неинкапсулированные сети

Эта модель обеспечивает сеть уровня 3 для маршрутизации пакетов между контейнерами. Нет изолированной сети уровня 2 или служебных данных, но это происходит за счет рабочих процессов Kubernetes, которые должны управлять любым необходимым распределением маршрутов. Сетевой протокол реализован для подключения рабочих процессов Kubernetes и использования BGP для распространения информации о маршрутизации среди модулей. В модулях компонентом, управляющим связью с рабочими нагрузками, является Docker или другой контейнерный движок.

Эта модель предполагает расширение сетевого маршрутизатора между рабочими процессами Kubernetes — этот маршрутизатор предоставляет информацию о том, как получить доступ к модулям. Неинкапсулированные сети подходят для вариантов использования, в которых предпочтение отдается маршрутизируемой сети уровня 3. Маршруты для рабочих процессов Kubernetes динамически обновляются на уровне операционной системы, что снижает задержку.

Примеры поставщиков, использующих неинкапсулированную сетевую модель, включают Romana и Calico.


Kubernetes worker
Workload 1
10.42.X.X
Workload 2
10.42.X.X
Docker0
BGP routing
192.168.X.X
etcd
Kubernetes worker
Workload 1
10.42.X.X
Workload 2
10.42.X.X
Docker0
BGP routing
192.168.X.X

5. Service mesh. Знакомство с Istio и Envoy

Service Mesh – это конфигурируемый инфраструктурный уровень с низкой задержкой, который нужен для обработки большого объема сетевых межпроцессных коммуникаций между программными интерфейсами приложения (API). Service Mesh обеспечивает быструю, надёжную и безопасную коммуникацию между контейнеризированными и часто эфемерными сервисами инфраструктуры приложений. Service Mesh предоставляет такие возможности, как обнаружение сервисов, балансировку нагрузки, шифрование, прозрачность, трассируемость, аутентификацию и авторизацию, а также поддержку шаблона автоматического выключения (*circuit breaker*).

Service Mesh обычно реализуется путем предоставления каждому экземпляру сервиса экземпляра прокси, который называется *Sidecar. Sidecar* обрабатывают коммуникации между сервисами, производят мониторинг и устраняют проблемы безопасности, то есть все, что может быть абстрагировано от отдельных сервисов. Таким образом, разработчики могут писать, поддерживать и обслуживать код приложения в сервисах, а системные администраторы могут работать с Service Mesh и запускать приложение.

Istio от Google, IBM и Lyft, на данный момент является самый известной Service Mesh-архитектурой. А Kubernetes, который изначально разрабатывался в Google, сейчас является единственным фреймворком для оркестрации контейнеров, который поддерживается Istio. Вендоры пытаются создать коммерческие поддерживаемые версии Istio. Интересно, что нового они смогут привнести в проект с открытым исходным кодом. Однако Istio – это не единственный вариант, поскольку разрабатываются и другие реализации Service Mesh. Паттерн sidecar proxy является наиболее популярной реализацией, как можно судить по проектам Buoyant, HashiCorp, Solo.io и другим. Существуют и альтернативные архитектуры: технологический инструментарий Netflix – это один из подходов, где функционал Service Mesh реализуется за счет библиотек Ribbon, Hysterix, Eureka, Archaius, а также таких платформ, как Azure Service Fabric.

Service Mesh также имеет свою собственную терминологию для компонентов-сервисов и функций:

- **Фреймворк оркестрации контейнеров**. По мере того, как все больше и больше контейнеров добавляется в инфраструктуру приложения, появляется необходимость в отдельном инструменте для мониторинга и управления контейнерами – фреймворке оркестрации контейнеров.

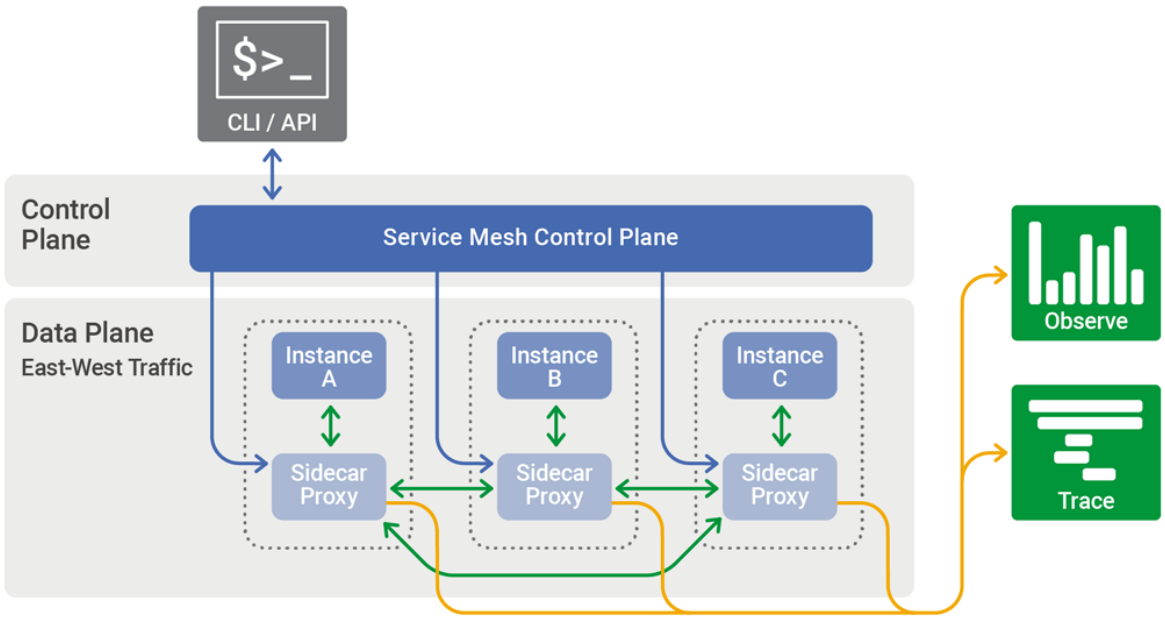
Kubernetes плотно занял эту нишу, причем настолько, что даже его основные конкуренты Docker Swarm и Mesosphere DC/OS предлагают в качестве альтернативы интеграцию с Kubernetes.

- **Сервисы и экземпляры (поды Kubernetes)**. Экземпляр – это единственная запущенная копия микросервиса. Иногда один экземпляр – это один контейнер. В Kubernetes экземпляр состоит из небольшой группы независимых контейнеров, который называется подом. Клиенты редко обращаются напрямую к экземпляру или поду, чаще они обращаются к сервису, который представляет из себя набор идентичных масштабируемых и отказоустойчивых экземпляров или подов (реплик).
- **Sidecar Proxy**. Sidecar Proxy работает с одним экземпляром или подом. Смысл Sidecar Proxy в том, чтобы направлять или проксировать трафик, приходящий от контейнера, с которым он работает, и обратный трафик. Sidecar взаимодействует с другими Sidecar Proxy и управляется фреймворком оркестрации. Многие реализации Service Mesh используют Sidecar Proxy для перехвата и управления всем входящим и исходящим трафиком экземпляра или пода.
- **Обнаружение сервисов**. Когда экземпляру нужно взаимодействовать с другим сервисом, ему нужно найти (обнаружить) исправный и доступный экземпляр другого сервиса. Как правило экземпляр выполняет поиск по DNS. Фреймворк оркестрации контейнеров хранит список экземпляров, которые готовы к получению запросов, и предоставляет интерфейс для DNS-запросов.
- **Балансировка нагрузки**. Большинство фреймворков оркестрации контейнеров обеспечивают балансировку нагрузки на 4 уровне (транспортном). Service Mesh реализует более сложную балансировку нагрузки на 7 уровне (прикладном), богатую алгоритмами и более эффективную в вопросе управления трафиком. Параметры балансировки нагрузки могут быть изменены с помощью API, что позволяет оркестрировать сине-зеленое или канареечное развертывание.
- **Шифрование**. Service Mesh может зашифровывать и расшифровывать запросы и ответы, снимая это бремя с сервисов. Service Mesh также может повысить производительность за счет приоритезации или переиспользования существующих постоянных соединений, что снижает необходимость в дорогих вычислениях для создания новых соединений. Наиболее распространенной реализацией шифрования трафика является *mutual TLS (mTLS)*, где инфраструктура открытых ключей (PKI)

генерирует и распространяет сертификаты и ключи для использования их в Sidecar Proxy.

- **Аутентификация и авторизация**. Service Mesh может авторизовывать и аутентифицировать запросы сделанные снаружи или изнутри приложения, отправляя экземплярам только валидированные запросы.
- **Поддержка шаблона автоматического выключения**. Service Mesh поддерживает шаблон автоматического выключения, который изолирует нездоровые экземпляры, а затем постепенно возвращает их в пул здоровых экземпляров при необходимости.

Та часть приложения Service Mesh, которая управляет сетевым трафиком между экземплярами называется *Data Plane*. Создание и развертывание конфигурации, которая управляет поведением *Data Plane*, выполняется с помощью отдельной *Control Plane*. *Control Plane* обычно включает в себя или спроектирована таким образом, чтобы подключаться к API, CLI или GUI для управления приложением.



Control Plane в Service Mesh распределяет конфигурацию между Sidecar Proxy и Data Plane.

Часто архитектура Service Mesh применяется для решения сложных операционных задач с использованием контейнеров и микросервисов. Пионерами в области микросервисов являются такие компании как Lyft,

Netflix и Twitter, которые предоставляют стабильно работающие сервисы миллионам пользователей по всему миру. (Здесь вы можете познакомиться с подробным описанием некоторых архитектурных задач, с которыми столкнулся Netflix). Для менее требовательных прикладных задач скорее всего будет достаточно более простых архитектур.

Архитектура Service Mesh вряд ли когда-нибудь станет ответом на все вопросы, связанные с работой приложений и их доставкой. Архитекторы и разработчики обладают огромным арсеналом инструментов, и только один из них — молоток, который среди множества задач должен решать лишь одну – забивать гвозди. Microservices Reference Architecture от NGINX, например, включает в себя несколько различных моделей, которые дают непрерывный спектр подходов к решению задач с помощью микросервисов. Элементы, которые объединяются в архитектуре Service Mesh, такие как NGINX, контейнеры, Kubernetes и микросервисы в качестве архитектурного подхода, могут быть не менее продуктивно использованы и в реализациях без Service Mesh. Например, Istio был разработан как полноценная архитектура Service Mesh, но модульность говорит о том, что разработчики могут выбирать и применять только необходимые им компоненты технологий. Помня об этом, необходимо сформировать четкое понимание концепции Service Mesh, даже если вы не уверены в том, что сможете когда-нибудь полноценно ее реализовать в приложении.

Istio

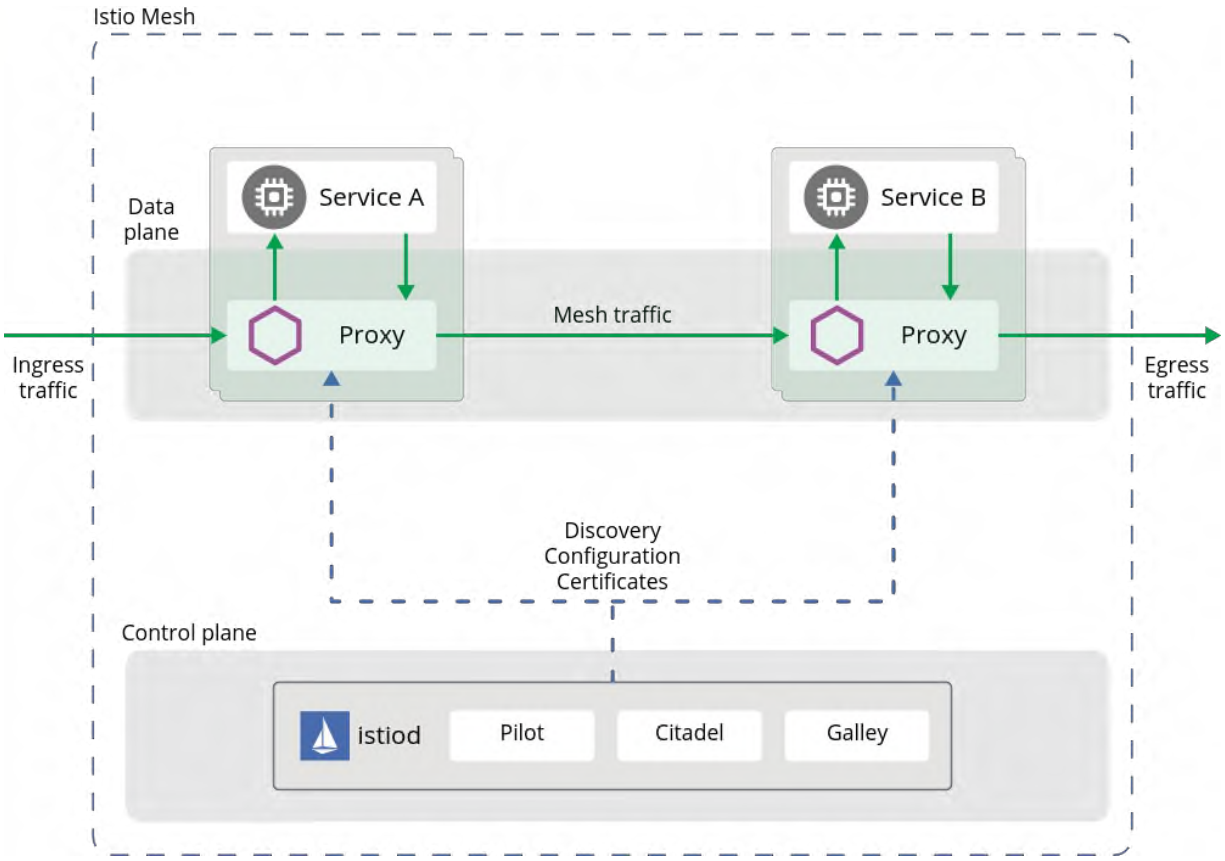
Istio- одна из реализации концепии Service Mesh, позволяющая реализовать Service Discovery, Load Balancing, контроль над трафиком, canary rollouts и blue-green deployments, мониторить трафик между приложениями.

Архитектура Istio

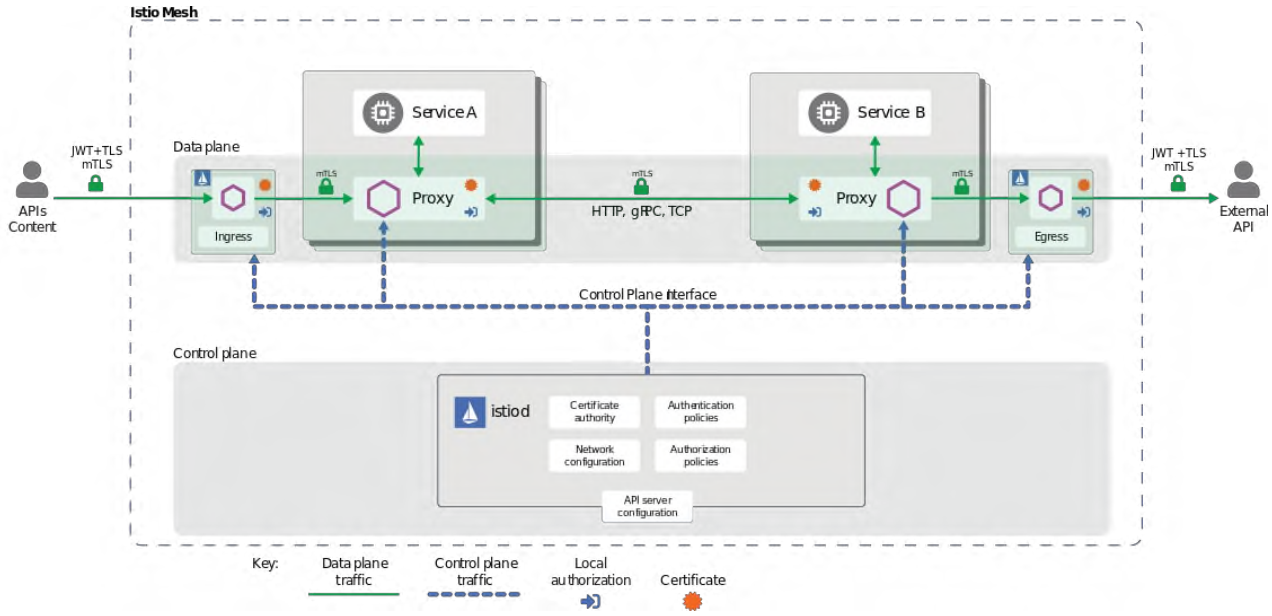
Итак, Istio как service mesh состоит из двух основных частей – *Data plane* и *Control plane*:

- **Data plane** (“уровень данных”): состоит из набора прокси-сервисов, sidecar-контейнеров в каждом поде, для которых использует расширенную версию прокси-сервера Envoy. Эти сайдакры связывают и контролируют трафик между приложениями, собирают и отправляют метрики
- **Control plane** (“уровень управления”): управляет и настраивает сайдкары, управляет метриками, TLS-сертификатами

В целом схема Istio выглядит так:



Или другая схема:



Control Plane

Istio включает в себя четыре основных компонента:

- *Pilot*: центральный контроллер, отвечающий за коммуникацию с сайдкарами используя Envoy API. Он считывает правила, описанные в манифестах istio, и отправляет их в Envoy для настройки прокси-сервисов.

Отвечает за service discovery, управление трафиком, роутинг, устойчивостью сети с помощью таймаутов и circuit breaking.

- *Citadel*: Identity and Access management – шифрование трафика, аутентификация сервисов и пользователей, управление ключами шифрования. См. Istio Security.
- *Galley*: управление конфигурацией – валдиация, обработка новых настроек, отправка их остальным компонентам системы
- *Mixer*: мониторинг, метрики, логи, контроль трафика

В версии 1.9.1 все они, кроме Pilot, который работает в отдельном Docker-контейнере, объединены в отдельный бинарный файл istiod, плюс Ingress и Egress Controllers – контроллеры входящего и исходящего трафика

Data Plane

Состоит из sidecar-контейнеров, которые в Kubernetes запускаются внутри подов через kube-inject, см. Installing the Sidecar.

Контейнеры являются инстансами Envoy-прокси, которые позволяют:

- Dynamic service discovery
- Load balancing
- TLS termination
- HTTP/2 and gRPC proxies
- Circuit breakers
- Health checks
- Deployment staged rollouts
- Rich metrics

Почитать по теме:

- Architecture
- Istio Architecture

Модель сети Istio

Перед тем, как приступать к запуску и настройке приложения и Istio – кратко рассмотрим ресурсы, которые участвуют в управлении трафиком.

При установке Istio создаёт ресурс Ingress Gateway (и Egress Gateway, если было определено при установке) – новый объект в Kubernetes, который описывается в Kubernetes CRD при установке Istio.

В AWS при настройках по-умолчанию Ingress Gateway создаёт AWS Classic LoadBalancer, т.к. Ingress Gateway является Kubernetes-объектом Ingress с типом LoadBalancer.

“За” Ingress Gateway создаётся ресурс Gateway, который также описывается в Kubernetes CRD во время установки Istio и описывает хосты и порты, на которые будет отправляться трафик через этот Gateway.

Далее следует ещё один ресурс – VirtualService, который описывает маршрутизацию трафика, прошёдшего через Gateway, и отправляет его на Kubernetes Services следуя заданным Destination rules.

Ingress Gateway состоит из двух компонентов – Kubernetes Pod с инстансом Envoy, который управляет входящим трафиком, и Kubernetes Ingress, который принимает подключения.

В свою очередь Gateway и VirtualService управляют конфигурацией Envoy, который является Ingress Gateway controller.

Т.е. в целом схема прохождения пакета выглядит так:

1. пакет попадает на порт внешнего LoadBalancer, который шлёт его на порт Kubernetes WorkerNode
2. там пакет попадает к Istio IngressGateway Service
3. перенаправляется на Istio IngressGateway Pod
4. этот под настраивается через Gateway и VirtualService
1. Gateway описывает порты, протоколы, SSL-сертификаты
2. VirtualService описывает роутинг пакета к нужному Kubernetes Service нашего приложения
5. Istio IngressGateway Pod отправляет пакет на Service приложения
6. сервис приложения отправляет пакет на под с приложением

Почитать по теме:

- Understanding Istio Ingress Gateway in Kubernetes
- Traffic Management и Traffic Management

Запуск Istio в Kubernetes

Istio поддерживает различные Deployment models. В нашем случае используется один EKS-кластер, а поды в нём работают в общей VPC сети.

Кроме того, Istio поддерживает различные Config Profiles, из которых нам может быть интересен *default* – установка istiod и istio-ingressgateway, *demo* – аналогично, но с запуском istio-egressgateway и *preview* – для исследования новых возможностей, не включенных пока в основной релиз Istio.

Установить Istio можно разными способами – с помощью istioctl из манифест-файлов, из Helm-чарта, либо с помощью Ansible.

Также, стоит обратить внимание на Mutual TLS (mTLS) – см. Permissive mode. Если кратко, то по-умолчанию Istio устанавливается в Permissive mode, что позволяет уже существующим приложениям продолжать коммуникацию используя plaintext-трафик. При этом новые подключения через Envoy-контейнеры, sidecars, уже будут выполняться с TLS-шированием.

Пока выполним установку руками, а на Дев и Прод кластера Кубера скорее всего будем инсталить через Helm.

Загружаем Istio:

```
curl -L https://istio.io/downloadIstio | sh -
cd istio-1.9.1/
```

istioctl использует файл `~/.kube/config`, который можно переопределить с помощью `--kubeconfig` или указать нужный контекст через `--context`.

В загруженном только что каталоге он находится в папке bin – добавляем его в `$PATH`:

```
export PATH=$PWD/bin:$PATH
```

Проверяем:

```
istioctl version
no running Istio pods in "istio-system"
1.9.1
```

Генерируем kubeconfig для нового тестового кластера (для AWS EKS, если используется Minikube – он при старте сгенерирует конфиг сам):

```
aws eks update-kubeconfig --name bttrm-eks-test-1-18 --alias iam-bttrm-eks-root-role-kubectl@bttrm-eks-test-1-18
Added new context iam-bttrm-eks-root-role-kubectl@bttrm-eks-test-1-18 to /home/setevoy/.kube/config
```

Устанавливаем Istio с профилем *default*:

```
istioctl install --set profile=default -y
 Istio core installed
 Istiod installed
 Ingress gateways installed
 Installation complete
```

Проверяем версию ещё раз:

```
istioctl version
client version: 1.9.1
control plane version: 1.9.1
data plane version: 1.9.1 (1 proxies)
```

Поды Istio:

```
kubectl -n istio-system get pod
NAME                                READY   STATUS    RESTARTS   AGE
istio-ingressgateway-d45fb4b48-jsz9z   1/1     Running   0          64s
istiod-7475457497-6xskm                1/1     Running   0          77s
```

Envoy

Kubernetes стал де-факто средой выполнения для приложений микрослужб на основе контейнеров, но сама по себе эта среда оркестрации не обеспечивает всей инфраструктуры, необходимой для работы распределенной системы. Микросервисы обычно взаимодействуют через протоколы уровня 7, такие как HTTP, gRPC или WebSockets, и поэтому жизненно важно иметь возможность принимать решения о маршрутизации, управлять метаданными протокола и наблюдать на этом уровне. Однако традиционные балансировщики нагрузки и пограничные прокси в основном ориентированы на трафик L3/4. Здесь в игру вступает Envoy Proxy .

Прокси-сервер Envoy был с нуля разработан командой инженеров Lyft как универсальная плоскость данных для современного распределенного мира, ориентированного на L7, с широкой поддержкой протоколов L7, API в реальном времени для управления его конфигурацией, первоклассной наблюдаемостью, и высокая производительность при небольшом объеме памяти. Однако обширный набор функций и гибкость работы Envoy также делают его настройку очень сложной — это видно из его богатого, но многословного синтаксиса плоскости управления.

6. Контейнеры, поды, deployment, statefulset, services, endpoints Kubernetes

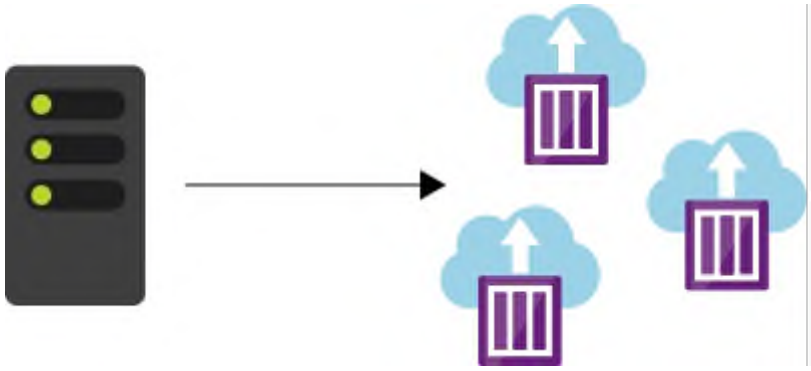
Характеристики отделенных микрослужб в сочетании с атомарностью контейнеров позволяют масштабировать приложения и реагировать на повышение спроса путем развертывания дополнительных экземпляров контейнеров. При этом в случае снижения спроса масштаб уменьшается. При разработке сложных решений, таких как отслеживание дронов, процесс развертывания, обновления, мониторинга и удаления контейнеров сопряжен с множеством трудностей.

Прежде чем рассматривать Kubernetes, давайте обобщим несколько ключевых понятий, связанных с рабочими нагрузками в контейнере.

Что такое управление контейнерами?

Управление контейнерами — это процесс организации, добавления, удаления или обновления значительного количества контейнеров.

Приложение отслеживания дронов состоит из нескольких микрослужб, отвечающих за такие задачи, как кэширование, постановка в очередь или обработка данных. Каждая из этих служб размещается в контейнере и может развертываться, обновляться и масштабироваться независимо от других.



Например, на веб-сайте приложения для отслеживания дронов вы обнаруживаете, что в определенное время в течение дня вам требуются дополнительные экземпляры службы кэширования сайта для поддержания производительности, так что вам необходимо добавлять дополнительные экземпляры контейнера службы кэширования.

Теперь предположим, что вы увеличили количество экземпляров кэширования и хотите развернуть новую версию микрослужбы. Необходимо обновить *все* активные контейнеры.

Управление контейнерами поможет вам в этих задачах, которые иначе были бы ручными.

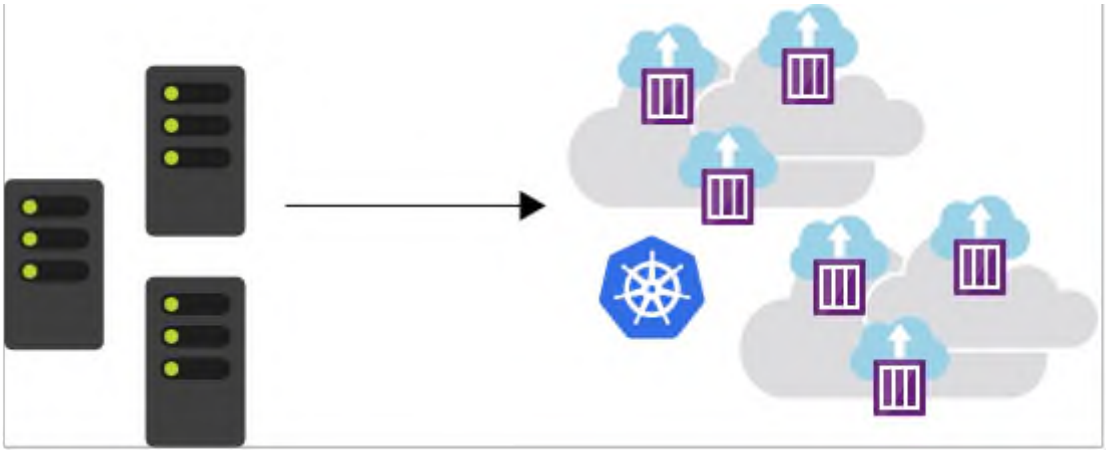
Что такое оркестрация контейнеров?

Оркестратор контейнеров — это система, которая автоматически развертывает контейнерные приложения и управляет ими. Например,

оркестратор динамически реагирует на изменения в среде, чтобы увеличивать или уменьшать количество развернутых экземпляров управляемого приложения. Кроме того, он может гарантировать, что все развернутые экземпляры контейнеров будут обновлены при выпуске новой версии службы.

Определение Kubernetes

Kubernetes — это переносимая расширяемая платформа с открытым кодом для управления и оркестрации контейнерных рабочих нагрузок. Kubernetes абстрагирует сложные операции по управлению контейнерами, позволяя оркестрировать контейнеры в разных вычислительных средах путем декларативной настройки. Эта платформа оркестрации настолько же проста и гибка в использовании, как и уже известные вам предложения "платформа как услуга" (PaaS) и "инфраструктура как услуга" (IaaS).



Преимущества Kubernetes

Преимущества использования Kubernetes основаны на абстракции задач.

Ниже перечислены эти задачи.

- Самостоятельное восстановление контейнеров. Например, перезапуск контейнеров, в которых произошел сбой, или замена контейнеров.
- Масштабирование количества развернутых контейнеров приложений динамически в зависимости от спроса.
- автоматические последовательные обновления и откат контейнеров;
- управление хранилищем;
- управление сетевым трафиком;
- хранение конфиденциальных данных, таких как пароли, и управление ими.

Важно!

Помните, что для всех перечисленных выше аспектов Kubernetes требуется настройка и хорошее понимание базовых технологий. Например,

для настройки сети Kubernetes необходимо понимать, что такое виртуальные сети, подсистемы балансировки нагрузки и обратные прокси-серверы.

Рекомендации по Kubernetes

Kubernetes позволяет представить центр обработки данных как один компьютерный ресурс. Вам не нужно беспокоиться о том, как и где развертываются контейнеры. Ваша задача заключается в развертывании и масштабировании приложений по мере необходимости.

Однако важно понимать, что при установке Kubernetes вы не получаете сразу все компоненты, необходимые для управления контейнерным решением и его оркестрации.

- Компоненты, отвечающие за такие аспекты, как развертывание, масштабирование, балансировка нагрузки, ведение журналов и мониторинг, добавляются по мере необходимости. Ваша задача — подобрать оптимальное решение в соответствии с вашими потребностями.
- Kubernetes не накладывает ограничений на типы приложений, которые могут выполняться на платформе. Если приложение может выполняться в контейнере, оно может выполняться и в Kubernetes. Для оптимального использования контейнерных решений разработчики должны разбираться в таких концепциях, как архитектура микрослужб.
- Kubernetes не предоставляет ПО промежуточного слоя, платформы обработки данных, базы данных, кэш или системы хранения данных кластера. Все эти компоненты добавляются как контейнеры или в составе другого предложения.
- Чтобы платформа Kubernetes запускала контейнеры, она должна поддерживать среду выполнения контейнеров, например Docker. Среда выполнения контейнеров — это объект, отвечающий за управление контейнерами. Например, среда выполнения контейнеров запускается, останавливается и сообщает о состоянии контейнера.
- Вы занимаетесь обслуживанием среды Kubernetes. Например, вам необходимо управлять обновлением ОС, а также установкой и обновлением самой платформы Kubernetes. Кроме того, вы управляете конфигурацией оборудования узлов, в том числе сетевого оборудования, памяти и хранилища.

Облачные сервисы, такие как Azure Kubernetes Service (AKS), снижают влияние этих проблем, предоставляя размещенную среду Kubernetes. Эти службы также упрощают развертывание контейнерных приложений и управление ими в Azure. Благодаря AKS вы получаете преимущества платформы Kubernetes с открытым кодом и при этом избегаете сложностей и

временных затрат, связанных с управлением собственным кластером Kubernetes.

Что такое кластер компьютеров?

Кластер — это набор компьютеров, настроенных для совместной работы и представленных в виде единой системы. Компьютеры, настроенные в кластере, обычно выполняют одинаковые задачи. Например, на всех них размещаются веб-сайты, программные интерфейсы (API) или выполняются ресурсоемкие операции.

Кластер использует централизованное программное обеспечение, ответственное за планирование этих задач и управление ими. Компьютеры в кластере, где выполняются задачи, называются *узлами*, а компьютеры, на которых выполняется программное обеспечение для планирования, — *уровнями управления*.

Архитектура Kubernetes

Как вы помните, оркестратор — это система, которая развертывает приложения и управляет ими. Вы также узнали, что кластер — это набор компьютеров, которые работают вместе и представлены в виде единой системы. Kubernetes используется в качестве программного обеспечения оркестрации и кластера для развертывания приложений и реагирования на изменения потребности в ресурсах вычислений.

Кластер Kubernetes содержит по крайней мере один уровень управления и один узел или несколько. Уровень управления и экземпляры узлов могут быть физическими устройствами, виртуальными машинами или экземплярами в облаке. Операционной системой сервера виртуальных машин в Kubernetes по умолчанию является Linux с поддержкой по умолчанию для рабочих нагрузок на основе Linux.

Также на узлах кластера можно запускать рабочие нагрузки Майкрософт с помощью Windows Server 2019 или более поздней версии. Предположим, что служба обработки данных в приложении отслеживания дронов написана как приложение .NET 4.5, использующее определенные вызовы API ОС Windows. Эта служба может выполняться только на узлах, работающих под управлением ОС Windows Server.

Теперь подробнее рассмотрим уровни управления и рабочие узлы, а также программное обеспечение, которое выполняется в каждом из них. Понимание роли каждого компонента и того, где каждый компонент работает в кластере, помогает при установке Kubernetes.

Уровень управления Kubernetes

Главный узел Kubernetes, также известный как уровень управления в кластере Kubernetes, запускает набор сервисов, управляющих функциональностью оркестрации в Kubernetes.

С точки зрения обучения имеет смысл использовать один уровень управления в тестовой среде при изучении функций Kubernetes. Однако в рабочих и облачных развертываниях, таких как Служба Azure Kubernetes (AKS), вы обнаружите, что предпочтительной конфигурацией является развертывание с высоким уровнем доступности с несколькими (от трех до пяти) реплицированными уровнями управления.

Примечание

Тот факт, что главный узел запускает конкретное программное обеспечение для поддержания состояния кластера, не исключает выполнение других вычислительных рабочих нагрузок на нем. Однако, как правило, на нем не следует выполнять некритические и пользовательские рабочие нагрузки приложений.

Узел Kubernetes

Узел в кластере Kubernetes — это место выполнения вычислительных рабочих нагрузок. Каждый узел взаимодействует с уровнем управления через сервер API, чтобы сообщить об изменениях состояния на узле.

Службы, работающие на уровне управления

Kubernetes полагается на несколько административных сервисов, работающих на плоскости управления. Эти службы управляют такими аспектами, как взаимодействие с компонентами кластера, планирование рабочей нагрузки и сохранение состояния кластера.

Следующие сервисы составляют уровень управления в кластере Kubernetes:

- Сервер API
- Резервное хранилище
- Планировщик
- Диспетчер контроллеров
- Диспетчер облачных контроллеров

Что такое сервер API?

Сервер API можно представить как внешний интерфейс для уровня управления в кластере Kubernetes. Все взаимодействия между компонентами в Kubernetes осуществляются через этот API.

Например, вы, как пользователь, используете приложение командной строки kubectl, которое позволяет выполнять команды на сервере API

кластера Kubernetes. Компонент, предоставляющий этот API, называется kube-apiserver, и можно развернуть несколько экземпляров этого компонента для поддержки масштабирования в кластере.

Этот API предоставляет RESTful API, который позволяет публиковать команды или файлы конфигурации на основе YAML. YAML — это понятный человеку стандарт сериализации данных для языков программирования. Файлы YAML используются для определения предполагаемого состояния всех объектов в кластере Kubernetes.

Пусть, например, необходимо увеличить количество экземпляров приложения в кластере. Вы определяете новое состояние с помощью файла на основе YAML и отправляете его на сервер API. Сервер API проверит конфигурацию, сохранит ее в кластере и, наконец, произведет настроенное увеличение числа развернутых приложений.

Что такое резервное хранилище?

Резервное хранилище — это постоянное хранилище, используемое кластером Kubernetes для хранения полной конфигурации кластера Kubernetes. Kubernetes использует распределенное, высокодоступное и надежное хранилище пар "ключ-значение", которое называется etcd. В этом хранилище пар "ключ-значение" хранится текущее состояние, а также требуемое состояние всех объектов в кластере.

В рабочем кластере Kubernetes официально рекомендуется наличие от трех до пяти реплицируемых экземпляров базы данных etcd для обеспечения высокой доступности.

Что такое планировщик?

Планировщик — это компонент, отвечающий за назначение рабочих нагрузок на всех узлах. Планировщик отслеживает создаваемые в кластере контейнеры и назначает их узлам.

Что такое диспетчер контроллеров?

Диспетчер контроллеров запускает и отслеживает контроллеры, настроенные для кластера через сервер API.

Kubernetes использует контроллеры для отслеживания состояния объектов в кластере. Каждый контроллер работает в неразрывном цикле, отслеживая события в кластере и реагируя на них. Например, есть контроллеры для мониторинга узлов, контейнеров и конечных точек.

Контроллер взаимодействует с сервером API для определения состояния объекта. Если текущее состояние отличается от нужного, контроллер примет меры, чтобы обеспечить требуемое состояние.

Предположим, что один из трех контейнеров, работающих в кластере, перестает отвечать на запросы и отключается. В этом случае контроллер решает, требуется ли запустить новые контейнеры, чтобы обеспечить доступность приложений. Если нужно, чтобы всегда работало три контейнера, будет запланирован запуск нового.

Что такое диспетчер облачных контроллеров?

Диспетчер облачных контроллеров интегрируется с базовыми облачными технологиями в кластере, если кластер работает в облачной среде. Этими службами, например, могут быть подсистемы балансировки нагрузки, очереди и хранилище.

Службы, выполняющиеся в узле

На узле Kubernetes запущено несколько служб, которые управляют выполнением рабочих нагрузок.

Следующие службы выполняются на узле Kubernetes:

- kubelet
- kube-proxy
- Среда выполнения контейнера

Что такое kubelet?

kubelet — это агент, который выполняется на каждом узле в кластере и отслеживает рабочие запросы с сервера API. Он гарантирует, что запрошенная единица работы запущена и работоспособна.

kubelet выполняет мониторинг узлов и гарантирует, что контейнеры, запланированные на каждом узле, выполняются должным образом. kubelet управляет только контейнерами, созданными Kubernetes. Он не несет ответственности за повторное планирование заданий на других узлах, если текущий узел не может выполнить работу.

Что такое kube-proxy?

Компонент kube-proxy отвечает за локальную сеть кластера и выполняется на каждом узле. Он гарантирует, что каждый узел имеет уникальный IP-адрес. Он также реализует правила для управления маршрутизацией и балансировкой нагрузки трафика с помощью iptables и IPVS.

Эта прокси-служба сама не предоставляет службу DNS. Рекомендуется использовать установленную по умолчанию надстройку кластера DNS на основе CoreDNS.

Что такое среда выполнения контейнера?

Среда выполнения контейнера — это базовое программное обеспечение, которое выполняет контейнеры в кластере Kubernetes. Среда

выполнения отвечает за получение, запуск и остановку образов контейнеров. Kubernetes поддерживает несколько сред выполнения контейнеров, в том числе Docker, rkt, CRI-O, containerd и frakti. Поддержка многих типов среды выполнения контейнеров основана на интерфейсе среды выполнения контейнеров (CRI). CRI — это конструкция подключаемого модуля, которая позволяет kubelet взаимодействовать с доступной средой выполнения контейнеров.

Среда выполнения контейнеров по умолчанию в AKS — Docker. Однако вы также можете использовать Kata Containers и containerd (доступен как управляющая программа для Windows и Linux).

Взаимодействие с кластером Kubernetes

Kubernetes предоставляет программу командной строки kubectl для управления кластером. kubectl можно использовать для отправки команд на уровень управления кластера или получения сведений обо всех объектах Kubernetes через сервер API.

kubectl использует файл конфигурации, который содержит следующие сведения о конфигурации.

- Конфигурация **Cluster** определяет имя кластера, сведения о сертификате и конечную точку API службы, связанную с кластером. Это определение позволяет подключаться с одной рабочей станции к нескольким кластерам.
- Конфигурация **User** определяет пользователей и их уровни разрешений при доступе к настроенным кластерам.
- Конфигурация **Context** обединяет кластеры и пользователей в группы с понятным именем. Например, вы можете назвать кластер разработки и рабочий кластер dev-cluster и prod-cluster соответственно.

Можно настроить kubectl для подключения к нескольким кластерам, указав правильный контекст в качестве части синтаксиса командной строки.

Модули pod Kubernetes

Модуль pod представляет отдельный экземпляр приложения, работающего в Kubernetes. Рабочие нагрузки, выполняемые в Kubernetes, являются контейнерными приложениями. В отличие от среды Docker вы не можете запускать контейнеры непосредственно в Kubernetes. Контейнер упаковывается в объект Kubernetes, который называется pod. Модуль pod является наименьшим объектом, который можно создать в Kubernetes.

Один модуль pod также может содержать группу из одного или нескольких контейнеров. Однако обычно модуль pod не содержит несколько экземпляров одного приложения.

В модуле pod содержатся сведения о конфигурации общего хранилища и сети, а также описание того, как выполнять упакованные контейнеры. Шаблоны модулей pod используются для определения сведений о модулях, которые выполняются в кластере. Шаблоны модулей pod — это файлы YAML, которые можно использовать повторно и включать в другие объекты для управления развертываниями pod.

Пусть, например, требуется развернуть веб-сайт в кластер Kubernetes. Вы создаете файл определения pod, в котором указаны образы и конфигурация контейнера приложения. Затем вы развертываете файл определения pod в Kubernetes.

Вряд ли веб-сайт будет единственным компонентом веб-приложения в решении. Веб-приложение обычно имеет хранилище данных и другие вспомогательные элементы. Модули pod Kubernetes также могут содержать несколько контейнеров.

Предположим, что сайт использует базу данных. Веб-сайт упаковывается в основной контейнер, а база данных — во вспомогательный контейнер. Чтобы эти два контейнера могли функционировать и взаимодействовать друг с другом, они должны работать в среде, предоставляющей ОС узла, сетевой стек, пространства имен ядра, общую память и тома для сохранения данных. Модуль pod — это изолированная среда, которая предоставляет все эти службы приложению. Она также позволяет контейнерам совместно использовать свой назначенный IP-адрес.

Так как вы потенциально можете создать множество модулей pod, которые выполняются на многих узлах, найти их бывает трудно. Вы распознаете и группируете модули pod, используя строковые метки, указанные при определении модуля.

Жизненный цикл модуля pod в Kubernetes

Модули pod Kubernetes имеют четкий жизненный цикл, который влияет на способ их развертывания, запуска и обновления. Сначала вы отправляете файл YAML с манифестом модуля pod в кластер. После отправки и сохранения файла манифеста в кластере он определяет желаемое состояние модуля pod. Планировщик планирует модуль pod на работоспособном узле, который имеет достаточное количество ресурсов для запуска модуля.

Ниже приведены этапы жизненного цикла модуля pod.

Этап	Описание
Ожидает	Модуль pod был принят кластером, но один или несколько контейнеров не настроены или готовы к выполнению. Состояние ожидания включает в себя время,

Этап	Описание
	в течение которого модуль pod ожидает планирования, и время, затраченное на загрузку образов контейнеров.
Выполнение	Модуль pod переходит в состояние выполнения после того, как все ресурсы в нем будут готовы.
Выполнено	Модуль pod переходит в состояние успешного выполнения, когда завершает свою задачу.
Сбой	Сбои модулей pod могут возникать по различным причинам. Возможно, произошел сбой контейнера в модуле pod, что привело к завершению работы остальных контейнеров. Или, возможно, образ не был найден во время подготовки контейнеров pod. В подобных случаях может возникать сбой модуля pod. Модули pod могут переходить в состояние сбоя либо из состояния ожидания, либо из состояния выполнения. Определенный сбой также может вернуть модуль pod в состояние ожидания.
Неизвестно	Если состояние модуля pod не может быть определено, он находится в неизвестном состоянии.

Модули pod хранятся в кластере, пока контроллер, уровень управления или пользователь явно не удалят их. Когда модуль pod удаляется и заменяется новым, этот новый модуль pod представляет собой совершенно новый экземпляр на основе манифеста pod.

Кластер не сохраняет состояние pod или динамически назначенную конфигурацию. Например, он не сохраняет идентификатор или IP-адрес модуля pod. Этот аспект влияет на развертывание модулей pod и разработку приложений. Например, нельзя полагаться на заранее назначенные IP-адреса для модулей pod.

Состояния контейнеров

Помните, что этапы представляют собой общие сведения о том, где находится модуль pod в жизненном цикле. При проверке модулей pod кластер использует три состояния для отслеживания контейнеров внутри модуля.

Состояние	Описание
Ожидание	Состояние по умолчанию контейнера и состояние, в котором находится контейнер, если не выполняется или не завершен.

Состояние	Описание
Выполнение	Контейнер выполняется так, как ожидалось, без проблем.
Завершен	Контейнер больше не выполняется. Причина перехода в это состояние в том, что все задачи завершены или по какой-либо причине произошел сбой контейнера. Код причины и выхода доступен при отладке обоих вариантов.

Как работают развертывания Kubernetes

Приложение отслеживания дронов содержит несколько компонентов, которые развертываются отдельно друг от друга. Ваша задача — настроить развертывания этих компонентов в кластере. Здесь мы рассмотрим некоторые варианты развертывания, доступные для этих компонентов.

Варианты развертывания модулей pod

Существует несколько вариантов управления развертыванием модулей pod в кластере Kubernetes при использовании kubectl. Допустимые значения:

- Шаблоны модулей pod
- Контроллеры репликации
- Наборы реплик
- Развертывания

Для развертывания одного модуля pod или нескольких можно использовать любое из четырех определений типов объекта Kubernetes. Эти файлы используют YAML для описания предполагаемого состояния модулей pod, которые будут развернуты.

Что такое шаблон pod?

Шаблон pod позволяет определить конфигурацию модуля, который вы хотите развернуть. Шаблон содержит такие сведения, как имя образа контейнера и реестр контейнеров, используемый для получения образов. Шаблон также может включать сведения о конфигурации среды выполнения, например используемые порты. Шаблоны определяются с помощью YAML таким же образом, как и при создании файлов DOCKER.

Шаблоны можно использовать для ручного развертывания модулей Pod. Тем не менее развернутый вручную модуль pod не перезапускается после сбоя. Он удаляется или завершается. Чтобы управлять жизненным циклом Pod, необходимо создать объект Kubernetes более высокого уровня.

Что такое контроллер репликации?

Контроллер репликации использует шаблоны pod и определяет указанное число модулей pod, которые необходимо выполнить. Контроллер

помогает запустить несколько экземпляров одного и того же модуля pod и убедиться в том, что модули pod всегда выполняются на одном узле в кластере или нескольких. Контроллер заменит модули pod, запущенные таким образом, новыми модулями pod, если в них происходит сбой, они удаляются или завершаются.

Предположим, что вы развертываете интерфейсный веб-сайт для отслеживания дронов и пользователи начинают переходить на этот веб-сайт. Если по какой-либо причине произошел сбой всех модулей pod, веб-сайт будет недоступен для пользователей, пока не будут запущены новые модули pod. Контроллер репликации помогает убедиться, что веб-сайт всегда доступен.

Что такое набор реплик?

Наборы реплик заменяют контроллер репликации как предпочтительный способ развертывания реплик. Набор реплик включает те же функциональные возможности, что и контроллер репликации. Однако он также имеет дополнительный параметр конфигурации для включения значения селектора.

Селектор позволяет набору реплик определить все модули pod, выполняющиеся под ним. С помощью этой функции можно управлять модулями pod, помеченными значением, которое совпадает со значением селектора, но не созданными с помощью реплицируемого набора.

Что такое развертывание?

Развертывание создает объект управления на один уровень выше, чем набор реплик, и позволяет развертывать обновления для модулей pod в кластере и управлять ими.

Предположим, в кластере развернуто пять экземпляров приложения. Существует пять модулей pod с версией 1.0.0 вашего приложения.

Если вы решите обновить приложение вручную, можно удалить все модули pod, а затем запустить новые модули pod с версией приложения 2.0.0. При такой стратегии в приложении будут возникать простои.

Вместо этого следует выполнить последовательное обновление, когда вы запускаете модули pod, в которых выполняется новая версия приложения, прежде чем удалять модули pod со старой версией. Последовательные обновления запускают по одному модулю pod вместо того, чтобы одновременно завершать все более старые модули. Развертывания учитывают количество реплик, настроенных в разделе со сведениями о наборе реплик. Это позволит поддерживать число модулей pod, заданного в наборе реплик, при замене старых модулей pod новыми.

Развертывания по умолчанию предоставляют стратегию последовательного обновления для модулей pod. Также можно использовать стратегию повторного создания. Эта стратегия завершит работу модулей pod перед запуском новых модулей.

Развертывания также предоставляют стратегию отката, которую можно выполнить с помощью kubectl.

Развертывания используют файлы определений на основе YAML, которые упрощают управление развертываниями. Помните, что развертывания позволяют применять любые изменения в кластере. Например, можно развертывать новые версии приложения, обновлять метки и запускать другие реплики модулей pod.

kubectl имеет удобный синтаксис для автоматического создания развертывания при использовании команды kubectl run для развертывания модуля pod. Эта команда создает развертывание с необходимым набором реплик и модулями pod. Однако команда не создает файл определения. Рекомендуется управлять всеми развертываниями с помощью файлов определений развертывания и отслеживать изменения с помощью системы управления версиями.

Рекомендации по развертыванию

Kubernetes предъявляет особые требования к настройке сети и хранилища для кластера. Настройка этих двух аспектов влияет на решения о том, как предоставлять приложения в сети кластера и хранить данные.

Например, каждая из служб в приложении отслеживания дронов предъявляет особые требования к доступу пользователей, межпроцессному доступу в сети и хранению данных. Давайте рассмотрим эти аспекты кластера Kubernetes и узнаем, как они влияют на развертывание приложений.

Сети Kubernetes

Предположим, у вас есть кластер с одним уровнем управления и двумя узлами. При добавлении узлов в Kubernetes IP-адрес автоматически назначается каждому узлу из внутреннего диапазона частной сети. Предположим, что диапазон локальной сети — 192.168.1.0/24.

Каждый развертываемый модуль pod назначает IP-адрес из пула IP-адресов. Допустим, что в конфигурации используется сетевой диапазон 10.32.0.0/12, как показано на приведенном ниже рисунке.

По умолчанию модули pod и узлы не могут взаимодействовать друг с другом, используя разные диапазоны IP-адресов.

Более того, эти модули pod являются временными. IP-адрес модуля pod является временным и не может использоваться для повторного

подключения к вновь созданному модулю pod. Эта конфигурация влияет на взаимодействие приложения с его внутренними компонентами, а также на то, как вы и службы взаимодействуете с ним извне.

Для упрощения взаимодействия Kubernetes ожидает следующих настроек сети.

- Модули pod могут взаимодействовать друг с другом между узлами без преобразования сетевых адресов (NAT).
- Узлы могут взаимодействовать со всеми модулями pod и, наоборот, без NAT.
- Агенты на узле могут взаимодействовать со всеми узлами и модулями pod.

Kubernetes предлагает несколько сетевых решений, которые можно установить для настройки сети. Например, Antrea, Cisco Application Centric Infrastructure (ACI), Cilium, Flannel, Kubenet, VMware NSX-T, Weave Net.

Поставщики облачных служб также предоставляют собственные сетевые решения. Например, служба Azure Kubernetes Service (AKS) поддерживает сетевой интерфейс контейнера виртуальной сети Azure, Kubenet, Flannel, Cilium и Antrea.

Службы Kubernetes

Служба — это объект Kubernetes, который обеспечивает стабильное сетевое подключение для модулей pod. Служба Kubernetes используется для обеспечения обмена данными между узлами, модулями pod и пользователями приложения (как внутренними, так и внешними) в кластере.

Kubernetes назначает службе IP-адрес при создании, как узлу или модулю pod. Эти адреса назначаются из диапазона IP-адресов кластера служб. Например, 10.96.0.0/12. Службе также назначается DNS-имя на основе имени службы и IP-порт.

В приложении отслеживания дронов сетевое взаимодействие происходит следующим образом.

- Веб-сайт и RESTful API доступны пользователям за пределами кластера.
- Службы кэша в памяти и очереди сообщений доступны для интерфейсных и RESTful API соответственно, но не для внешних пользователей.
- Очереди сообщений требуется доступ к службе обработки данных, но не к внешним пользователям.
- База данных NoSQL доступна для кэша в памяти и службы обработки данных, но не для внешних пользователей.

Для поддержки этих сценариев можно настроить три типа служб для предоставления компонентов приложения.

Служба	Описание
ClusterIP	Адрес, назначенный службе, предоставляющей нашу службу для набора служб в кластере. Например, связь между интерфейсными и серверными компонентами приложения.
NodePort	Порт узла в диапазоне от 30000 до 32767, который уровень управления Kubernetes назначает службе. Например, узел clusters01 имеет адрес 192.169.1.11. Затем вы настраиваете службу с целевым портом на модуле pod, который вы хотите предоставить. Например, настраиваете порт 80 на модуле pod, который работает на одном из внешних интерфейсов. Теперь вы можете получить доступ к интерфейсной части через IP-адрес и порт узла.
LoadBalancer	Подсистема балансировки нагрузки, позволяющая распределять нагрузку между узлами, на которых выполняется приложение, и предоставлять модуль pod для доступа из общедоступной сети. Обычно подсистемы балансировки нагрузки настраиваются при использовании поставщиков облачных служб. В этом случае трафик из внешней подсистемы балансировки нагрузки направляется в модули pod, выполняющие приложение.

В приложении отслеживания дронов можно предоставить доступ к веб-сайту отслеживания и RESTful API с помощью LoadBalancer и службы обработки данных с ClusterIP.

Группировка модулей pod

Управлять модулями pod по IP-адресу неудобно. IP-адреса pod изменяются по мере их повторного создания контроллерами и могут работать с любым числом модулей pod.

Объект службы позволяет управлять отдельными модулями pod в кластере с помощью меток селектора. Установите метку селектора в определении службы в соответствии с метким модулем pod, указанной в файле определения pod.

Предположим, у вас есть много работающих модулей pod. Только некоторые из этих модулей являются интерфейсными, и вы хотите настроить службу LoadBalancer, предназначенную только для интерфейсных модулей pod. Вы можете применить службу для предоставления этих модулей pod,

указав метку pod в качестве значения селектора в файле определения службы. Теперь служба будет группировать только те модули pod, которые соответствуют метке. Если модуль pod удаляется и создается повторно, новый модуль pod автоматически добавляется в группу служб с помощью соответствующей метки.

Хранилище Kubernetes

Kubernetes использует ту же концепцию тома хранилища, что и Docker. Тома Docker менее управляемые по сравнению с томами Kubernetes, так как вы не можете управлять их временем существования. Время существования тома Kubernetes задается явно и соответствует времени существования модуля pod. Это сопоставление значений времени существования означает, что том будет существовать дольше, чем контейнеры, выполняющиеся в модуле pod. Однако если модуль pod удаляется, том также будет удален.

Kubernetes предоставляет возможности для подготовки постоянного хранилища с использованием *PersistentVolumes*. Вы также можете запросить определенное хранилище для модулей pod с помощью *PersistentVolumeClaims*.

Вам следует рассмотреть оба этих варианта при развертывании компонентов приложений, требующих постоянного хранения, таких как очереди сообщений и базы данных.

Вопросы интеграции с облаком

Kubernetes не определяет стек технологий, используемых в облачном приложении. В облачной среде, такой как Azure, можно использовать несколько служб за пределами кластера Kubernetes.

Вспомним ранее, что Kubernetes не предоставляет следующие сервисы:

- ПО промежуточного слоя
- Платформы обработки данных
- Базы данных
- Кэши
- Кластерные системы хранения

В этом решении для отслеживания дронов три службы предоставляют функциональные возможности ПО промежуточного слоя: база данных NoSQL, служба кэша в памяти и очередь сообщений. Вы можете использовать MongoDB Atlas для решения NoSQL, Redis — для управления кэшем в памяти и RabbitMQ или Kafka в зависимости от потребностей очереди сообщений.

При использовании облачной среды, такой как Azure, рекомендуется использовать службы за пределами кластера Kubernetes. Это решение может упростить настройку кластера и управление им. Например, можно

использовать *кэш Azure для Redis* для служб кэширования в памяти, *обмен сообщениями служебной шины Azure* для очереди сообщений и *Azure Cosmos DB* для базы данных NoSQL.

Когда следует использовать Kubernetes

Решение об использовании платформы оркестрации контейнеров, такой как Kubernetes, зависит от бизнес-требований и потребностей разработки. Давайте в общих чертах рассмотрим архитектуру решения для отслеживания дронов.

Это решение создается как слабо связанные и работающие друг с другом микрослужбы. Для упрощения проектирования и обслуживания решения эти службы развертываются отдельно друг от друга. Вот как выглядит текущая конфигурация решения.

- Веб-интерфейс: показывает карты и сведения об отслеживаемых дронах.
- Служба кэша: хранит часто запрашиваемые сведения, отображаемые на веб-сайте.
- RESTful API: используется отслеживаемыми дронами для отправки данных об их состоянии, таких как GPS-расположение и уровень заряда аккумулятора.
- Очередь: содержит необработанные данные, собираемые RESTful API.
- Служба обработки данных: извлекает и обрабатывает данные из очереди.
- База данных NoSQL: в ней хранятся обработанные данные отслеживания и сведения о пользователях, полученные с веб-сайта и от службы обработки данных.

В вашей компании разрабатывают и владеют этими службами разные команды. Каждая команда использует контейнеры для создания и развертывания своей службы. Эта новая стратегия позволяет группам разработчиков учитывать требования к современной разработке программного обеспечения в сфере автоматизации, тестирования и общей стабильности и качества.

Изменения в мышлении разработчиков привели к некоторым процессам и бизнес-преимуществам для компании. Например, к более эффективному использованию размещенных вычислительных ресурсов, ускорению выпуска новых функций и расширению охвата клиентов.

Однако сложности с управлением контейнерами заставили вашу компанию рассмотреть решения по оркестрации контейнеров. Ваши команды обнаружили, что масштабировать приложение отслеживания до нескольких развертываний было относительно просто, но масштабирование множества экземпляров и управление ими оказалось сложной задачей.

Также нужно учесть и другие аспекты. Например, работу с контейнерами в случае сбоев, выделение хранилища, настройку сети и управление секретами приложений.

Как вы уже знаете, Kubernetes поддерживает все эти задачи в качестве платформы оркестрации.

Вы хотите использовать Kubernetes, когда ваша компания:

- разрабатывает приложения как микрослужбы;
- разрабатывает приложения как облачные;
- развертывает микрослужбы с помощью контейнеров;
- обновляет контейнеры в большом масштабе;
- требует централизованной сети контейнеров и управление хранилищем.

Когда не следует использовать Kubernetes

Не все приложения необходимо выполнять в Kubernetes. В результате Kubernetes может оказаться неподходящим решением для вашей компании.

Например, усилия по упаковке в контейнеры и развертыванию монолитного приложения могут перевесить преимущества выполнения приложения в Kubernetes. Монолитная архитектура не может с легкостью использовать такие функции, как масштабирование отдельных компонентов или обновление.

Kubernetes предлагает множество преимуществ для бизнеса при разработке, развертывании, управлении и упрощении процессов программного обеспечения. Однако Kubernetes требует длительного обучения. Модульная конструкция Kubernetes представляет потенциально новые концепции, которые разработчикам придется освоить.

Разработчики должны будут использовать современные концепции проектирования при разработке и проектировании приложений. В число этих концепций входит использование микрослужб и контейнеризация этих служб. Также им необходимо экспериментировать с контейнерами и средами оркестрации для оптимального использования всех доступных преимуществ.

Если ваша компания не готова к таким изменениям, Kubernetes, возможно, не ваш вариант.



МОДУЛЬ 18. KUBERNETES КОНФИГУРАЦИЯ РАЗВЕРТЫВАНИЯ

1. Разделы и монтирование
2. Работа с Kubectl
3. Инструменты для упрощения написания конфигурационных файлов. Helm и Jsonnet
4. Поддержка нескольких окружений на примере Qbec
5. Создание и использование секретов

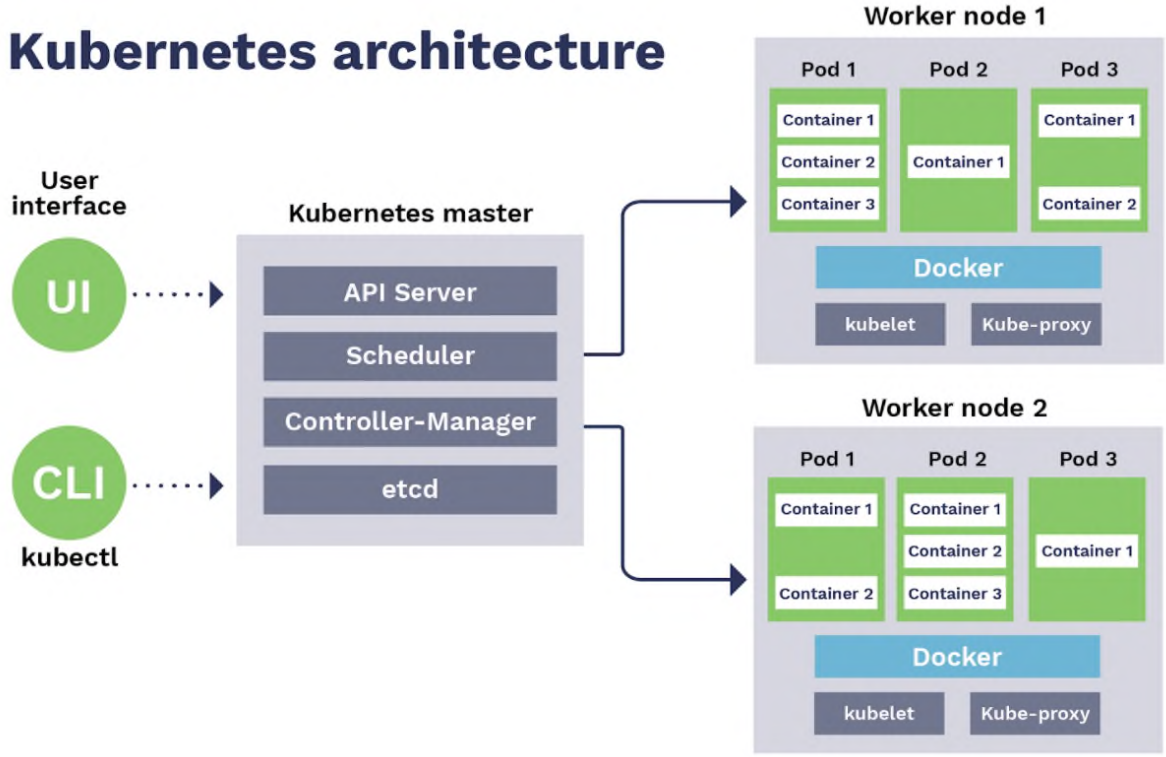
1. Разделы и монтирование

Что такое Kubernetes?

Kubernetes – это портативная расширяемая платформа для управления контейнерами на серверах. Она поставляется с открытым кодом, поэтому на рынке достаточно сервисов и инструментов для декларативной настройки и автоматизации процессов.

Зачем нужен Kubernetes?

Платформа создавалась с учетом наработок в управлении удаленными проектами с масштабными рабочими нагрузками. Поэтому изначально в ней применяются продвинутые технологии, которые давно заменили принципы физического и даже виртуального развертывания систем. Контейнерное построение считается «облегченным» вариантом виртуальных машин.



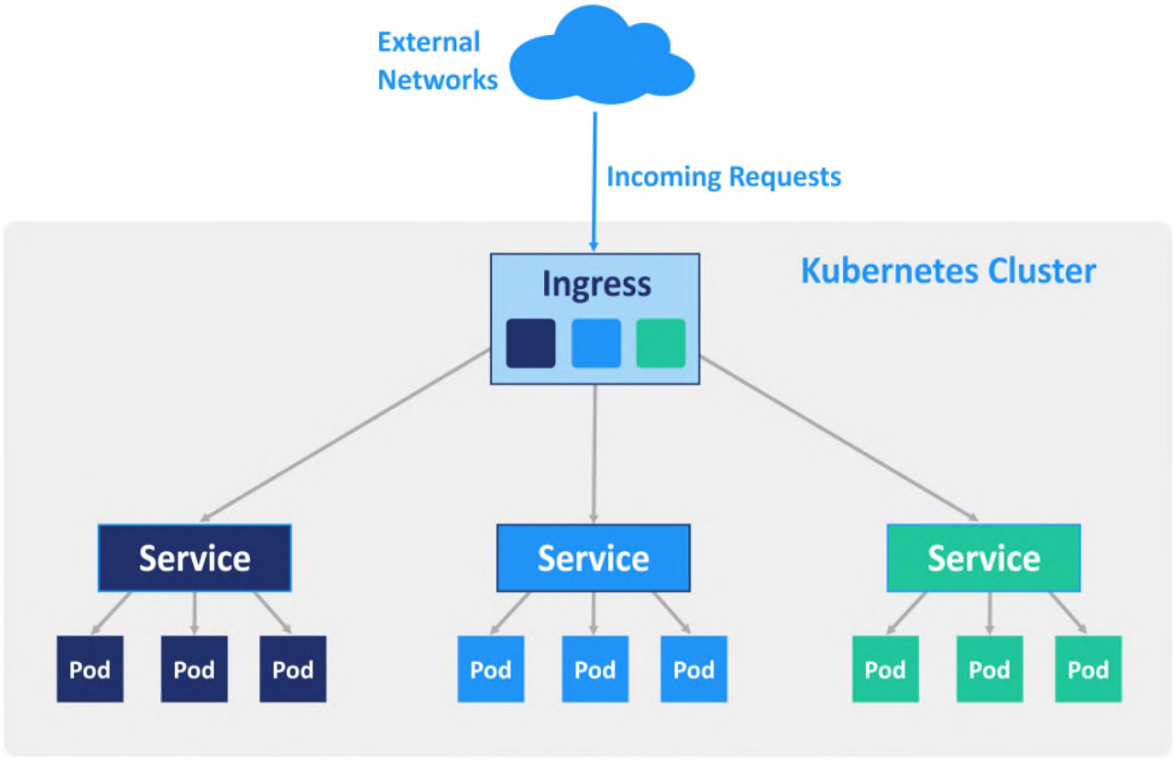
Причины популярности Kubernetes:

- Развертываемые приложения отделены от инфраструктуры для повышения безопасности.
- Разработка и тестирование работают одинаково на локальном компьютере и в облаке.
- Контейнеры свободно переносятся между Ubuntu, CoreOS, RHEL и другими системами.

- Происходит изоляция ресурсов с прогнозированием производительности.
- Вместо монолитного стека одной выделенной машины, приложения разбиваются на независимые друг от друга микросервисы с возможностью динамического развертывания и управления. За счет такого подхода ресурсы используются более грамотно. Приложения наблюдают не только за метриками операционной системы, а еще и другими программами и оборудованием.

Назначение Kubernetes

Контейнеры упрощают блочную разработку продуктов и их последующую сборку перед запуском. Такой подход дает возможность дорабатывать отдельные модули без влияния на работу остальных функций, и все это происходит на платформе фреймворка Kubernetes. Она обрабатывает ошибки в приложениях, занимается масштабированием, содержит шаблоны развертывания и пр.



Перечень возможностей:

- Обнаружение контейнера происходит по имени DNS или IP-адресу.
- Система самостоятельно балансирует нагрузку и распределяет трафик в сети.

•Подключение выбранного типа хранилища происходит в автоматическом режиме.

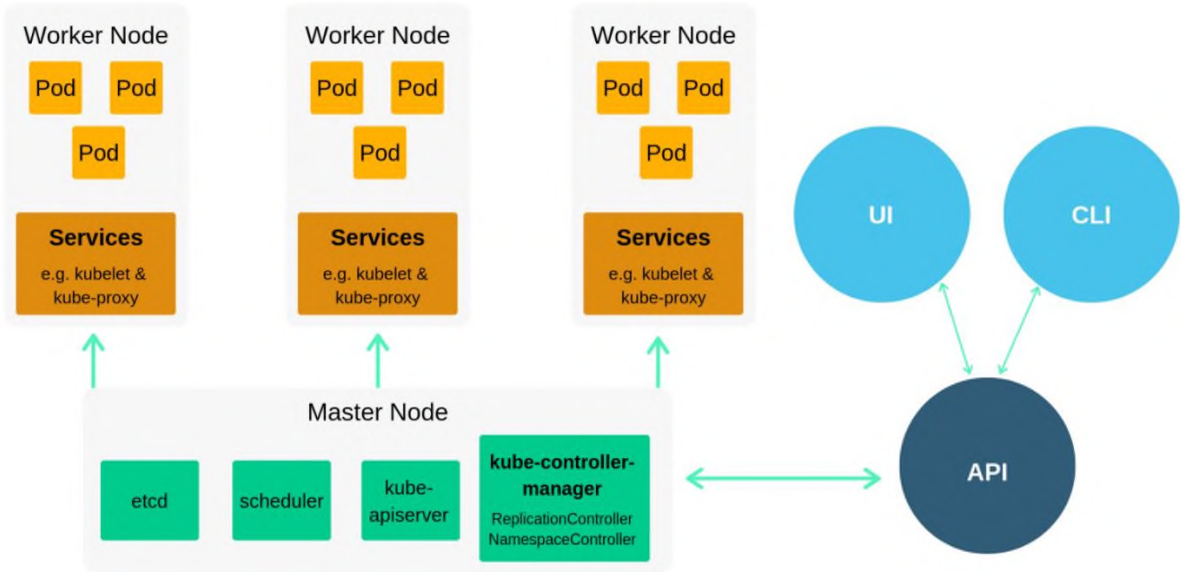
•Платформа перезапускает отказавшие контейнеры или блокирует к ним доступ.

•Конфиденциальная информация хранится изолированно от других данных.

•Развертывание или откат изменений происходит по заданному сценарию, без участия пользователя. То же относится к резервированию аппаратных ресурсов, количеству процессорных ядер, оперативной памяти (на каждый отдельно взятый контейнер). Единственное ограничение Kubernetes заключается в отсутствие встроенных компонентов вроде базы данных, модуля обработки данных и организации кэша.

Основные компоненты архитектуры

Несмотря на удобство и простоту фреймворка Kubernetes, перед его использованием, при разработке или развертывании приложений, необходимо разобраться в архитектуре. Например, понять, как соединяются между собой контейнеры (через интерфейс API), и узнать, почему компоненты разделены на две основные группы – Master Node и Worker Node.



Базовые компоненты:

- Nodes – виртуальная (физическая) машина, на мощностях которой запущены контейнеры.
- Pods – базовые модули управления приложениями, состоящие из одного или нескольких контейнеров.

- Volume – ресурс, позволяющий одновременно запускать несколько контейнеров.
- Kube-Proxy – комплекс из прокси-сервера и модуля балансировки нагрузки, позволяющий маршрутизировать входящий трафик под конкретный контейнер Pods.
- Kubelet – транслятор статусов Pods на узле и контроллер корректности работы контейнера и образа в целом.

Перечисленные компоненты устанавливаются автоматически при помощи сторонних инструментов или вручную, по отдельности. Они составляют модуль под названием Master Node, где собраны все управляющие и контролирующие функции. Через API они связываются с контейнерами, считывают их показатели, дают команду на запуск, штатную остановку или принудительное закрытие.

Процесс установки Kubernetes

При выборе автоматической установки вникать в детали не понадобится, но требуется выделить достаточное количество системных ресурсов, чтобы платформа работала бесперебойно. Например, при небольшом количестве контейнеров и простой взаимосвязи достаточно 1-2 процессорных ядер, 2-4 Гб оперативки и двух виртуальных машин, выполняющих функции Master и Worker Node.

Инсталляция на Ubuntu

Проще всего воспользоваться одной из готовых реализаций – Minikube или Kubespray. Если нужно установить Kubernetes на сервер с операционной системой Ubuntu вручную, понадобятся права суперпользователя. Начнем с установки Docker для узла. Перечень команд для этого выглядит следующим образом:

```
apt-get update
apt-get install -y docker.io
```

При необходимости организовать создание контейнеров более новых версий перечень команд будет несколько иным:

```
apt-get update
apt-get install -y \
apt-transport-https \
ca-certificates \
curl \
software-properties-common
```

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add –

*add-apt-repository *

*"deb [arch=amd64] https://download.docker.com/linux/$(. /etc/os-release; echo "$ID") *

*$(lsb_release -cs) *

stable"

apt-get update

apt-get install -y docker-ce docker-ce-cli containerd.io

Docker для одного узла установлен. Следующий шаг – установка модулей kubeadm (создание и настройка кластеров), kubelet (их запуск на хостах), kubectl (настройка компонентов, входящих в кластер). Для этого вводятся следующие команды:

apt-get update && apt-get install -y apt-transport-https software-properties-common

curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

add-apt-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"

apt-get update

apt-get install -y kubelet kubeadm kubectl

systemctl enable kubelet && systemctl start kubelet

Инсталляция на CentOS

При установке на операционную систему CentOS любой версии нужно выполнить ряд требований. Так, понадобится минимум 3 машины (1 главный, 2 рабочих узла), которые подключены к общей сети или интернету. Здесь также вся работа проводится в учетной записи sudo или root. Процедура проводится, как и в случае с Ubuntu, через консоль.

Команды для установки Docker:

yum install -y docker

systemctl enable docker && systemctl start docker

Компоненты Kubernetes (kubeadm, kubelet, kubectl) инсталлируются так:

cat <<EOF > /etc/yum.repos.d/kubernetes.repo

[kubernetes]

name=Kubernetes

baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64

enabled=1

gpgcheck=1

repo_gpgcheck=1

gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg

EOF

setenforce 0

yum install -y kubelet kubeadm kubectl

systemctl enable kubelet && systemctl start kubelet

После ввода команд (при отсутствии ошибок) можно приступать к настройке платформы. Процесс включает инициализацию кластера, создание CNI, добавление узлов Nodes и получение токена для авторизации.

Настройка Kubernetes

Сначала указывается сервер, на который был инсталлирован Kubernetes. Он становится первичным, где будут запускаться остальные операции. Инициализация выполняется при помощи команды:

kubeadm init --pod-network-cidr=10.244.0.0/16

В результате создается адрес виртуальной сети Pods (цифры выбираются по желанию пользователя). По умолчанию используется указанный IP. При правильной обработке команды на дисплее будет отображаться команда для присоединения остальных Nods-кластеров к первичному серверу, чтобы создать в итоге рабочую систему.

Следующие команды задают пользователя, который будет управлять работой комплекса:

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

Теперь можно настраивать Container Network Interface (CNI, сетевой интерфейс контейнера). Чтобы избавить себя от рутины ручного ввода команд, рекомендуется установить плагин Flannel или ему подобный (Weave Net, Calico). Первый устанавливается так:

kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

После ввода команды на экране отображаются имена всех созданных ресурсов.

Теперь пользователь добавляет Nods в существующий кластер. Для этого требуется подключение к серверу через SSH, установка модулей Docker, Kubelet, Kubeadm (вопрос рассматривался выше). Выполнение команды:

kubeadm join --token <token> <control-plane-host>:<control-plane-port> --discovery-token-ca-cert-hash sha256:<hash>

Остается получить токен авторизации кластера. Если подключение SSH еще не прервано, повторно заходить на сервер не нужно. Токен выдается после ввода команды:

kubeadm token list

По умолчанию он действует 24 часа. Если поставлена задача добавить новый узел по завершении периода, новый создается командой:

kubeadm token create

Вывод выглядит примерно так:

5didvk.d09sbcov8ph2amjw.

2. Работа с Kubectl

Что такое Kubectl?

Kubectl — это инструмент командной строки для управления кластерами Kubernetes. kubectl ищет файл config в директории $HOME/.kube. Вы можете указать другие файлы kubeconfig, установив переменную окружения KUBECONFIG или флаг --kubeconfig.

Синтаксис

Используйте следующий синтаксис для выполнения команд kubectl в терминале:

kubectl [command] [TYPE] [NAME] [flags]

где command, TYPE, NAME и flags:

- command: определяет выполняемую операцию с одним или несколькими ресурсами, например, create, get, describe, delete.
- TYPE: определяет тип ресурса. Типы ресурсов не чувствительны к регистру, кроме этого вы можете использовать единственную, множественную или сокращенную форму. Например, следующие команды выведут одно и то же.

```shell
kubectl get pod pod1
kubectl get pods pod1
kubectl get po pod1
```

- NAME: определяет имя ресурса. Имена чувствительны к регистру. Если имя не указано, то отображаются подробности по всем ресурсам, например, kubectl get pods.

При выполнении операции с несколькими ресурсами можно выбрать каждый ресурс по типу и имени, либо сделать это в одном или нескольких файлов:

Выбор ресурсов по типу и имени:

Сгруппировать ресурсы, если все они одного типа: *TYPE1 name1 name2 name<#>.*

Пример: *kubectl get pod example-pod1 example-pod2*

Выбор нескольких типов ресурсов по отдельности: *TYPE1/name1 TYPE1/name2 TYPE2/name3 TYPE<#>/name<#>.*

Пример: *kubectl get pod/example-pod1 replicationcontroller/example-rc1*

Выбор ресурсов по одному или нескольким файлов: *-f file1 -f file2 -f file<#>*

Используйте YAML вместо JSON, так как YAML удобнее для пользователей, особенно в конфигурационных файлах.

Пример: *kubectl get pod -f ./pod.yaml*

- flags: определяет дополнительные флаги. Например, вы можете использовать флаги -s или --server, чтобы указать адрес и порт API-сервера Kubernetes.

Внимание: Указанные вами флаги из командной строки переопределят значения по умолчанию и связанные переменные окружения.

Если вам нужна помощь, выполните команду *kubectl help.*

Операции

В следующей таблице приведены краткие описания и общий синтаксис всех операций kubectl:

Операция	Синтаксис	Описание
annotate	kubectl annotate (-f FILENAME | TYPE NAME | TYPE/NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--overwrite] [--all] [--resource-version=version] [flags]	Добавить или обновить аннотации одного или нескольких ресурсов.
api-versions	kubectl api-versions [flags]	Вывести доступные версии API.
apply	kubectl apply -f FILENAME [flags]	Внести изменения в конфигурацию ресурса из файла или потока stdin.
attach	kubectl attach POD -c CONTAINER [-i] [-t] [flags]	Подключиться к запущенному контейнеру либо для просмотра потока вывода, либо для работы с контейнером (stdin).
autoscale	kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min=MINPODS] --max=MAXPODS [--cpu-percent=CPU] [flags]	Автоматически промасштабировать набор подов, управляемых контроллером репликации.
cluster-info	kubectl cluster-info [flags]	Показать информацию о главном узле и сервисах в кластере.
config	kubectl config SUBCOMMAND [flags]	Изменить файлы kubeconfig. Подробные сведения смотрите в отдельных подкомандах.
create	kubectl create -f FILENAME [flags]	Создать один или несколько ресурсов из файла или stdin.
delete	kubectl delete (-f FILENAME | TYPE [NAME | /NAME | -l label | --all]) [flags]	Удалить ресурсы из файла, потока stdin, либо с помощью селекторов меток, имен, селекторов ресурсов или ресурсов.
describe	kubectl describe (-f FILENAME | TYPE [NAME_PREFIX | /NAME | -l label]) [flags]	Показать подробное состояние одного или нескольких ресурсов.

Операция	Синтаксис	Описание
diff	kubectl diff -f FILENAME [flags]	Diff file or stdin against live configuration (**BETA**)
edit	kubectl edit (-f FILENAME | TYPE NAME | TYPE/NAME) [flags]	Отредактировать и обновить определение одного или нескольких ресурсов на сервере, используя редактор по умолчанию.
exec	kubectl exec POD [-c CONTAINER] [-i] [-t] [flags] [-- COMMAND [args...]]	Выполнить команду в контейнере пода.
explain	kubectl explain [--recursive=false] [flags]	Посмотреть документацию по ресурсам. Например, поды, узлы, сервисы и т.д.
expose	kubectl expose (-f FILENAME | TYPE NAME | TYPE/NAME) [--port=port] [--protocol=TCP|UDP] [--target-port=number-or-name] [--name=name] [--external-ip=external-ip-of-service] [--type=type] [flags]	Создать Kubernetes-сервис из контроллера репликации, сервиса или пода.
get	kubectl get (-f FILENAME | TYPE [NAME | /NAME | -l label]) [--watch] [--sort-by=FIELD] [[-o | --output]=OUTPUT_FORMAT] [flags]	Вывести один или несколько ресурсов.
label	kubectl label (-f FILENAME | TYPE NAME | TYPE/NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--overwrite] [--all] [--resource-version=version] [flags]	Добавить или обновить метки для одного или нескольких ресурсов.
logs	kubectl logs POD [-c CONTAINER] [--follow] [flags]	Вывести логи контейнера в поде.
patch	kubectl patch (-f FILENAME | TYPE NAME | TYPE/NAME) --patch PATCH [flags]	Обновить один или несколько полей ресурса, используя стратегию слияния патча.
port-forward	kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT [...[LOCAL_PORT_N:]REMOTE_PORT_N] [flags]	Переадресовать один или несколько локальных портов в под.
proxy	kubectl proxy [--port=PORT] [--www=static-dir] [--www-prefix=prefix] [--api-prefix=prefix] [flags]	Запустить прокси для API Kubernetes.
replace	kubectl replace -f FILENAME	Заменить ресурс из файла или потока stdin.
rolling-update	kubectl rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] --image=NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC) [flags]	Выполните плавающее обновление, постепенно заменяя указанный контроллер репликации и его поды.
run	kubectl run NAME --image=image [--env="key=value"] [--port=port] [--replicas=replicas] [--dry-run=bool] [--overrides=inline-json] [flags]	Запустить указанный образ в кластере.
scale	kubectl scale (-f FILENAME | TYPE NAME | TYPE/NAME) --replicas=COUNT [--resource-	Обновить размер указанного контроллера репликации.

Операция	Синтаксис	Описание
	version=version] [--current-replicas=count] [flags]	
version	kubectl version [--client] [flags]	Отобразить версию Kubernetes, запущенного на клиенте и сервере.

Примечание: подробную информацию о командных операциях смотрите в справочную документацию kubectl.

Типы ресурсов

В следующей таблице перечислены все доступные типы ресурсов вместе с сокращенными аббревиатурами.

Resource Name	Short Names	API Group	Namespaced	Resource Kind
bindings			true	Binding
componentstatuses	cs		false	ComponentStatus
configmaps	cm		true	ConfigMap
endpoints	ep		true	Endpoints
limitranges	limits		true	LimitRange
namespaces	ns		false	Namespace
nodes	no		false	Node
persistentvolumeclaims	pvc		true	PersistentVolumeClaim
persistentvolumes	pv		false	PersistentVolume
pods	po		true	Pod
podtemplates			true	PodTemplate
replicationcontrollers	rc		true	ReplicationController
resourcequotas	quota		true	ResourceQuota
secrets			true	Secret
serviceaccounts	sa		true	ServiceAccount
services	svc		true	Service
mutatingwebhookconfigurations		admissionregistration.k8s.io	false	MutatingWebhookConfiguration
validatingwebhookconfigurations		admissionregistration.k8s.io	false	ValidatingWebhookConfiguration
customresourcedefinitions	crd, crds	apiextensions.k8s.io	false	CustomResourceDefinition
apiservices		apiregistration.k8s.io	false	APIService
controllerrevisions		apps	true	ControllerRevision
daemonsets	ds	apps	true	DaemonSet
deployments	deploy	apps	true	Deployment
replicasets	rs	apps	true	ReplicaSet
statefulsets	sts	apps	true	StatefulSet
tokenreviews		authentication.k8s.io	false	TokenReview
localsubjectaccessreviews		authorization.k8s.io	true	LocalSubjectAccessReview

Resource Name	Short Names	API Group	Namespaced	Resource Kind
selfsubjectaccessreviews		authorization.k8s.io	false	SelfSubjectAccessReview
selfsubjectrulesreviews		authorization.k8s.io	false	SelfSubjectRulesReview
subjectaccessreviews		authorization.k8s.io	false	SubjectAccessReview
horizontalpodautoscalers	hpa	autoscaling	true	HorizontalPodAutoscaler
cronjobs	cj	batch	true	CronJob
jobs		batch	true	Job
certificatesigningrequests	csr	certificates.k8s.io	false	CertificateSigningRequest
leases		coordination.k8s.io	true	Lease
events	ev	events.k8s.io	true	Event
ingresses	ing	extensions	true	Ingress
networkpolicies	netpol	networking.k8s.io	true	NetworkPolicy
poddisruptionbudgets	pdb	policy	true	PodDisruptionBudget
podsecuritypolicies	psp	policy	false	PodSecurityPolicy
clusterrolebindings		rbac.authorization.k8s.io	false	ClusterRoleBinding
clusterroles		rbac.authorization.k8s.io	false	ClusterRole
rolebindings		rbac.authorization.k8s.io	true	RoleBinding
roles		rbac.authorization.k8s.io	true	Role
priorityclasses	pc	scheduling.k8s.io	false	PriorityClass
csidrivers		storage.k8s.io	false	CSIDriver
csinodes		storage.k8s.io	false	CSINode
storageclasses	sc	storage.k8s.io	false	StorageClass
volumeattachments		storage.k8s.io	false	VolumeAttachment

Опции вывода

В следующих разделах рассматривается форматирование и сортировка вывода определенных команд. Дополнительные сведения о том, какие команды поддерживают разные варианты вывода, смотрите в справочной документации kubectl.

Форматирование вывода

Стандартный формат вывода всех команд kubectl представлен в понятном для человека текстовом формате. Чтобы вывести подробности в определенном формате можно добавить флаги -o или --output к команде kubectl.

Синтаксис

kubectl [command] [TYPE] [NAME] -o <output_format>

В зависимости от операции kubectl поддерживаются следующие форматы вывода:

Выходной формат	**Описание**
-o custom-columns=<spec>	Вывести таблицу с использованием списка пользовательских столбцов, разделённого запятыми.
-o custom-columns-file=<filename>	Вывести таблицу с использованием шаблона с пользовательскими столбцами в файле <filename>.
-o json	Вывести API-объект в формате JSON.
-o jsonpath=<template>	Вывести поля, определенные в выражении jsonpath.
-o jsonpath-file=<filename>	Вывести поля, определённые в выражении jsonpath из файла <filename>.
-o name	Вывести только имя ресурса.
-o wide	Вывести в текстовом формате с дополнительной информацией. Для подов отображается имя узла.
-o yaml	Вывести API-объект в формате YAML

Пример

В данном примере следующая команда выводит подробную информацию по указанному поду в виде объекта в YAML-формате:

```
kubectl get pod web-pod-13je7 -o yaml
```

Примечание: подробную информацию о доступных форматах вывода в определенной команде смотрите в справочной документации kubectl.

Пользовательские столбцы

Для определения пользовательских столбцов и вывода в таблицу только нужных данных, можно использовать опцию custom-columns. Вы можете определить пользовательские столбцы в самой опции, либо сделать это в файле шаблона: -o custom-columns=<spec> или -o custom-columns-file=<filename>.

Примеры

Столбцы указаны в самой команде:

kubectl get pods <pod-name> -o custom-columns=NAME:.metadata.name,RSRC:.metadata.resourceVersion

Столбцы указаны в файле шаблона:

kubectl get pods <pod-name> -o custom-columns-file=template.txt

где файл template.txt содержит следующее:

NAME RSRC

metadata.name metadata.resourceVersion

Результат выполнения любой из показанной выше команды:

NAME RSRC

submit-queue 610995

Получение вывода с сервера

kubectl может получать информацию об объектах с сервера. Это означает, что для любого указанного ресурса сервер вернет столбцы и строки по этому ресурсу, которые отобразит клиент. Благодаря тому, что сервер инкапсулирует реализацию вывода, гарантируется единообразный и понятный для человека вывод на всех клиентах, использующих один и тот же кластер.

Эта функциональность включена по умолчанию, начиная с kubectl 1.11 и выше. Чтобы отключить ее, добавьте флаг --server-print=false в команду kubectl get.

Примеры

Для вывода информации о состоянии пода, используйте следующую команду:

kubectl get pods <pod-name> --server-print=false

Вывод будет выглядеть следующим образом:

NAME READY STATUS RESTARTS AGE

pod-name 1/1 Running 0 1m

Сортировка списка объектов

Для вывода объектов в виде отсортированного списка в терминал используется флаг --sort-by к команде kubectl. Для сортировки объектов нужно указать любое числовое или строковое поле в флаге --sort-by. Для определения поля используйте выражение jsonpath.

Синтаксис

kubectl [command] [TYPE] [NAME] --sort-by=<jsonpath_exp>

Пример

Чтобы вывести список подов, отсортированных по имени, выполните команду ниже:

kubectl get pods --sort-by=.metadata.name

Примеры: распространенные операции

Посмотрите следующие примеры, чтобы ознакомиться с часто используемыми операциями kubectl:

kubectl apply - Внести изменения или обновить ресурс из файла или потока stdin.

 # Создать сервис из определения в example-service.yaml.

kubectl apply -f example-service.yaml

 # Создать контроллер репликации из определения в example-controller.yaml.

kubectl apply -f example-controller.yaml

 # Создать объекты, которые определены в файлах с расширением .yaml, .yml или .json в директории <directory>.

kubectl apply -f <directory>

kubectl get - Вывести один или несколько ресурсов.

 # Вывести все поды в текстовом формате вывода.

kubectl get pods

 # Вывести все поды в текстовом формате вывода и включить дополнительную информацию (например, имя узла).

kubectl get pods -o wide

 # Вывести контроллер репликации с указанным именем в текстовом формате вывода. Совет: вы можете использовать сокращенный псевдоним 'rc' вместо 'replicationcontroller'.

kubectl get replicationcontroller <rc-name>

 # Вывести все контроллеры репликации и сервисы вместе в текстовом формате вывода.

kubectl get rc,services

 # Вывести все наборы демонов в текстовом формате вывода.

kubectl get ds

 # Вывести все поды, запущенные на узле server01

kubectl get pods --field-selector=spec.nodeName=server01

kubectl describe - Показать подробное состояние одного или нескольких ресурсов, по умолчанию также включаются неинициализированные ресурсы.

 # Показать информацию об узле с именем <node-name>.

kubectl describe nodes <node-name>

 # Показать подробности пода <pod-name>.

kubectl describe pods/<pod-name>

 # Показать подробности всех подов, управляемые контроллером репликации <rc-name>.

 # Обратите внимание: любые поды, созданные контроллером репликации, имеют префикс с именем контроллера репликации.

kubectl describe pods <rc-name>

 # Показать подробности по всем подам

kubectl describe pods

Заметка: как правило, команда kubectl get используется для получения одного или нескольких ресурсов одного и того же типа. Она поддерживает большой набор флагов, с помощью которых можно настроить формат вывода, например, с помощью флага -o или --output. Вы можете указать флаг -w или --watch, чтобы отслеживать изменения в конкретном объекте. Команда kubectl describe в основном сфокусирована на описание многих взаимосвязанных аспектов указанного ресурса. При генерации вывода для пользователя она может обращаться к API-серверу. К примеру, команда kubectl describe node выдает не только информацию об узле, но и краткий обзор запущенных на нем подов, генерируемых событий и т.д.

kubectl delete - Удалить ресурсы из файла, потока stdin или с помощью селекторов меток, имена, селекторов ресурсов или имен ресурсов.

 # Удалить под по типу и имени, указанных в файле pod.yaml.

kubectl delete -f pod.yaml

 # Удалить все поды и сервисы с именем метки <label-name>.

kubectl delete pods,services -l name=<label-name>

 # Удалить все поды, включая неинициализированные.

kubectl delete pods --all

kubectl exec - Выполнить команду в контейнере пода.

 # Получить вывод от запущенной команды 'date' в поде <pod-name>. По умолчанию отображается вывод из первого контейнера.

kubectl exec <pod-name> date

 # Получить вывод из запущенной команды 'date' в контейнере <container-name> пода <pod-name>.

kubectl exec <pod-name> -c <container-name> date

 # Получить интерактивный терминал (TTY) и запустить /bin/bash в поде <pod-name>. По умолчанию отображается вывод из первого контейнера.

kubectl exec -ti <pod-name> /bin/bash

kubectl logs - Вывести логи контейнера в поде.

 # Возвращает текущие логи в поде <pod-name>.

kubectl logs <pod-name>

 # Вывод логов в поде <pod-name> в режиме реального времени. Это похоже на команду 'tail -f' Linux.

kubectl logs -f <pod-name>

Примеры: создание и использование плагинов

Посмотрите следующие примеры, чтобы ознакомиться с тем, как писать и использовать плагины kubectl:

Плагин может быть на на любом языке, а сам исполняемый файл должен начинается с префикса "kubectl-".

cat ./kubectl-hello

#!/bin/bash

Этот плагин выводит строку "hello world"

echo "hello world"

Сделать плагин исполняемым

sudo chmod +x ./kubectl-hello

Переместить его в директорию из PATH

sudo mv ./kubectl-hello /usr/local/bin

Плагин дял kubectl создан и "установлен".

Воспользоваться плагином можно через kubectl, вызвав его подобно обычной команды.

kubectl hello

hello world

"Отмена установки" плагина происходит через удаление его файла из директории в PATH.

sudo rm /usr/local/bin/kubectl-hello

Посмотреть все доступные плагины kubectl можно с помощью подкоманды kubectl plugin list:

kubectl plugin list

The following kubectl-compatible plugins are available:

/usr/local/bin/kubectl-hello

/usr/local/bin/kubectl-foo

/usr/local/bin/kubectl-bar

Эта команда также может сообщить, что плагин является неисполняемым,

либо что плагин переопределен другими плагинами

sudo chmod -x /usr/local/bin/kubectl-foo

kubectl plugin list

The following kubectl-compatible plugins are available:

/usr/local/bin/kubectl-hello

/usr/local/bin/kubectl-foo

- warning: /usr/local/bin/kubectl-foo identified as a plugin, but it is not executable

/usr/local/bin/kubectl-bar

error: one plugin warning was found

Плагины можно рассматривать как способ создания более сложной функциональности поверх существующих команд kubectl:

cat ./kubectl-whoami

#!/bin/bash

Этот плагин использует команду `kubectl config` для вывода

информации о текущем пользователе из текущего выбранного контекста

kubectl config view --template='{{ range .contexts }}{{ if eq .name "'$(kubectl config current-context)'" }}Current user: {{ .context.user }}{{ end }}{{ end }}'

Выполнение этого плагина генерирует вывод, содержащий пользователя для текущего выбранного контекста в файле KUBECONFIG:

Сделать файл исполняемым

sudo chmod +x ./kubectl-whoami

Перенести файл в директорию из PATH

sudo mv ./kubectl-whoami /usr/local/bin

kubectl whoami

Current user: plugins-user.

3. Инструменты для упрощения написания конфигурационных файлов. Helm и Jsonnet

Один из популярных диспетчеров пакетов для Kubernetes называется Helm. Вы можете использовать утилиту командной строки helm для установки и конфигурации приложений (собственных или чужих) и создавать пакеты, которые полностью описывают все необходимые для работы приложения ресурсы, включая их зависимости и настройки. Пакеты в Helm называются чартами (charts).

Система Helm входит в семейство проектов Cloud Native Computing Foundation, что свидетельствует о ее стабильности и широком использовании.

Важно понимать, что Helm chart, в отличие от двоичных пакетов, которые используются такими инструментами, как APT или Yum, на самом деле не включает в себя образ контейнера как таковой. Вместо этого он содержит лишь метаданные о том, где образ можно найти, — точно так же, как при развертывании в Kubernetes.

Когда вы устанавливаете чарт, Kubernetes самостоятельно находит и загружает двоичный образ контейнера по указанному адресу. На самом деле это просто удобная обертка вокруг YAML-манифестов в Kubernetes.

Установка Helm

Следуйте инструкциям по установке Helm (helm.sh/docs/using_helm/#installing-helm) для вашей операционной системы.

Установив Helm, вы должны создать определенные ресурсы Kubernetes, чтобы открыть этому диспетчеру пакетов доступ к вашему кластеру. В каталоге hellohelm репозитория с примерами находится подходящий YAML-файл под названием helm-auth.yaml. Чтобы его применить, выполните следующие команды:

cd ../hello-helm

kubectl apply -f helm-auth.yaml

serviceaccount "tiller" created

clusterrolebinding.rbac.authorization.k8s.io "tiller" created

Установив необходимые права доступа, вы можете воспользоваться такой командой, чтобы инициализировать Helm для работы с кластером:

helm init --service-account tiller

$HELM_HOME has been configured at /Users/john/.helm.

Инициализация Helm может занять около пяти минут. Убедитесь в том, что все работает:

helm version

Client: &version.Version{SemVer:"v2.9.1",

GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}

Server: &version.Version{SemVer:"v2.9.1",

GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}

Если эта команда выполнилась успешно, вы готовы к работе с Helm. Если же появилось сообщение Error: cannot connect to Tiller, подождите несколько минут и попробуйте снова.

Установка чарта Helm

Как будет выглядеть чарт Helm для нашего приложения? В каталоге hello-helm можно заметить подкаталог k8s, который в предыдущем примере (hello-k8s) содержал только файлы манифеста Kubernetes для развертывания нашего кода.

Теперь внутри него есть папка demo с чартом Helm внутри:

ls k8s/demo

Chart.yaml prod-values.yaml staging-values.yaml templates values.yaml

Для начала очистите ресурсы, оставшиеся от каких-либо предыдущих развертываний:

kubectl delete all --selector app=demo

Затем запустите следующую команду:

helm install --name demo ./k8s/demo

NAME: demo

LAST DEPLOYED: Wed Jun 6 10:48:50 2018

NAMESPACE: default

STATUS: DEPLOYED

RESOURCES:

==> v1/Service

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

demo-service ClusterIP 10.98.231.112 <none> 80/TCP 0s

==> v1/Deployment

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

demo 1 1 1 0 0s

==> v1/Pod(related)

NAME READY STATUS RESTARTS AGE

demo-6c96484c48-69vss 0/1 ContainerCreating 0 0s

Вы видите, что диспетчер Helm создал ресурс Deployment (который запускает подоболочку) и Service. Для этого команда helm install создает объект Kubernetes под названием «выпуск» (release).

Чарты, репозитории и выпуски

Это самые важные термины, относящиеся к Helm, которые вам нужно знать.

- Чарт — пакет Helm с определениями всех ресурсов, необходимых для выполнения приложения в Kubernetes.
- Репозиторий — место, где можно делиться своими чартами и загружать чужие.
- Выпуск — конкретный экземпляр чарта, запущенный в кластере Kubernetes.

Один чарт можно устанавливать несколько раз в одном и том же кластере. Например, у вас может быть запущено несколько копий чарта с веб-сервером Nginx, каждая из которых обслуживает свой сайт. Один экземпляр чарта является отдельным выпуском.

У каждого выпуска есть уникальное имя, которое можно указать в команде helm install с помощью флага -name (если не указать имя, Helm выберет случайный выпуск, что вряд ли вам понравится).

Вывод списка выпусков Helm

Чтобы проверить, какие выпуски запущены на данный момент, выполните команду helm list:

helm list

NAME REVISION UPDATED STATUS CHART NAMESPACE

demo 1 Wed Jun 6 10:48:50 2018 DEPLOYED demo-1.0.1 default

Чтобы вывести состояние конкретного выпуска, передайте его имя команде helm status. Вы увидите ту же информацию, что и при первом развертывании выпуска:

- список всех ресурсов Kubernetes, которые с ним связаны, включая развертывания,
- pod-оболочки и сервисы.

Полный список публичных чартов Helm (github.com/helm/charts/tree/master/stable) можно просмотреть в GitHub.

Список доступных чартов также можно получить, выполнив команду helm search без аргументов (или helm search redis, если вам, к примеру, нужно найти чарт для Redis).

Построение манифестов с помощью Helm

Развертывать и администрировать приложения возможно с помощью ресурсов Kubernetes, созданных из манифестов в формате YAML. Ничто не мешает вам управлять приложениями исключительно посредством необработанных (сырых) файлов YAML, однако это не лучший способ. Помимо сложности обращения со всеми этими файлами, встает еще и вопрос дистрибуции.

Предположим, вы хотите, чтобы другие люди могли запускать ваше приложение в своих собственных кластерах. Вы можете передать им манифесты файлов, однако новым пользователям неизбежно придется подгонять некоторые параметры под свою среду.

Для этого потребуется создать копию конфигурации Kubernetes, найти, где определяются различные настройки (которые могут дублироваться в разных местах), и отредактировать их.

Со временем пользователи должны будут сохранять уже свои копии файлов. Но когда вы выпустите обновление, его придется загружать и вручную объединять с локальными изменениями. Рано или поздно такие действия станут утомительными. Поэтому нужна возможность отделять необработанные файлы манифестов от определенных параметров и переменных, которые могут потребовать модификации с вашей стороны или со стороны любых других пользователей. В идеале все это можно было бы сделать доступным в стандартном формате, чтобы кто угодно мог загрузить и установить ваше приложение в кластере Kubernetes.

Тогда каждое приложение сможет предоставлять не только конфигурационные значения, но и свои зависимости от других программ и сервисов, а умный инструмент для управления пакетами установит и запустит приложения вместе со всеми их зависимостями, используя лишь одну

команду. Взглянем на чарты Helm немного поближе и посмотрим, как они создаются.

Что внутри у чарта Helm

Откройте папку hello-helm/k8s в репозитории demo: в ней вы увидите, что хранится внутри чарта Helm.

Каждый чарт имеет стандартную структуру. Прежде всего, он находится внутри каталога с таким же именем (в данном случае demo):

```
demo
├── Chart.yaml
├── production-values.yaml
├── staging-values.yaml
├── templates
│ ├── deployment.yaml
│ └── service.yaml
└── values.yaml
```

Файл Chart.yaml

Кроме того, он содержит файл Chart.yaml, который определяет имя и версию чарта:

```
name: demo
sources:
 - https://github.com/cloudnativedevops/demo
version: 1.0.1
```

В Chart.yaml можно указать множество дополнительных полей — в том числе и ссылку на исходный код проекта, как здесь показано, — но обязательной информацией являются лишь имя и версия.

Файл values.yaml

Есть также файл под названием values.yaml, содержащий параметры, которые автор чарта позволяет менять пользователям:

```
environment: development
container:
 name: demo
 port: 8888
 image: cloudnatived/demo
```

tag: hello

replicas: 1

Это чем-то напоминает файл YAML манифеста Kubernetes, но с одним важным отличием: файл values.yaml ограничен лишь форматом YAML и не имеет никакой предопределенной структуры. То, какие переменные устанавливать, какие имена и значения им присваивать, решаете вы сами.

В чарте может вообще не быть никаких переменных, но если таковые имеются, поместите их в values.yaml и затем ссылайтесь на них с других участков вашего чарта.

Пока не обращайте внимания на файлы production-values.yaml и stagingvalues.yaml.

Шаблоны Helm

Так где же упоминаются эти переменные? Если заглянуть в папку templates, можно увидеть несколько знакомых файлов:

ls k8s/demo/templates

deployment.yaml service.yaml

Вместо того чтобы напрямую что-то упоминать — например, имя контейнера, — шаблоны содержат заглушки, которые Helm заменит настоящими значениями из values.yaml.

Вот как выглядит шаблонное развертывание:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: {{ .Values.container.name }}-{{ .Values.environment }}
spec:
 replicas: {{ .Values.replicas }}
 selector:
 matchLabels:
 app: {{ .Values.container.name }}
 template:
 metadata:
 labels:
 app: {{ .Values.container.name }}
 environment: {{ .Values.environment }}
 spec:
```

containers:

- name: {{ .Values.container.name }}

image: {{ .Values.container.image }}:{{ .Values.container.tag }}

ports:

- containerPort: {{ .Values.container.port }}

env:

- name: ENVIRONMENT

value: {{ .Values.environment }}

Фигурные скобки указывают на место, куда диспетчер Helm должен подставить значение переменной. На самом деле это часть синтаксиса шаблонов Go. Да, от Go никуда не деться. Kubernetes и Helm сами написаны на Go, поэтому неудивительно, что в чартах Helm применяется синтаксис этого языка.

Интерполяция переменных

В данном шаблоне упоминается несколько переменных:

...

metadata:

name: {{ .Values.container.name }}-{{ .Values.environment }}

Весь фрагмент, включая фигурные скобки, будет интерполирован (то есть заменен) значениями container.name и environment, взятыми из values.yaml. Сгенерированный результат будет выглядеть так:

...

metadata:

name: demo-development

Это очень удобно, потому что такие значения, как container.name, упоминаются в шаблоне по нескольку раз. Естественно, они присутствуют и в сервисе:

apiVersion: v1

kind: Service

metadata:

name: {{ .Values.container.name }}-service-{{ .Values.environment }}

labels:

app: {{ .Values.container.name }}

spec:

ports:

- port: {{ .Values.container.port }}
protocol: TCP
targetPort: {{ .Values.container.port }}
selector:
app: {{ .Values.container.name }}
type: ClusterIP

Вы видите, сколько раз упоминается, скажем, Values.container.name: даже в простом чарте вроде этого вам необходимо многократно повторять одни и те же фрагменты информации. Благодаря переменным Helm подобного дублирования можно избежать. Например, чтобы поменять имя контейнера, достаточно отредактировать values.yaml и заново установить свой чарт: в результате изменение распространится на все шаблоны.

Шаблоны в Go имеют очень мощный синтаксис и далеко не ограничиваются простой подстановкой переменных, как в нашем примере: они поддерживают циклы, выражения, условия и даже вызов функций. Чарты Helm могут использовать эти возможности для создания довольно сложной конфигурации на основе вводных значений.

Больше о написании шаблонов можно почитать в документации Helm (helm.sh/docs/chart_template_guide).

Цитирование значений в шаблонах

Helm позволяет использовать функцию quote для автоматического цитирования значений в шаблонах:

name: {{.Values.MyName | quote }}

Цитировать следует только строковые значения. Не применяйте функцию quote для чисел, таких как номера портов.

Задание зависимостей

Что, если ваш чарт зависит от других чартов? Например, если приложение использует Redis, в его чарте Helm, возможно, придется указать чарт redis в качестве зависимости.

Это можно сделать в файле requirements.yaml:

dependencies:
- name: redis
version: 1.2.3
- name: nginx

version: 3.2.1

Теперь выполните команду helm dependency update: Helm загрузит чарты и подготовит их к установке вместе с вашим собственным приложением.

Продвинутые инструменты управления манифестами

Helm является отличным и широко используемым инструментом, но имеет несколько недостатков: написание и редактирование его шаблонов — не самое приятное занятие, а файлы YAML в Kubernetes довольно сложные, объемные и повторяющиеся.

Сейчас в разработке находится несколько инструментов, которые пытаются решить данные проблемы и упростить работу с манифестами Kubernetes: либо с помощью более мощных по сравнению с YAML языков описания, таких как Jsonnet, либо путем объединения манифестов в базовые шаблоны, которые затем настраиваются с помощью накладываемых файлов.

ksonnet

Иногда декларативных возможностей YAML недостаточно, особенно если речь идет о больших и сложных развертываниях, в которых нужно использовать вычисления и логику. Например, вы можете устанавливать количество реплик динамически, в зависимости от размера кластера. Для этого нужен настоящий язык программирования.

«Файлы YAML в Kubernetes никогда не предназначались для ручного редактирования. ksonnet стремится предоставить единый инструмент, который облегчает настройку приложений Kubernetes в разных кластерах и средах.» - Джо Беда (Heptio)

ksonnet (ksonnet.io) позволяет писать манифесты Kubernetes на языке под названием Jsonnet, который является расширенной версией JSON — декларативного формата данных, аналогичного YAML. Манифесты в формате JSON также поддерживаются в Kubernetes. Jsonnet расширяет возможности JSON: позволяет работать с переменными, циклами, условными выражениями, выполнять арифметические операции, обрабатывать ошибки и т. д.

```
local env = std.extVar("__ksonnet/environments");
local params = std.extVar("__ksonnet/params").components.demo;
[
 {
```

```
"apiVersion": "v1",
"kind": "Service",
"metadata": {
"name": params.name
},
"spec": {
"ports": [
{
"port": params.servicePort,
"targetPort": params.containerPort
...
```

Но самое важное то, что ksonnet вводит концепцию прототипов: заранее подготовленных наборов ресурсов Kubernetes, с помощью которых можно «штамповать» часто используемые шаблоны манифестов.

Например, один из встроенных в ksonnet прототипов — deployed-service — генерирует развертывание для заданного контейнера, а также сервис, который направляет к нему трафик. Это хорошая отправная точка для большинства приложений, которые запускаются в Kubernetes.

Вам доступна библиотека публичных прототипов, но вы можете определять собственные, чтобы не дублировать большое количество кода манифестов во всех своих приложениях и сервисах.

kapitan

kapitan (github.com/deepmind/kapitan) — это еще один инструмент на основе Jsonnet, предназначенный для использования общих конфигурационных значений в разных приложениях или даже кластерах. У kapitan есть иерархическая база данных параметров (под названием «инвентарь»), которая позволяет повторно использовать шаблоны манифестов, подключая разные значения в зависимости от среды или приложения:

```
local kube = import "lib/kube.libjsonnet";
local kap = import "lib/kapitan.libjsonnet";
local inventory = kap.inventory();
local p = inventory.parameters;
{
 "00_namespace": kube.Namespace(p.namespace),
```

"10_serviceaccount": kube.ServiceAccount("default")

}

kustomize

kustomize (github.com/kubernetes-sigs/kustomize) — это еще один инструмент для работы с манифестами, который использует стандартный формат YAML вместо шаблонов или альтернативных языков наподобие Jsonnet. Вы начинаете с базового манифеста, на который накладываются дополнительные слои, чтобы подогнать его под разные среды или конфигурации. Утилита командной строки kustomize берет все эти файлы и генерирует из них итоговые манифесты:

namePrefix: stagingcommonLabels:

environment: staging

org: acmeCorporation

commonAnnotations:

note: Hello, I am staging!

bases:

- ../../base

patchesStrategicMerge:

- map.yaml

EOF

Благодаря этому развертывание манифестов можно свести к такой простой команде:

kustomize build /myApp/overlays/stagingq | kubectl apply -f -

Если вам не нравятся шаблоны или Jsonnet и вы хотите работать с обычными манифестами Kubernetes, этот инструмент заслуживает вашего внимания.

kompose

Скорее всего, вы знакомы с Docker Compose, если запускали промышленные сервисы в контейнерах Docker, но без использования Kubernetes.

Compose позволяет определить и развернуть группу контейнеров, работающих вместе: например, веб-сервер, серверное приложение и базу данных, такую как Redis.

То, как эти контейнеры будут взаимодействовать между собой, можно описать с помощью единственного файла docker-compose.yml.

kompose (github.com/kubernetes/kompose) — инструмент для преобразования dockercompose.yml в манифесты Kubernetes. Он позволяет мигрировать из Docker Compose без необходимости писать с нуля собственные манифесты Kubernetes или чарты Helm.

Ansible

Возможно, вы уже знакомы с Ansible, популярным инструментом для автоматизации инфраструктуры. Он не имеет непосредственного отношения к Kubernetes, но способен управлять разными видами ресурсов, используя модули расширений, такие как Puppet.

Кроме установки и настройки кластеров, Ansible умеет управлять такими ресурсами Kubernetes, как развертывания и сервисы: для этого используется модуль k8s (docs.ansible.com/ansible/latest/modules/k8s_module.html).

Как и Helm, Ansible способен шаблонизировать манифесты Kubernetes с помощью стандартного языка шаблонов (Jinja), предоставляя более развитый — иерархический — механизм поиска переменных. Например, вы можете задать общие значения для группы приложений или среды развертывания, такой как staging.

Даже если вы уже применяете Ansible в своей организации, стоит хорошо подумать, прежде чем использовать его и для работы с ресурсами Kubernetes. Для инфраструктуры, основанной исключительно на Kubernetes, Ansible может оказаться слишком мощным. Но в случае смешанной инфраструктуры вполне удобно иметь один инструмент для управления всем и вся:

```
kube_resource_configmaps:
 my-resource-env: "{{ lookup('template', template_dir +
'/my-resource-env.j2') }}"
kube_resource_manifest_files: "{{ lookup('fileglob', template_dir +
'/*manifest.yml') }}"
- hosts: "{{ application }}-{{ env }}-runner"
 roles:
 - kube-resource
```

Возможность управления Kubernetes с помощью Ansible (willthames.github.io/ansiblefest2018/#/) рассматривается в докладе Уилла Тэймса, специализирующегося на этой теме.

kubeval

В отличие от других инструментов, на которые мы обратили внимание в этом разделе, kubeval (github.com/instrumenta/kubeval) предназначен не для генерации или шаблонизации манифестов Kubernetes, а для их проверки.

У каждой версии Kubernetes есть своя схема манифестов в форматах YAML и JSON, поэтому важно иметь возможность автоматически проверять манифесты на соответствие этой схеме. Например, kubeval помогает удостовериться, все ли обязательные поля вы указали для заданного объекта и корректные ли типы у значений.

И kubectl проверяет манифесты во время их применения, выдавая сообщения об ошибках, когда вы пытаетесь применить некорректный манифест. Но возможность сделать это предварительно тоже может быть полезной. kubeval не требует доступа к кластеру и поддерживает любые версии Kubernetes. kubeval лучше использовать в рамках процесса непрерывного развертывания, чтобы манифесты проверялись всякий раз, когда вы вносите в них изменения. kubeval также позволяет, к примеру, перед обновлением до последней версии Kubernetes проверить, требуют ли ваши манифесты какой-либо модификации для работы с ней.

4. Поддержка нескольких окружений на примере Qbec

Qbec — это инструмент деплоя, который позволяет декларативно описывать манифесты вашего приложения и деплоить их в Kubernetes. Использование Jsonnet в качестве основного синтаксиса позволяет очень сильно упростить описание различий для нескольких окружений, а также почти полностью избавляет от повторяемости кода.

Это может быть особенно актуально в тех случаях, когда вам нужно задеплоить приложение в несколько кластеров с разными параметрами и вы хотите декларативно описать их в Git.

Qbec также позволяет рендерить Helm-чарты передавая им необходимые параметры и в дальнейшем оперировать ими также как и обычными манифестами, в том числе можно накладывать на них различные мутации, а это, в свою очередь, позволяет избавиться от необходимости использовать ChartMuseum. То есть можно хранить и рендерить чарты прямо из git, где им и самое место.

Все деплойменты мы будем хранить в директории deploy/:

mkdir deploy

cd deploy

Давайте инициализируем наше первое приложение:

qbec init website

cd website

Сейчас структура нашего приложения выглядит так:

```
├── components
├── environments
│   ├── base.libsonnet
│   └── default.libsonnet
├── params.libsonnet
└── qbec.yaml
```

посмотрим на файл qbec.yaml:

```
apiVersion: qbec.io/v1alpha1
kind: App
metadata:
  name: website
spec:
  environments:
```

```
  default:
    defaultNamespace: docs
    server: https://kubernetes.example.org:8443
 vars: {}
```

Здесь нас интересует в первую очередь spec.environments, qbec уже создал за нас default окружение и взял адрес сервера, а также namespace из нашего текущего kubeconfig.

Теперь при деплое в default окружение, qbec всегда будет деплоить только в указанный Kubernetes-кластер и в указанный неймспейс, то есть вам больше не придётся переключаться между контекстами и неймспейсами для того, чтобы выполнить деплой.

В случае необоходимости вы всегда можете обновить настройки в этом файле.

Все ваши окружения описываются в qbec.yaml, и в файле params.libsonnet, где сказано откуда нужно брать для них параметры.

Дальше мы видим две директории:

- components/ — здесь будут храниться все манифесты для нашего приложения, они могут быть описаны как в jsonnet так и обычными yaml-файлами
- environments/ — здесь мы будем описывать все переменные (параметры) для наших окружений.

По умолчанию мы имеем два файла:

- environments/base.libsonnet — он будет содержать общие параметры для всех окружений
- environments/default.libsonnet — содержит параметры переопределённые для окружения default

Давайте откроем environments/base.libsonnet и добавим туда параметры для нашего первого компонента:

```
{
 components: {
   website: {
     name: 'example-docs',
     image: 'registry.gitlab.com/kvaps/docs.example.org/website:v0.0.1',
     replicas: 1,
     containerPort: 80,
     servicePort: 80,
```

```
    nodeSelector: {},
    tolerations: [],
    ingressClass: 'nginx',
    domain: 'docs.example.org',
   },
 },
}
```

Создадим также наш первый компонент components/website.jsonnet:

```
local env = {
 name: std.extVar('qbec.io/env'),
 namespace: std.extVar('qbec.io/defaultNs'),
};
local p = import '../params.libsonnet';
local params = p.components.website;
[
 {
  apiVersion: 'apps/v1',
  kind: 'Deployment',
  metadata: {
   labels: { app: params.name },
   name: params.name,
  },
  spec: {
   replicas: params.replicas,
   selector: {
    matchLabels: {
     app: params.name,
    },
   },
   template: {
    metadata: {
     labels: { app: params.name },
    },
    spec: {
     containers: [
      {
```

```
          name: 'darkhttpd',
          image: params.image,
          ports: [
            {
              containerPort: params.containerPort,
            },
          ],
        },
      ],
      nodeSelector: params.nodeSelector,
      tolerations: params.tolerations,
      imagePullSecrets: [{ name: 'regsecret' }],
    },
  },
 },
},
{
 apiVersion: 'v1',
 kind: 'Service',
 metadata: {
  labels: { app: params.name },
  name: params.name,
 },
 spec: {
  selector: {
   app: params.name,
  },
  ports: [
   {
    port: params.servicePort,
    targetPort: params.containerPort,
   },
  ],
 },
},
{
```

```
  apiVersion: 'extensions/v1beta1',
  kind: 'Ingress',
  metadata: {
   annotations: {
    'kubernetes.io/ingress.class': params.ingressClass,
   },
   labels: { app: params.name },
   name: params.name,
  },
  spec: {
   rules: [
    {
     host: params.domain,
     http: {
      paths: [
       {
        backend: {
         serviceName: params.name,
         servicePort: params.servicePort,
        },
       },
      ],
     },
    },
   ],
  },
 },
]
```

В данном файле мы описали сразу три Kubernetes-сущности, это: Deployment, Service и Ingress. При желании мы могли бы вынести их в разные компоненты, но на данном этапе нам хватит и одного.

Синтаксис jsonnet очень похож на обычный json, в принципе обычный json уже является валидным jsonnet, так что первое время вам возможно будет проще воспользоваться онлайн-сервисами вроде yaml2json, чтобы сконвертировать привычный вам yaml в json, либо, если ваши компоненты не

содержат никаких переменных, то их вполне можно описать в виде обычного yaml.

При работе с jsonnet очень советую установить плагин для вашего редактора

К примеру, для vim есть плагин vim-jsonnet, который включает подсветку синтаксиса и автоматически выполняет jsonnet fmt при каждом сохранении (требует наличия установленно jsonnet).

Всё готово, теперь можем начинать деплой:

Чтобы посмотреть, что у нас получилось, выполним:

qbec show default

На выходе вы увидите отрендеренные yaml-манифесты, которые будут применены в кластер default.

Отлично, теперь применим:

qbec apply default

На выходе вы всегда увидите, что будет проделанно в вашем кластере, qbec попросит вас согласиться с изменениями, набрав y вы сможете подтвердить свои намерения.

Готово теперь наше приложение задеплоено.

В случае внесения изменений вы всегда сможете выполнить:

qbec diff default

чтобы посмотреть, как эти изменения отразятся на текущем деплое

Не забываем закоммитить наши изменения:

cd ../..

git add deploy/website

git commit -m "Add deploy for website".

5. Создание и использование секретов

Возможность отделить логику приложения Kubernetes от его конфигурации (то есть от любых значений или настроек, которые со временем могут поменяться) очень полезна. К конфигурационным значениям обычно относят параметры, предназначенные для определенной среды, DNS-адреса сторонних сервисов и учетные данные для аутентификации.

Конечно, все это можно поместить непосредственно в код, но такой подход недостаточно гибок. Например, для изменения конфигурационного значения тогда придется заново собирать и развертывать ваш код. Намного лучшим решением было бы отделить конфигурацию от кода и считывать ее из файла или переменных среды.

Kubernetes предоставляет несколько разных способов управления конфигурацией. Во-первых, вы можете передавать значения в приложение через переменные среды, указанные в спецификации pod-оболочки. Во-вторых, конфигурационные данные можно хранить непосредственно в Kubernetes, используя объекты ConfigMap и Secret.

Использование объектов Secret в качестве переменных среды

Объект Secret можно сделать доступным в контейнере в виде переменных среды или файла на его диске. В следующем примере мы присвоим переменной среды значение из Secret:

spec:
containers:
- name: demo
image: cloudnatived/demo:hello-secret-env
ports:
- containerPort: 8888
env:
- name: GREETING
valueFrom:
secretKeyRef:
name: demo-secret
key: magicWord

Мы установили переменную GREETING.

Выполните следующую команду в репозитории demo, чтобы применить манифесты:

kubectl apply -f hello-secret-env/k8s/

deployment.extensions "demo" configured

secret "demo-secret" created

Как и раньше, перенаправьте локальный порт к развертыванию, чтобы увидеть результат в своем браузере:

kubectl port-forward deploy/demo 9999:8888

Forwarding from 127.0.0.1:9999 -> 8888

Forwarding from [::1]:9999 -> 8888

При открытии адреса http://localhost:9999/ вы должны увидеть следующее:

The magic word is "xyzzy"

Запись объектов Secret в файлы

В этом примере мы подключим объект Secret к контейнеру в виде файла. Код находится в папке hello-secret-file репозитория demo.

Чтобы подключить Secret в виде файла, воспользуемся следующим развертыванием:

```
spec:
containers:
- name: demo
image: cloudnatived/demo:hello-secret-file
ports:
- containerPort: 8888
volumeMounts:
- name: demo-secret-volume
mountPath: "/secrets/"
readOnly: true
volumes:
- name: demo-secret-volume
secret:
secretName: demo-secret
```

Как и в подразделе «Создание конфигурационных файлов из объектов ConfigMap» на с. 240, мы создаем том (в данном случае это demo-secret-volume) и подключаем его к контейнеру в разделе спецификации

volumeMounts. В поле mountPath указано /secrets, поэтому Kubernetes создаст в этой папке по одному файлу для каждой пары «ключ — значение», определенной в объекте Secret.

В нашем примере мы определили только одну пару «ключ — значение» с именем magicWord, поэтому манифест создаст в контейнере один файл /secrets/magicWord с конфиденциальными данными, доступный исключительно для чтения.

Если применить этот манифест таким же образом, как и в предыдущем примере, должен получиться тот же результат:

The magic word is "xyzzy"

Чтение объектов Secret

В предыдущем разделе мы использовали команду kubectl describe для вывода содержимого ConfigMap. Можно ли то же самое сделать с Secret?

kubectl describe secret/demo-secret
Name: demo-secret
Namespace: default
Labels: <none>
Annotations:
Type: Opaque
Data
====
magicWord: 5 bytes

Обратите внимание на то, что сами данные не отображаются. Объекты Secret в Kubernetes имеют тип Opaque: это означает, что их содержимое не показывается в выводе kubectl describe, журнальных записях и терминале, благодаря чему невозможно случайно раскрыть конфиденциальную информацию.

Чтобы просмотреть закодированную версию конфиденциальных данных в формате YAML, воспользуйтесь командой kubectl get:

kubectl get secret/demo-secret -o yaml
apiVersion: v1
data:
magicWord: eHl6enk=
kind: Secret
metadata:

...

type: Opaque

base64

Что за eHl6enk=, совсем не похожий на наше исходное значение? На самом деле это объект Secret, представленный в кодировке *base64*. Base64 — это схема кодирования произвольных двоичных данных в виде строки символов.

Поскольку конфиденциальная информация может быть двоичной и недоступной для вывода (как, например, в случае с ключом шифрования TLS), объекты Secret всегда хранятся в формате base64.

Текст beHl6enk= является версией нашего секретного слова xyzzy, закодированной в base64. В этом можно убедиться, если выполнить в терминале команду base64 --decode:

echo "eHl6enk=" | base64 --decode

xyzzy

Таким образом, несмотря на то, что Kubernetes защищает вас от случайного вывода конфиденциальных данных в терминале или журнальных файлах, при наличии прав на чтение объектов Secret в определенном пространстве имен эти данные можно получить в формате base64 и впоследствии их раскодировать.

Если вам нужно закодировать в base64 какой-нибудь текст (например, чтобы поместить его в Secret), используйте команду base64 без аргументов:

echo xyzzy | base64

eHl6enkK

Доступ к объектам Secret

Кто может читать и редактировать объекты Secret? Это определяется RBAC — механизмом контроля доступа. Если вы используете кластер, в котором система RBAC отсутствует или не включена, все ваши объекты Secret доступны любым пользователям и контейнерам (позже мы объясним, что у вас не должно быть ни одного промышленного кластера без RBAC).



Модуль 19. Планирование безопасности для кластера Kubernetes

- Синхронизация секретов с внешними сервисами. Vault
- Карты конфигураций
- Сервис-аккаунты
- NetworkPolicies, SecurityContext

1. Синхронизация секретов с внешними сервисами. Vault

Vault от HashiCorp

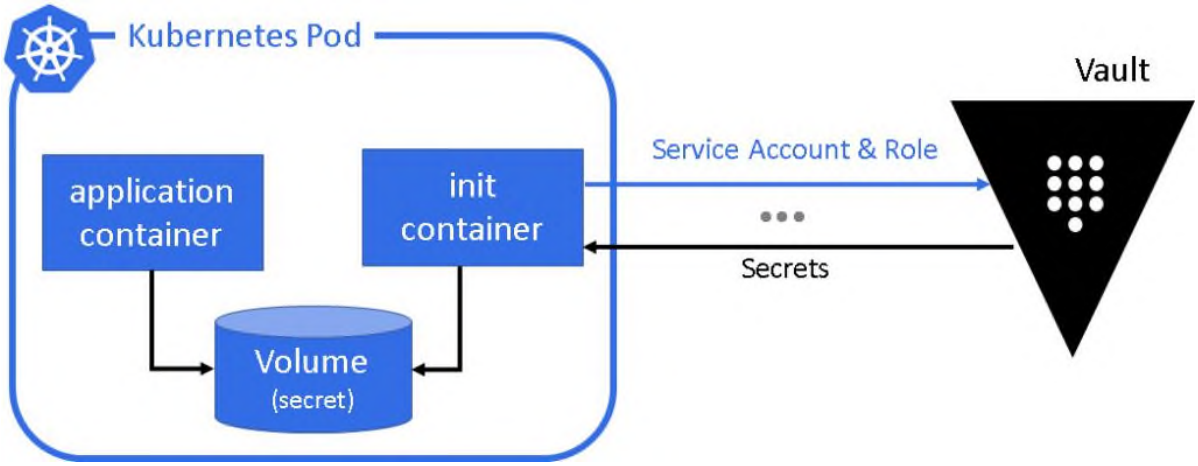
Vault — это инструмент для управления конфиденциальными данными (также известными как секреты), такими как пароли, ключи доступа и сертификаты. Vault позволяет отделить секреты от приложений. Исторически сложилось так, что конфиденциальными данными было трудно управлять и требовали ручных запросов и координации между группами разработки приложений, эксплуатации и безопасности. Vault помогает устранить большую часть этих проблем, полностью автоматизируя процессы запроса секретов и управления ими.

Помимо основ защиты данных при передаче и хранении, журналов аудита и контроля доступа, вот некоторые из расширенных функций, предлагаемых Vault:

Динамические секреты: Vault может создавать секреты точно в срок, т.е. только тогда, когда приложение запрашивает их. Это гарантирует, что секреты уникальны для использования (компонент приложения) и среды (разработка, промежуточное хранение, производство и т. Д.). Например, каждый экземпляр приложения в рабочей среде может иметь свой пароль. При удалении экземпляра приложения его пароль также может быть немедленно удален.

Аренда: Vault предоставляет возможность связать аренду с **динамическими секретами**, гарантируя, что секреты истекают и чередуются или удаляются, если в них больше нет необходимости.

При использовании с Kubernetes основная идея заключается в использовании метода проверки подлинности Kubernetes Vault для предоставления секретов для приложений Kubernetes.



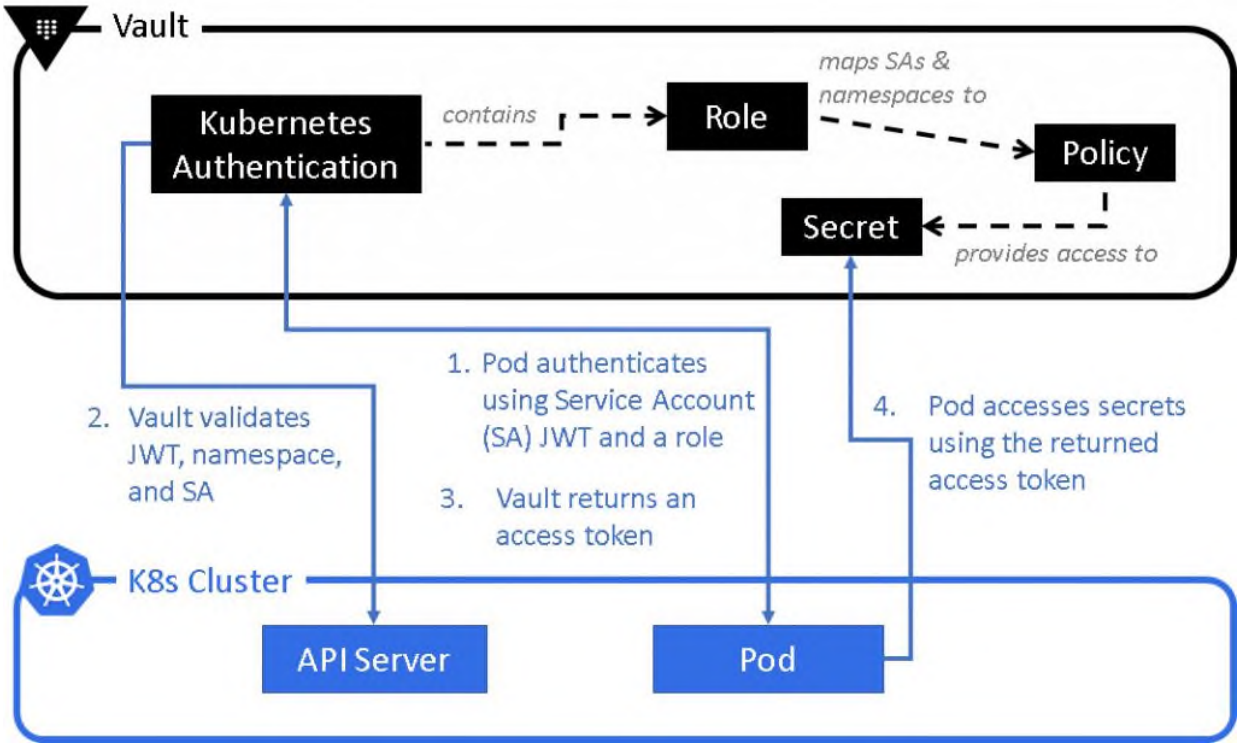
Запуск Vault

Как правило, одна централизованная служба Vault обслуживает несколько кластеров Kubernetes. Хранилище можно запускать в Kubernetes или непосредственно на виртуальных или физических серверах.

Для систем разработки и тестирования можно запустить Vault в режиме разработки или использовать соответствующий образ, предназначенный для запуска Vault в Kubernetes в качестве одного экземпляра, использующего утверждение постоянного тома (PVC) и хранилище на основе файловой системы.

Доступ к секретам из Kubernetes

Vault имеет встроенную поддержку Kubernetes и может использовать API Kubernetes для проверки подлинности приложения. Это работает так: когда приложение взаимодействует с Vault, оно использует веб-токен JSON (JWT) из учетной записи службы для проверки подлинности и запроса именованной роли. Хранилище должно быть предварительно настроено для обратного взаимодействия с Kubernetes, используя учетные данные, позволяющие ему использовать роль token Reviewer и проверять JWT, предоставленный модулем приложения. Приложение, поставляемое JWT, содержит сведения об имени учетной записи службы и имени пространства имен для модуля приложения. Vault проверяет, является ли JWT допустимым и что запрошенная роль настроена на разрешение доступа к имени учетной записи службы и имени пространства имен. Если все хорошо, Vault возвращает маркер доступа, который затем может быть использован приложением для запроса секретов, разрешенных для этой роли, с помощью одной или нескольких политик.



Установка Vault:

1. Включите метод проверки подлинности Kubernetes для каждого кластера. Для этого требуется адрес сервера API кластера, открытые ключи и веб-токен JSON (JWT), предоставляющий доступ к роли рецензента маркеров Kubernetes.

2. Определите политику для приложения, предоставляющую доступ к секретам, необходимым приложению. Эту политику можно повторно использовать для доступа к кластерам Kubernetes.

3. Определите роль для приложения. Определение роли включает набор разрешенных имен учетных записей службы Kubernetes и разрешенных пространств имен, а также сопоставление с политикой. Роль специфична для кластера.

Адрес сервера API кластера и открытые ключи можно получить из файла конфигурации kubectl. Адрес сервера API находится в разделе кластеры/кластер/сервер, а сертификат — в разделе кластеры/кластер/центр сертификации-данные. Данные сертификата должны быть декодированы base64 для входа в Vault.

Вот подробная инструкция:

Создайте новую учетную запись службы, которая будет использоваться Vault для доступа к кластеру:

```
kubectl -n kube-system create sa vault-token-review
```

Добавьте кластерную роль для учетной записи службы с помощью следующего YAML:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: role-vault-token-review-binding
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: vault-token-review
  namespace: kube-system
```

Извлеките JWT с помощью следующих команд:

```
SECRET_NAME=$(kubectl -n kube-system get serviceaccount vault-token-review -o jsonpath='{.secrets[0].name}')
kubectl -n kube-system get secret ${SECRET_NAME} -o jsonpath='{.data.token}' | base64 --decode
```

Затем можно настроить метод проверки подлинности Vault Kubernetes, используя адрес кластера, сертификат и роль проверки маркера JWT.

В Kubernetes:

4. Запустите модуль приложения в разрешенном пространстве имен, используя разрешенную учетную запись службы.

5. Используйте JWT учетной записи службы для проверки подлинности в Vault и получения маркера доступа.

6. Используйте маркер доступа для получения секретов.

2. Карты конфигураций

ConfigMap - это функция Kubernetes для внедрения конфигурации в свои контейнеры. Они позволяют хранить настройки стека отдельно от кода. Узнайте, как работать с ConfigMaps и предоставить их своим модулям.

Для чего нужны ConfigMaps?

ConfigMaps - это специально разработан для инкапсуляции небольших объемов неконфиденциальных данных конфигурации. Они представляют собой механизм для получения произвольных пар ключ-значение в ваших модулях. Они обычно используются для хранения IP-адреса сервера базы данных, адреса исходящей электронной почты вашего приложения и других параметров приложения, которые необходимо настраивать вне модулей.

ConfigMap позволяет управлять этими данными в выделенном ресурсе Kubernetes. Модули получают пары "ключ-значение" как переменные среды или файлы на смонтированном томе.

Почему бы их не использовать?

В некоторых ситуациях ConfigMap должен *не* использоваться.

ConfigMaps хранятся ненадежно, и их значения не зашифрованы. Они не должны содержать конфиденциальных данных, которые могут представлять угрозу безопасности или конфиденциальности в случае утечки.

Не помещайте пароли, ключи API или ключи шифрования в ConfigMap - используйте секрет Kubernetes, поскольку они работают аналогично ConfigMaps, но с дополнительной защитой. Системы, которым требуется подключение к базе данных, должны поместить имя хоста в ConfigMap, а учетные данные - в отдельный секрет.

Размер отдельных файлов ConfigMaps не может превышать 1 МБ. Системы, которым требуется больше ключей конфигурации, могут лучше обслуживаться альтернативным подходом, таким как внедрение вручную сгенерированных файлов конфигурации через том.

Если вы хотите сохранить ConfigMaps, рассмотрите возможность разделения вашей конфигурации на различные ресурсы ConfigMap. При таком подходе следует избегать ограничения в 1 МБ, что позволяет предоставить каждому из ваших модулей минимальный набор необходимых вам ключей конфигурации.

Значения ConfigMap могут быть строками UTF-8 или двоичными данными, закодированными как строка base64. Имена клавиш могут содержать буквенно-цифровые символы, . (временной ход), - (дефис)

и _ (подчеркнутые) символы. Некоторые языки программирования и фреймворки могут иметь другое соглашение для переменных конфигурации, поэтому обязательно используйте формат, совместимый с Kubernetes и вашим приложением.

Создание ConfigMap

ConfigMaps имеют простые манифесты YAML. Каждой ConfigMap требуется name в стандартном формате Kubernetes и data поле, содержащее его пары ключ-значение:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  database_host: "192.168.0.10"
  system_email: "k8s@example.com"
```

Поле `data` предназначено для указания ключей со строковыми значениями. Вы можете использовать `binaryData` или `data` для добавления двоичных значений в кодировке base64. Ключи должны быть уникальными на обоих `data` и `binaryData`.

Примените манифест к своему кластеру, используя `kubectl` или ваш любимый инструмент.

Связывание карт ConfigMaps и модулей

Сама по себе ConfigMap ничего не делает. Вы добавили данные в свой кластер; теперь давайте свяжем его с модулем:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
```

```
    - name: example-container
      image: example-image:latest
      envFrom:
        - configMapRef:
            name: example-configmap
```

Поле envFrom извлекает переменные среды, определенные другим ресурсом, на который имеется ссылка. В этом случае configMapRef идентифицирует созданную ранее ConfigMap. Контейнеры для капсул будут начинаться с database_host и system_email Определены переменные среды.

Выборочное добавление переменных среды

envFrom полезен, если вы хотите использовать все ключи в ConfigMap и уверены, что не будет конфликтов с другими переменными среды в вашем модуле. В более контролируемых ситуациях используйте env раздел, определите отдельные ключи и извлеките значение каждого ключа из ConfigMap:

```
env:
  - name: DATABASE_HOST_IP
    valueFrom:
      configMapKeyRef:
        name: example-configmap
        key: database_host
```

В этом примере показано, как можно запустить модуль только с database_host Ключ ConfigMap. Ключ также переименовывается перед инъекцией, чтобы модуль получил его как DATABASE_HOST_IP.

Использование ConfigMaps с томами

ConfigMaps можно монтировать как файлы внутри модулей. Kubernetes создает том, внедряет содержимое ConfigMap в виде набора файлов и монтирует том в вашем модуле.

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
      image: example-image:latest
      volumeMounts:
        - name: app-config
          mountPath: "/etc/config-data"
          readOnly: true
  volumes:
    - name: app-config
      configMap:
        name: example-configmap
```

Этот манифест модуля создает том с именем app-config, или configMap будет предварительно заполнить том, используя данные из указанной ConfigMap.

Pod монтирует том к /etc/config-data. Его контейнеры могут читать файлы в каталоге, чтобы получить доступ к их значениям конфигурации. Каждый ключ ConfigMap будет иметь свой собственный файл в точке монтирования.

Обновление значений ConfigMap

Поскольку ConfigMap является стандартной функцией Kubernetes API, вы можете обновить значения в любое время, изменив его манифест и повторно применив его к кластеру. Как новые значения достигают ваших модулей, зависит от используемого вами механизма инъекции.

Смонтированные тома

ConfigMaps, смонтированные в подах через том, будут обновлены Kubernetes. Изменения в ConfigMaps периодически проверяются; при обнаружении разницы файлы на вашем томе будут обновлены, поэтому ваш

модуль получит новые данные. Задержка зависит от интервал синхронизации, настроенный для экземпляров Kubelet на их рабочих узлах.

Переменные среды

Изменение переменных среды модуля невозможно, поэтому изменения ConfigMap не дойдут до существующих модулей, которые ссылаются на ключи через этот механизм. Чтобы использовать новые данные, необходимо заменить модули.

Вновь созданные модули всегда будут получать текущие данные ConfigMap, независимо от того, используете ли вы тома или переменные среды. Если вам нужно принудительно обновить конфигурацию, измените аннотацию в модуле, чтобы Kubernetes воссоздал ее.

Неизменяемые карты конфигурации

У ConfigMaps есть опция immutable поле, которое препятствует их обновлению. Если это поле установлено, вы не можете обновить данные ConfigMap или удалить неизменяемый статус.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: immutable-configmap
data:
  foo: bar
immutable: true
```

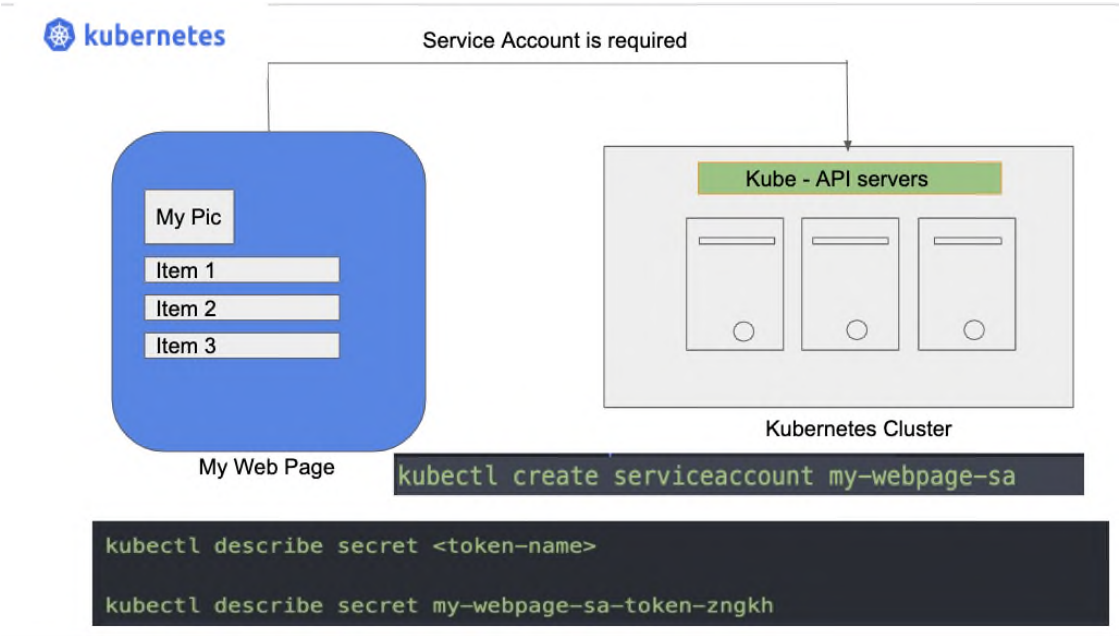
Это может быть полезно, если вы уверены, что значения конфигурации никогда не изменятся. Кроме того, это повышает безопасность, устраняя возможность случайного редактирования. Производительность также можно улучшить, поскольку Kubernetes больше не нужно отслеживать ConfigMap для распространения любых изменений значений на ваши поды.

Резюме

ConfigMaps должен быть вашим выбором для предоставления неконфиденциальных ключей конфигурации для модулей Kubernetes. Это первоклассный ресурс API, который можно использовать как переменные среды или файлы, смонтированные на томах.

Пароли и другие учетные данные принадлежат Секретам. Они работают очень похоже на ConfigMaps и ссылаются на модули таким же образом. Замените на, чтобы извлечь ключ из именованного секрета вместо ConfigMap.configMapRefsecretRef.

3. Сервис-аккаунты



Что такое учетная запись службы в Kubernetes?

В Kubernetes существует два типа учетных записей

- **Учетная запись пользователя**: она используется, чтобы позволить нам, людям, получить доступ к данному кластеру Kubernetes. Для этого любой пользователь должен пройти проверку подлинности на сервере API. Учетная запись пользователя может быть администратором или разработчиком, который пытается получить доступ к ресурсам уровня кластера.
- **Учетная запись службы**: используется для проверки подлинности процессов на уровне компьютера для получения доступа к кластеру Kubernetes. Сервер API отвечает за такую аутентификацию процессов, запущенных в модуле

Учетная запись службы :

В кластере Kubernetes любые процессы или приложения в контейнере, находящемся в модуле, могут получить доступ к кластеру, пройдя проверку подлинности на сервере API с помощью учетной записи службы.

Например:

Приложение, **подобное Prometheus**, обращающееся к кластеру для его мониторинга, является типом учетной записи службы.

Так

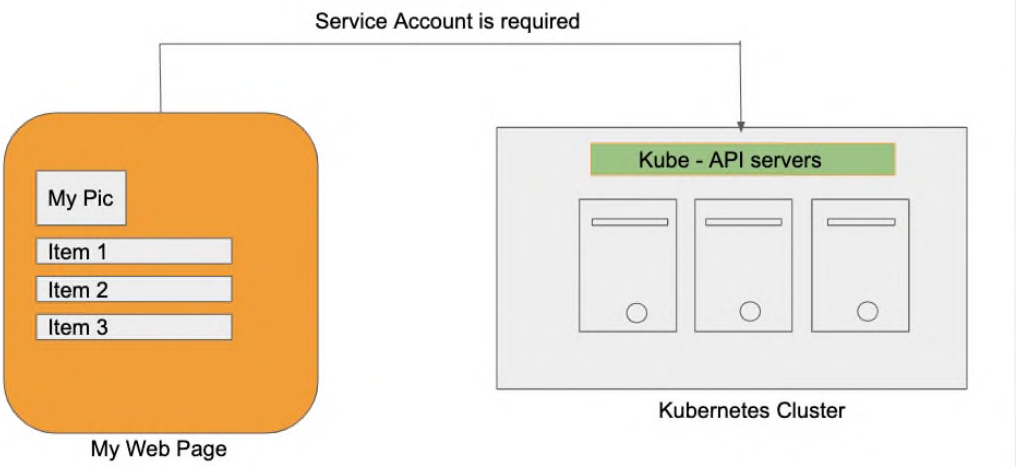
Учетная запись службы — это удостоверение, которое присоединяется к процессам, выполняемым в модуле.

Заметка!

При создании модуля, если учетная запись службы не указана, ей автоматически назначается учетная запись службы в том же пространстве имен.default

Как работает учетная запись службы Kubernetes?

Случай 1: Когда внешнее приложение пытается получить доступ к серверам API кластера Kubernetes.



Веб-страница пытается получить доступ к кластеру Kubernetes

Предположим, что существует веб-страница: **Моя веб-страница** со списком отображаемых элементов, эти данные должны быть извлечены с сервера API, размещенного в кластере Kubernetes, как показано выше на рисунке. Для этого нам нужна учетная запись службы, которая будет включена серверами API кластера для проверки подлинности и доступа к данным с серверов кластера.

Создание учетной записи службы:

Теперь, чтобы получить доступ к кластеру kubernetes, как обсуждалось выше, нам нужно создать учетную запись службы, что мы можем сделать с помощью следующей команды:

```
kubectl create serviceaccount my-webpage-sa
```

Эта команда сгенерирует учетную запись службы с именем: **my-webpage-sa**

Когда вы закончите создавать учетную запись службы, также генерируется **маркер учетной записи службы**, этот маркер будет требоваться нашему приложению My Web Page для доступа к данным через **API**.

Давайте посмотрим, как вы можете просмотреть токен и другие вложенные сведения с помощью созданной учетной записи службы.

Описание учетной записи службы:

необходимо ввести следующую команду **kubectl**:

```
kubectl describe serviceaccount my-webpage-sa
```

Вывод приведенной выше команды:

```
(base) Pramods-MacBook-Air:~ prammobibt$ kubectl describe serviceaccount my-webpage-sa
Name:                my-webpage-sa
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   my-webpage-sa-token-zngkh
Tokens:              my-webpage-sa-token-zngkh
Events:              <none>
(base) Pramods-MacBook-Air:~ prammobibt$
```

Поэтому, если вы внимательно посмотрите на вывод, вы увидите, что атрибут **Tokens** создан со значением: **my-webpage-sa-token-zngkh.** Этот токен хранится как секретный объект, этот секретный объект прикрепляется к учетной записи службы:**my-webpage-sa.**

Чтобы просмотреть секретный объект:

Если вы хотите посмотреть, каково содержимое объекта secrte, мы можем ввести следующую команду

```
$ kubectl описать секрет <token-value>
```

как показано ниже:

```
kubectl describe secret <token-name>

kubectl describe secret my-webpage-sa-token-zngkh
```

вывод**:**

```
(base) Pramods-MacBook-Air:~ prammobibt$ kubectl describe secret my-webpage-sa-token-zngkh
Name:         my-webpage-sa-token-zngkh
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: my-webpage-sa
              kubernetes.io/service-account.uid: ba84d467-f17b-44e8-9a3a-e989b80ffc56

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1066 bytes
namespace:  7 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6Im9HMGt2Qk8tY0xOcnVGXzZ6TGx4V0VMZ1pWUlVvNXhQXy1PSFB4dHdzSWcifQ.eyJpc
3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0I
iwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6Im15LXdlYnBhZ2Utc2EtdG9rZW4tem5na2giLCJrdWJlcm5ld
GVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoibXktd2VicGFnZS1zYSIsImt1YmVybmV0ZXMuaW8vc2VydmljZ
WFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6ImJhODRkNDY3LWYxN2ItNDRlOC05YTNhLWU5ODliODBmZmM1NiIsInN1YiI6InN5c3Rlb
TpzZXJ2aWNlYWNjb3VudDpkZWZhdWx0Om15LXdlYnBhZ2Utc2EifQ.R4aGnFUAj_i9GK3vnKiBW8BhBIKFTx6iT-31261M2huJ2XpUNV4hwP
5MYNVGvSv3Dva87mahp9eiCg5v0KOSYAKhOsxQSO0lDTi0WcL98-e6pToeRm6po0i76Abz3pdm_lUzHbsy--l8xFEvHh7m0k2-99bz-Wv236
AVClqDPJMpK0PvHipQTGrDFyEMO57OGnbd-hhOrn3oG4zcbxA4BgcNdFMr0wNp7Ivloa1QXRZyO2HjwAhlcPjkTxkcqe_upc1hrEm6RI9r_V
r1tZNOm90nBlRsaJEgo9AymmYOA1-jhBVaR2pkdu4t7Lv5YgA-pYePUDAfi6k9TDs6Xc4s3A
```

при выполнении приведенной выше команды можно просмотреть закодированное значение хэш-ключа маркера, как показано на рисунке выше.

Это ключ, которым можно обмениваться в качестве маркера носителя проверки подлинности в вызове REST API для получения необходимых данных с сервера API кластера Kubernetes.

Сводка:

Подведем итоги:

- Сначала нам нужно создать учетную запись службы
- Затем на основе RBAC:(Управление аутентификацией на основе ролей) нам нужно извлечь и экспортировать хэш-ключ токена, который будет передан в наш заголовок REST API. Это позволит получить доступ к серверу API кластера в качестве учетной записи службы, прошедшей проверку подлинности.

Но что, если наше приложение является неотъемлемой частью самого кластера и лежит в одном из POD

Случай 2: Когда приложение размещено и запущено в кластерном POD

В этом сценарии при создании любого модуля в кластере Kubernetes с любым заданным **пространством имен** эти модули по умолчанию создают учетную запись службы с именем **по умолчанию**. Учетная запись службы по умолчанию автоматически создает маркер службы вместе с требуемым секретным объектом.

Таким образом, наше приложение сможет получить доступ к серверу API, находящемуся в том же пространстве имен, используя эту учетную запись службы **по умолчанию**, смонтированную в модуле.

Например, введите на терминале следующую команду:

-$ kubectl получить serviceaccount

```
(base) Pramods-MacBook-Air:~ prammobibt$ kubectl get serviceaccount
NAME              SECRETS    AGE
default           1          15d
my-webpage-sa     1          33m
```

Вы увидите секрет по умолчанию, как выделено выше, и если вы пойдете дальше, чтобы ввести приведенный ниже набор команд для доступа к секрету по умолчанию, присоединенному к маркеру **по умолчанию**

```
(base) Pramods-MacBook-Air:~ prammobibt$ kubectl get secret
NAME                  TYPE                                  DATA   AGE
default-token-7k7zj   kubernetes.io/service-account-token   3      15d
```

вы сможете получить имя значения токена по умолчанию, **default-token-7k7zj (обратите внимание, что это будет варьироваться в вашем случае),** это автоматически создается при создании любого модуля в данном пространстве имен узла**,**

Чтобы просмотреть сведения о секретном объекте для маркера по умолчанию, default-token-7k7z:

Введите следующую команду:

$ kubectl описать секретный токен по умолчанию-7k7zj

```
(base) Pramods-MacBook-Air:~ prammobibt$ kubectl describe secret default-token-7k7zj
Name:         default-token-7k7zj
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: default
              kubernetes.io/service-account.uid: 37031ed1-af21-442a-9303-7f1f788bf5bb

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1066 bytes
namespace:  7 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6Im9HMGt2Qk8tY0xOcnVGXzZ6TGx4V0VMZ1pWUlVvNXhQXy1PSFB4dHdzSWcifQ.eyJpc
3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0I
iwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6ImRlZmF1bHQtdG9rZW4tN2s3emoiLCJrdWJlcm5ldGVzLmlvL
3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoiZGVmYXVsdCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2Vyd
mljZS1hY2NvdW50LnVpZCI6IjM3MDMxZWQxLWFmMjEtNDQyYS05MzAzLTdmMWY3ODhiZjViYiIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb
3VudDpkZWZhdWx0OmRlZmF1bHQifQ.Y0y7_dVBEegt4HK9rhfOi6Oo6yNzxkaPhm9H7RZ3C62WCdxXnydIuz5hxJkthjTNjL6zubmYfb_3j9
3WQy5PamDfwicED4BIhntYxJRszkVqCUH5sLj32RGwbkdhnXxTfgjp8lQZpUAxOpCKqYVInRvPNteuatADUBWeL2iZzUGRYYZWohzjHkJGyX
7XQ8JdsJXpLucq80iar4UZ4_5sM59xSWmM7HPFlkpkcsaKRmV09ZzY-KKOBd_jjw2hXCCK9Ch62nelVBa7TUcepztFJEDsU4bOBsRMl40vX9
jAaX8Eb5vcTpG3zWSBfclrJfdkT-cjK_gA2w4-JWiBPu2VNQ
```

Таким образом, для нашего приложения, размещенного в модуле с тем же пространством имен, этот секретный объект по умолчанию можно использовать для предоставления доступа к серверам API, расположенным в том же пространстве имен кластера.

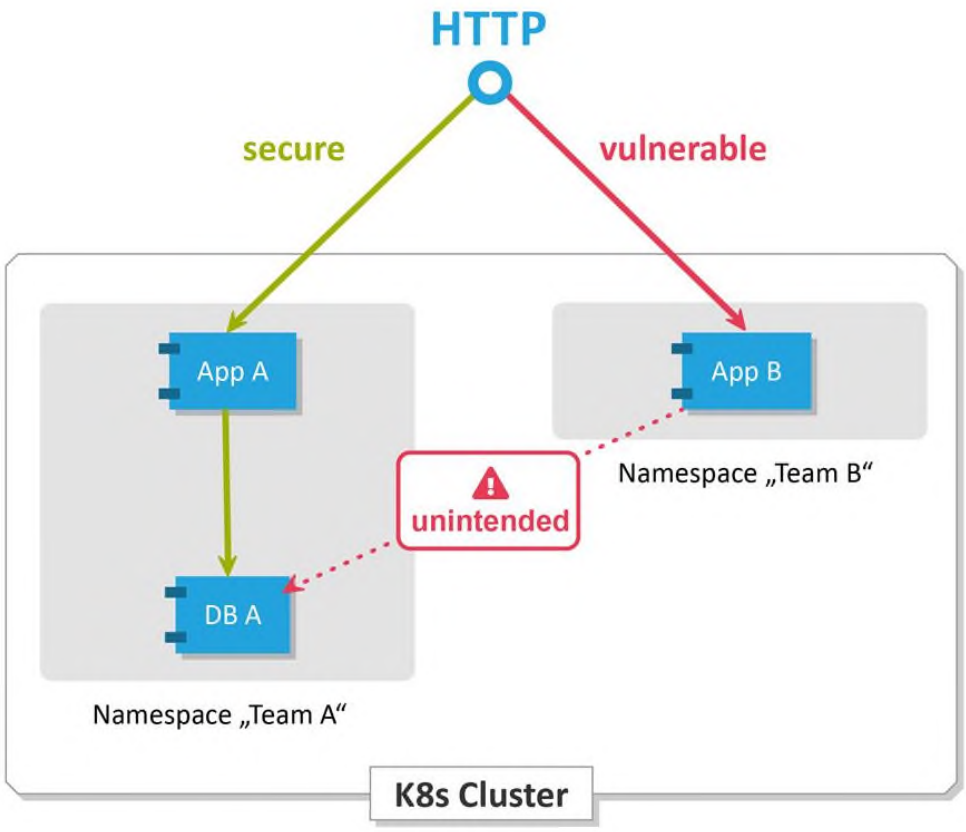
4. NetworkPolicies, SecurityContext

Использование сетевых политик

Модель сети Kubernetes может показаться необычной для многих: она требует плоской, неиерархической сети, в которой все (узлы, капсулы, кубелеты и т. Д.) Может взаимодействовать со всем остальным. Трафик в кластере не ограничен по умолчанию, в том числе даже между пространствами имен. Это тот момент, когда традиционные системные администраторы тщетно спрашивают о сегментах сети и брандмауэрах. Но Kubernetes предлагает ресурс, который можно использовать для ограничения сетевых подключений. Правила для входящего и исходящего трафика объявляются в сетевой политике для уровней ISO/OSI 3 и 4 (IP-адреса и порты), и эти правила назначаются группе модулей с помощью меток. Затем правила запрашиваются из API Kubernetes и применяются подключаемым модулем Интерфейса сети контейнеров (CNI) (например, Calico, Cilium).network policy

Таким образом, трафик модулей не ограничен, если

- Сетевых политик не существует
- Модуль не имеет метки, соответствующей сетевой политике, или
- Используемый подключаемый модуль CNI не реализует сетевые политики.



Такое поведение по умолчанию не является идеальным с точки зрения безопасности. В частности, когда в одном кластере работает несколько приложений, команд или клиентов, одно уязвимое приложение может обеспечить возможность для атак. На рисунке 1 приведен пример: Злоумышленник использует уязвимое приложение для "Team B" для получения данных о клиентах из базы данных "Team A". Хотя он не доступен напрямую через Интернет, он неправильно настроен и не требует аутентификации. Подобные случаи не являются академическими! Например, экземпляры MongoDB эксплуатировались тысячи раз без аутентификации и были напрямую доступны через Интернет.

Кроме того, исходящий трафик может представлять угрозу. Например, скомпрометированный узел может попытаться подключиться к командному серверу. Это можно смягчить, заблокировав исходящие соединения.

Общие возможности и синтаксис

Сетевые политики позволяют ограничить трафик и тем самым ограничить влияние атаки. Это позволяет разрешить входящий () и/или исходящий () трафик для определенных модулей, целых пространств имен или диапазонов IP-адресов (только исходящий). При желании трафик также может быть ограничен портом.ingressegress

Как это принято в Kubernetes, требуемое состояние кластера определяется декларативно в YAML. Простой пример сетевой политики приведен в листинге 1. Назначение стручков слабо привязано к соответствующей метке. Это относится как к модулям, к которым применяется сетевая политика (), так и к спецификации правил (которая приведена ниже в примере, начиная с) .podSelector podSelector ingress

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: db-allow
  namespace: team-a
spec:
  podSelector:
    matchLabels:
      db: a
  ingress:
  - from:
      - podSelector:
          matchLabels:
            app: a
```

Листинг 1: Простая сетевая политика, предотвращающая нежелательный доступ из рисунка 1

Поскольку трафик данных в кластере полностью не ограничен без сетевых политик, сначала должна быть применена сетевая политика, которая предотвращает все подключения (политика deny-all, см. листинг 2). Затем желаемые соединения могут быть разрешены с помощью дальнейших сетевых политик (белый список). Сетевая политика, приведенная в листинге 1, применяется ко всем модулям с меткой "db:a" и разрешает только входящий трафик от модулей с меткой "app:a". Как правило, сетевые политики применяются к модулям в пространстве имен, в котором они применяются (в

листинге 1 "team-a"). Тем не менее, можно указать или правила, которые применяются к другим пространствам имен (подробнее об этом вы можете узнать в следующем разделе) .ingress egress

Защита отдельных модулей, как показано в листинге 1, является началом. Но как мы можем гарантировать, что нежелательные сетевые доступы не происходят в больших кластерах и сложных ландшафтах приложений? Один из способов решения этой проблемы будет показан ниже в следующих примерах.

С помощью сетевых политик можно разрешать или запрещать входящий и исходящий сетевой трафик (входящий, исходящий). Вопрос здесь в том, с чего начать. Разумным первым шагом является внесение в белый список входящего трафика из приложений. На более поздних этапах книга описывает ограничение исходящего трафика и специальные пространства имен, такие как пространства имен операторов.kube-system

Белый список входящего трафика

Разумным первым шагом является белый список входящего трафика.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: ingress-default-deny-all
  namespace: team-a
spec:
  podSelector: {}
ingress: []
```

Листинг 2: Сетевая политика, запрещающая любое взаимодействие с модулями в пространстве имен

Это может быть реализовано с помощью сетевой политики для каждого пространства имен, которая выбирает все модули из пространства имен и блокирует весь входящий трафик. В листинге 2 показано, как это работает: выбраны все модули, но не описаны разрешенные правила. После этого модули в пространстве имен не могут быть доступны, если это явно не разрешено, как показано в листинге 1. ingress

Если это применяется к каждому пространству имен (за исключением начального ; дополнительные сведения см. в следующем разделе), можно установить только явно разрешенные соединения между модулями в кластере. Этот метод дает большой эффект с управляемым количеством усилий. kube-system

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: ingress-allow-traefik-to-access-app-a
 namespace:  team-a
spec:
 podSelector:
  matchLabels:
   app: a
 ingress:
 - ports:
   - port: 8080
  from:
   - namespaceSelector:
      matchLabels:
       namespace: kube-system
     podSelector:
      matchLabels:
       app: traefik
```

Листинг 3: Сетевая политика, разрешающая входящий трафик от контроллера входящего трафика

Однако, что не следует забывать, так это белый список трафика из

- контроллер входящего трафика (например, NGINX или Traefik), поскольку дальнейшие входящие запросы не могут быть сделаны (в листинге 3 показан белый список для Traefik), и
- инфраструктуру наблюдаемости, так как в противном случае Prometheus (листинг 4) перестанет собирать метрики приложения, например.

При выборе пространств имен обратите внимание, что по умолчанию они не имеют меток. Например, чтобы получить доступ к листингу 3, необходимо добавить метку в пространство имен. Преимущество назначения меток вместо имен заключается в том, что метка также может быть назначена нескольким пространствам имен. Это позволяет, например, применять сетевую политику ко всем производственным пространствам имен. namespace: kube-system kube-system

Одновременное использование и, как показано в листингах 3 и 4, возможно только начиная с Kubernetes версии 1.11.namespaceSelectorpodSelector

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: ingress-allow-prometheus-to-access-db-a
  namespace: production
spec:
  podSelector:
    matchLabels:
      db: a
  ingress:
  - ports:
      - port: 9080
    from:
      - namespaceSelector:
          matchLabels:
```

```
      namespace: monitoring
    podSelector:
     matchLabels:
       app: prometheus
       component: server
```

Листинг 4: Сетевая политика, позволяющая Prometheus отслеживать приложение

Для опытных пользователей: пространство имен kube-system

Как уже упоминалось, при внесении в белый список собственного пространства имен Kubernetes kube-system и любых других пространств имен операторов, которые могут существовать (контроллер Ingress, Prometheus, Velero и другие), следует учитывать больше, чем другие пространства имен, в которых выполняются общие приложения.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: allow-kube-dns-all-namespaces
 namespace: kube-system
spec:
 podSelector:
  matchLabels:
   k8s-app: kube-dns
 ingress:
  - ports:
   - protocol: UDP
     port: 53
   from:
    - namespaceSelector: {}
```

Листинг 5: Сетевая политика, позволяющая получить доступ к DNS-сервису kube-dns

Если какой-либо входящий трафик в пространстве имен запрещен, это повлияет, среди прочего:kube-system

- Разрешение имен с помощью надстройки DNS (например, kube-dns)
- и входящий трафик извне на контроллер входящего трафика, который часто работает в этом пространстве имен.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: ingress-allow-traefik-external
 namespace: kube-system
spec:
 podSelector:
  matchLabels:
   app: traefik
 ingress:
  - ports:
     - port: 80
     - port: 443
   from: []
```

Листинг 6: Сетевая политика, разрешающая HTTP-доступ к входящему контроллеру Traefik

В листинге 5 показано, как обычно можно авторизовать доступ DNS к модулям kube-dns: UDP-трафик разрешен на порту 53 из всех пространств имен в модули. В листинге 6 приведен пример обратного прокси-сервера Traefik HTTP, где доступ к портам 80 (этот трафик обычно перенаправляется на порт 443) и 443 разрешен без необходимости указания конкретного. from

Для опытных пользователей: ограничение исходящего трафика

Если входящий трафик занесен в белый список во всех пространствах имен, несанкционированный доступ больше не может осуществляться в

кластере: исходящий трафик модуля также является входящим трафиком другого модуля, занесенного в белый список. Однако есть еще один источник исходящего трафика, который требует более пристального внимания: трафик из кластера в окружающие сети, такие как Интернет или корпоративная сеть. Многим приложениям не нужно взаимодействовать с миром за пределами кластера или им нужно взаимодействовать только с выбранными узлами. Примерами разрешенных доступов являются загрузка плагинов (Jenkins, SonarQube и т. д.) или доступ к инфраструктуре аутентификации (OAuth-провайдеры, такие как Facebook, GitHub и т. Д. Или корпоративный LDAP / Active Directory). Как описано выше, злоумышленники могут использовать подключение к Интернету или доступ к корпоративной сети, которая может использоваться в качестве шлюза для других целей. Эти атаки можно предотвратить, начиная с Kubernetes версии 1.8 за счет ограничения трафика.egress

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: egress-allow-internal-only
  namespace: team-a
spec:
  policyTypes:
    - Egress
  podSelector: {}
  egress:
    - to:
    - namespaceSelector: {}
```

Листинг 7: Сетевая политика, разрешающая исходящий трафик только внутри кластера

Более прагматично разрешать исходящий трафик в кластере только с одной сетевой политикой на пространство имен, а затем разрешать отдельным модулям доступ к внешнему пространству. В листинге 7 показана

сетевая политика, которая делает это, применяя ее ко всем модулям и разрешая все пространства имен. Все, что находится за пределами сети модулей, не имеет пространства имен и поэтому отклоняется. ingress ingress egress egress

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: egress-allow-all-egress
 namespace: team-a
spec:
 policyTypes:
  - Egress
 podSelector:
  matchLabels:
   app: a
 egress:
  - to:
    - ipBlock:
       cidr: 0.0.0.0/0
```

Листинг 8: Сетевая политика, разрешающая весь исходящий трафик для определенных модулей

Затем белый список происходит таким же образом, как и через дополнительные сетевые политики. Однако это относится к блокам IP-адресов с использованием нотации бесклассовой междоменной маршрутизации (CIDR). Пример, разрешающий весь трафик для конкретных модулей, приведен в листинге 8. Здесь вы можете сделать еще более детальные корректировки, разрешив выбранные IP-адреса или диапазоны. ingress egress

При внесении в белый список, важно не забывать об облачных приложениях, которые напрямую взаимодействуют с API Kubernetes. Сопоставление IP-адресов сервера API является сложной задачей. Здесь

должен быть указан IP-адрес конечной точки, а не адрес службы. Этот IP-адрес можно получить с помощью следующей команды: "kubectl get endpoints –namespace default kubernetes". Полученный IP-адрес можно поместить в сетевую политику, как показано в листинге 8 (x.x.x.x/32 вместо 0.0.0.0/0), чтобы разрешить доступ к серверу API.egress

Контекст безопасности

Наиболее прямым способом применения конфигураций, относящихся к безопасности, в Kubernetes является контекст безопасности. Он доступен на двух уровнях; за под и за контейнер. Некоторые конфигурации возможны на обоих уровнях. В этом случае приоритет имеет конфигурация на уровне контейнера. В листинге 1 приведен пример в YAML: Здесь указан модуль, контейнер которого не может быть запущен с пользователем "root". Исключение сделано для специальных контейнеров инициализации.

```
apiVersion: v1
kind: Pod
# ...
spec:
 securityContext:
  runAsNonRoot: true
 containers:
  - name: mustNotRunAsRoot
 initContainers:
  - name: isAllowedToRunAsRoot
    securityContext:
                    runAsNonRoot: false
```

Листинг 1: Модуль с контекстом безопасности для модуля и контейнера

Надлежащая практика

В соответствующем «securityContext» есть несколько конфигураций контейнеров на выбор, и есть больше опций на рычаге контейнера, чем на уровне модуля. Какая конфигурация здесь рекомендуется, и сколько усилий

это потребует? Листинг 2 содержит ряд конфигураций контейнеров, которые ограничивают значения по умолчанию и которые, как показал опыт, обеспечивают хорошую отправную точку, поскольку они могут блокировать определенные векторы атаки, не причиняя слишком много усилий. Если приложение не выполняется с этой конфигурацией, его можно сделать менее ограничительным позже. Обратите внимание, что в Kubernetes версий 1.18 и ниже seccompProfile настраивается с помощью аннотации: "seccomp.security.alpha.kubernetes.io/pod: runtime/default".

```
apiVersion: v1
kind: Pod
# ...
metadata:
 annotations:
   seccomp.security.alpha.kubernetes.io/pod: runtime/default
spec:
 containers:
 - name: restricted
   securityContext:
     runAsNonRoot: true
     runAsUser: 100000
     runAsGroup: 100000
     readOnlyRootFilesystem: true
     allowPrivilegeEscalation: false
     capabilities:
       drop:
         - ALL
```

Листинг 2: Модуль в сравнении с конфигурацией по умолчанию

Запуск контейнера с непривилегированным пользователем

Используя дефектную конфигурацию или уязвимости в ядре или среде выполнения контейнера, злоумышленник может вырваться из изоляции контейнера («побег контейнера»). Если эта попытка увенчается успехом, злоумышленник получит права пользователя, который запускает процесс в хост-системе за пределами контейнера. Таким образом, крайне важно избегать запуска контейнера от имени пользователя "root" (User ID 0), так как в этом случае злоумышленник будет иметь все права на хост-систему. Кроме того, рекомендуется присвоить контейнерам идентификатор пользователя или идентификатор группы, который еще не выделен существующему пользователю в системе, так как они могут иметь права доступа к файлам в системе, например, которые могут быть использованы злоумышленником.

Существуют возможные решения этой проблемы на нескольких уровнях:

- В образе (например, с помощью команды Dockerfile).USER 100000
- В контексте безопасности Kubernetes с , контейнер будет запрещен к запуску, если он попытается запуститься с UID 0 ("root").runAsNonRoot: true
- В контексте безопасности Kubernetes с и . Желательно выбрать значение больше 10000, так как это снижает вероятность того, что это значение уже занято в хост-системе. В Dockerfiles часто используется 1000, что обычно соответствует первому пользователю, созданному в системе. Это удобно на этапе разработки, но может обеспечить вектор атаки в производственной системе.runAsUserrunAsGroup

Из всех конфигураций, показанных в листинге 2, они оказывают наибольшее влияние на уменьшение поверхности атаки. Тем не менее, они также вызывают наибольшие усилия. Потому что:

- Многие официальные изображения предназначены для работы в качестве «root» (например, NGINX или Postgresql). Обычно доступны сторонние альтернативы, такие как, например, от битнами. NGINX даже предлагает выделенное изображение, которое работает как непривилегированный пользователь. Как правило, необходимо соблюдать осторожность при выборе изображения, так как не все изображения, найденные в Интернете, заслуживают доверия. Другим альтернативным решением является создание образа для нужного приложения самостоятельно, который явно определяет пользователя. Самым простым

подходом является использование официального изображения в качестве базового изображения.

- "runAsNonRoot" работает только с числовым идентификатором. Поэтому, если Dockerfile содержит, например, контейнер не будет запущен. Это можно обойти, явно установив для него значение "runAsUser".USER node
- В зависимости от владельца и режима доступа к файлам в контейнере или в томах может оказаться, что контейнер не имеет доступа во время выполнения. Это может быть решено путем изменения владельцев, групп или (с помощью команды). В зависимости от того, как он используется, это потребует изменения при запуске модуля или создании нового образа. С томами это можно сделать с помощью, который имеет больше прав, чем фактический контейнер приложения. В листинге 1 показана одна возможность. Другим распространенным решением является запуск процессов в корневой группе (идентификатор группы 0), например, в OpenShift. Чтобы это работало, группе 0 должны быть предоставлены права доступа к файлам в изображении, например, так: «chgrp -R 0 /some/directory && chmod -R g=u /some/directory». change mode chmod init Container

Текущим примером уязвимости, которую эта конфигурация может предотвратить, является CVE-2019-5736. Здесь уязвимость в низкоуровневом runc среды выполнения контейнера (который также используется Docker) позволяет злоумышленникам избежать контейнера, если контейнер запущен от имени пользователя "root".

Корневая файловая система, доступная только для чтения

Злоумышленник также может нанести ущерб, не покидая контейнер. Например, он может скомпрометировать код приложения, тем самым получив доступ к пользовательским данным. Более того, внутри контейнера злоумышленник может загружать программы из Интернета, чтобы расширить атаку на сеть, которая является непубличной, но может быть доступна из контейнера. Первый случай, в частности, можно полностью предотвратить, используя файловую систему, доступную только для чтения в контейнере. Чтобы обеспечить максимально короткое время запуска и эффективное использование памяти, контейнеры используют алгоритм копирования при записи: файловая система образа не копируется при запуске. Вместо этого он является только многоуровневым: каждому контейнеру, запущенному из образа, присваивается свой собственный слой, где хранятся файлы, записанные во время выполнения. Если файл, присутствующий в образе,

изменяется, он сначала копируется в слой контейнера, а затем изменяется. Доступ на чтение сначала проверит, присутствует ли файл в контейнерном слое, а если нет, то он будет доставлен из образа.

С помощью, контейнерный слой деактивируется, что означает, что запись в файловую систему во время выполнения больше невозможна. Это означает, что код больше не может быть изменен. Даже если менеджеры пакетов установлены, они, как правило, больше не будут работать, что затруднит загрузку приложений. Положительным побочным эффектом является улучшение работоспособности при чтении.readOnlyRootFilesystem: true

Однако большинство приложений изначально не совместимы с файловыми системами только для чтения. Например, веб-серверам требуются временные каталоги для кэширования (часто). В модуле Kubernetes эти необходимые каталоги можно сделать доступными в виде томов, таких как, например, как. Команда в Docker может использоваться для идентификации этих каталогов. Он показывает, что находится в контейнерном слое. Один из возможных подходов заключается в следующем:/tmpemptyDirdiff

- Запустите приложение локально как контейнер Docker (не с файловой системой только для чтения),
- Запустите тесты на системе и просмотрите все потенциальные варианты использования (в противном случае результаты могут быть неполными),
- Run .docker diff <containerId>

Результатом является список файлов, которые были записаны во время выполнения контейнера. Если все затронутые каталоги подключены в Kubernetes как тома, контейнер можно запустить там без каких-либо проблем.readOnlyRootFilesystem

Для большинства приложений это решение легко реализовать и обеспечивает повышенную безопасность. В некоторых случаях, например, когда приложение предназначено для изменения файлов из образа при запуске или во время выполнения, это может быть более проблематичным. Существует несколько возможных способов исправить это: в Dockerfile затронутый каталог может быть объявлен как. Затем файлы будут скопированы туда обработчиком контейнеров во время выполнения. Кроме того, это также может быть решено в Kubernetes: в модуле, а другие

контейнеры совместно используют тома. Таким образом, можно копировать файлы из изображения на том. В обоих случаях важно настроить пользователей, группы и обеспечить доступ к контейнеру приложения. VOLUME init Container init Container change mode

Предотвращение эскалации привилегий

Хотя контейнер может не работать как «root» пользователя, все же возможно, что злоумышленник может получить дополнительные права (т.е. повысить привилегии). Этот механизм наверняка знаком большинству пользователей Linux, ведь «sudo» (без каких-либо дополнительных параметров) можно использовать для выполнения команды с правами «корневого» пользователя. Поэтому команда "sudo" никогда не должна устанавливаться или настраиваться в любом контейнере. Изображений, содержащих судо, почти нет. Другим потенциальным способом получения дополнительных прав являются уязвимости, такие как, например, в ядре CVE-2015-3339. Установив, вы можете легко предотвратить это. Нежелательные побочные эффекты для применения в контейнере при использовании этого метода маловероятны. Если образ был разработан для запуска без прав "root", он не нуждается в каких-либо дополнительных правах во время выполнения. allowPrivilegeEscalation: false

Ограничение возможностей

В Linux можно расширить права процессов мелким способом, без необходимости запускать его как пользовательский «root» (которому разрешено делать все, что угодно). Для этого существуют определенные возможности, которые могут быть предоставлены процессам. Хорошо известные примеры включают доступ к сокетам (например, для приложения, возможность) или привязку к портам < 1024 (например, для веб-серверов, возможность).ping NET_RAW NET_BIND_SERVICE

Среды выполнения контейнеров запускают контейнерные процессы с выбранными возможностями, что представляет собой компромисс между удобством использования и безопасностью (пример: Docker-capabilities). Часто эти возможности не требуются приложениям, но они предоставляют злоумышленнику больше прав и, следовательно, увеличивают поверхность атаки. Например, с возможностью «NET_RAW» атака типа «злоумышленник посередине» может быть выполнена на связь всех контейнеров с хостом с помощью DNS-Spoofing.

Чтобы сделать этот сценарий более безопасным, можно использовать подход к составлению белых списков: запуск без возможностей, допускающий только выбранные из них там, где это необходимо.

Как и в случае с файловой системой только для чтения, Docker можно использовать локально, чтобы эмпирически определить, какие возможности требуются. Это делается путем запуска соответствующего образа с помощью контейнера и его прохождения через все варианты использования с контейнером. Сообщения об ошибках можно использовать для быстрого определения отсутствующих возможностей, которые затем могут быть добавлены по мере необходимости с параметром. NGINX также является хорошим примером этого, поскольку он не может запускаться без определенных возможностей. К ним относится возможность привязки к порту 80. В этом конкретном случае одной из альтернатив добавлению возможности является использование другого образа, который настраивает NGINX таким образом, чтобы он не привязывался к порту 80, а вместо этого привязывался к порту больше 1024. См. nginx-unpriv, среди прочих. `--cap-drop ALL--cap-add NET_BIND_SERVICE`

Активация профиля seccomp по умолчанию

В ядре Linux изоляция контейнеров достигается с помощью механизмов безопасности, таких как seccomp. seccomp позволяет ограничивать системные вызовы, тем самым ограничивая доступ к функциям, реализованным в ядре. После всестороннего тестирования всех файлов Dockerfiles на GitHub [frazelle-container-security] Docker представил «профиль по умолчанию» в 2016 году, который предотвращает 44 из более чем 300 системных вызовов. Таким образом, он предлагает умеренную безопасность, но он по-прежнему совместим с большинством приложений (docker-seccomp). В Docker все процессы-контейнеры по умолчанию используют этот профиль. Однако в Kubernetes он был явно деактивирован из-за опасений по поводу совместимости (k8s-20870).. Однако параметр, который можно задать при запуске сервера API ("–seccompdefault"), был доступен только в качестве альфа-функции начиная с Kubernetes 1.22. Seccomp является ключевой функцией безопасности Docker и используется там в течение многих лет. Отсутствие Seccomp в Kubernetes также было отмечено в ходе аудита безопасности.

Поэтому очевидным является явный подход к явной настройке профиля для начала и его деактивации только по мере необходимости. Это должно

вызвать проблемы только в исключительных случаях. Например, официальные образы в DockerHub, стандартном реестре для Docker, могут отклоняться от конфигурации по умолчанию только в обоснованных исключительных случаях (docker-lib-official).

В Kubernetes seccomp еще не добрался до официального API. Поэтому он не может быть определен через контекст безопасности. Скорее, он должен быть определен с использованием аннотации. Это обычная практика для «альфы» функций до того, как они будут добавлены в определенный объект API. Аннотация указана для модуля и может применяться ко всему компоненту (см. листинг 2) или указываться для каждого контейнера.

Если вы хотите проверить, активен ли seccomp в контейнере, вы можете

- Запросите это в контейнере с помощью. означает, что нет активного профиля. В файловой системе отображается информация обо всех процессах, и только один процесс должен выполняться в контейнере, обычно с идентификатором процесса 1.grep seccomp /proc/1/status seccomp: 0 proc
- Вне контейнера это можно запросить в Docker с помощью , который найдет результаты только при наличии каких-либо отклонений от значения по умолчанию. Явно деактивированный seccomp (как в Kubernetes) отображается как .docker inspect seccomp:unconfined.