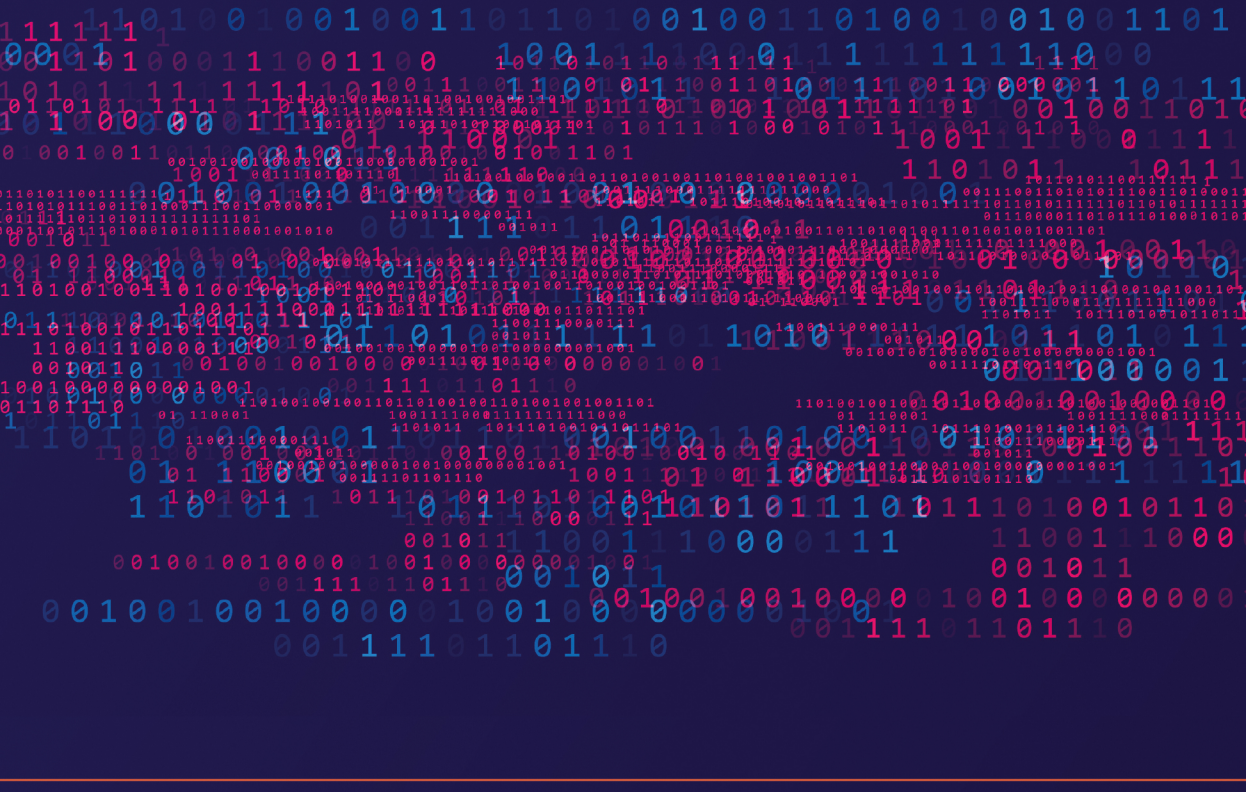
EXPERT INSIGHT

SPRINT book

Linux для разработчиков

Практическое углубленное руководство по командной строке и утилитам Linux для разработчиков программного обеспечения



Дэвид Коэн
Кристиан Штурм

packt

The Software Developer's Guide to Linux

A practical, no-nonsense guide to using the Linux command line and utilities as a software developer

David Cohen

Christian Sturm

‹packt›

BIRMINGHAM—MUMBAI

Linux для разработчиков

Практическое углубленное руководство по командной строке
и утилитам Linux для разработчиков программного обеспечения

Дэвид Коэн
Кристиан Штурм

SPRINT book 2025

Дэвид Коэн, Кристиан Штурм

Linux для разработчиков

Перевел с английского Р. Чебыкин

Научный редактор Д. Квист

ББК 32.973.2-018.2
УДК 004.451

Коэн Дэвид, Штурм Кристиан

К76 Linux для разработчиков. — Астана: «Спринт Бук», 2025. — 304 с.: ил.

ISBN 978-601-08-4837-5

Разработчики всегда стремятся подняться на новый уровень мастерства, но большинство полностью теряется, когда дело доходит до командной строки Linux.

С помощью этой книги вы сделаете следующий важный шаг в своей карьере. Большую часть навыков, которые вы получите после ее прочтения, можно сразу же применить на практике, чтобы стать более эффективным разработчиком.

Книга написана специально для программистов, а не для системных администраторов Linux. Каждая глава даст достаточно теоретических знаний, чтобы понять, что вы делаете, прежде чем переходить к практическим командам, которые вы сможете использовать в своей повседневной работе в качестве разработчика ПО.

По мере прочтения вы быстро освоите основы работы Linux и освоитесь с командной строкой.

Овладев основными навыками, вы разберетесь, как применять их в различных контекстах, с которыми столкнетесь как разработчик ПО: создание образов Docker и работа с ними, автоматизация скучных задач сборки с помощью сценариев оболочки и устранение неполадок в продакшен-средах.

К концу книги вы сможете с комфортом пользоваться Linux и командной строкой и применять приобретенные навыки в повседневной работе. Это позволит вам экономить время, быстро устранять неполадки и стать мастером работы с командной строкой, к которому обращается вся команда.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1804616925 англ.

978-601-08-4837-5

Изготовлено в России. Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес: 010000, Казахстан, город Астана, район Алматы, Проспект Ракымжан Кошкарбаев, дом 10/1, н. п. 18.

Дата изготовления: 03.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 14.02.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 700. Заказ 0000.

Краткое содержание

Оглавление

Создатели книги

Об авторах

Дэвид Коэн (David Cohen) в течение последних 15 лет побывал системным администратором Linux, разработчиком программного обеспечения, инженером по инфраструктуре, специалистом по надежности сайтов, экспертом по информационной безопасности, веб-разработчиком, а также опробовал несколько других профессий. В свободное время он ведет на YouTube канал *tutoriaLinux*, где уже обучил сотни тысяч посетителей основам Linux, программированию и DevOps. С 2019 года Дэвид трудится в компании HashiCorp : сначала он разрабатывал исследовательское ПО, затем эталонные архитектуры, а сейчас занимает должность инженера-программиста.

Спасибо тебе, Алейна, за то, что неизменно поддерживала меня последние несколько лет, когда я продумывал и писал эту книгу. Без тебя она превратилась бы в очередной «многообещающий» проект, который в конце концов затерялся бы в какой-нибудь папке «Архивы». Спасибо и Кристиану: он уже больше десяти лет остается моим другом и напарником во всех отчаянных технических проектах, которые я замышлял с тех пор, как мы познакомились. Наконец, сердечно благодарю друзей и коллег из HashiCorp и других компаний, где я работал последние 15 лет, за то, что эти люди помогли мне стать более грамотным специалистом и вдохновили на такие начинания, как эта книга.

Кристиан Штурм (Christian Sturm) — консультант по программной и системной архитектуре, более десяти лет проработавший на разных технических должностях. Он создавал приложения для фронтенда и бэкенда в больших и маленьких компаниях, таких как zoomsquare или Plutonium Labs. Кроме того, он активно участвует в различных проектах с открытым исходным кодом и выступает в качестве эксперта в нескольких областях, включая операционные системы, сетевые протоколы, информационную безопасность и системы управления базами данных.

О рецензентах

Марио Спливало (Mario Splivalo) — консультант в области баз данных, расширяемых на современные облачные архитектуры. Он также помогает компаниям налаживать инфраструктуру с помощью таких инструментов класса «инфраструктура как код», как *Terraform* и *AWS CloudFormation*. В течение пяти лет Марио работал специалистом по OpenStack в компании *Canonical*.

Марио увлечен цифровыми технологиями с тех пор, когда на рынке домашних компьютеров доминировал Commodore 64. Он начал программировать на «Бейсике» на отцовском C64, однако быстро переключился на ассемблер. В дальнейшем он пересел за полноценный персональный компьютер и проявил большую тягу к программированию, системной архитектуре и базам данных. В начале 2000 годов он переключился на Linux (сначала на Knoppix, а затем на Ubuntu, которая стала его основной системой) и продолжил профессиональную деятельность как администратор баз данных, программист и системный администратор.

Натан Ченслер (Nathan Chancellor) в качестве независимого подрядчика участвует в разработке ядра Linux и проживает в Аризоне (США). В основном его деятельность сосредоточена на том, чтобы улучшать совместимость ядра Linux с программной инфраструктурой LLVM. Натан работает на Linux с 2016 года, а с 2018 года использует Linux как основную операционную систему для разработки. Он предпочитает дистрибутивы Arch Linux и Fedora.

От издательства

Мы выражаем огромную благодарность компании Orion soft за помощь в работе над русскоязычным изданием книги и вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу

comp@sprintbook.kz (издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научном редакторе русского издания

Денис Квист — ведущий инженер-эксперт по виртуализации в компании Orion soft с более чем 24-летним опытом работы в IT. Его профессиональный путь начался с должности системного администратора и программиста в Национальном архиве Республики Карелия, где он стоял у истоков цифровизации архивной отрасли.

За свою карьеру Денис занимал ведущие позиции в различных организациях, включая гостиницу ParkInn, Республиканский перинатальный центр, Республиканский медицинский информационнно-аналитический центр. Под его руководством более двухсот студентов прошли обучение по основам Linux-систем на платформе GeekBrains. Сегодня он применяет знания как в личных проектах (обслуживает и развивает три FM-радиостанции), так и в профессиональной среде: активно участвует в разработке и поддержке системы виртуализации zVirt.

Предисловие

Многие разработчики ПО недостаточно хорошо разбираются в системах семейства Unix, несмотря на то что в мире разработки эти системы встречаются на каждом шагу. Некоторые программисты даже не подозревают, что в их должностные обязанности входит работа с Unix или подобными системами на своих компьютерах (macOS), в среде разработки (контейнеры Docker), в системах сборки и автоматизации (процессы непрерывной интеграции и GitHub), в среде развертывания (серверы и контейнеры Linux) и в других обстоятельствах.

Если вы уверенно владеете командной строкой Linux, то сможете добиваться большего, чем обычно ожидают от разработчика. Например, вы сумеете:

- экономить время благодаря тому, что вы знаете, когда лучше использовать встроенные средства Linux, вместо того чтобы писать тысячи строк собственных сценариев или вспомогательных программ;
- отлаживать сложные отказы в среде эксплуатации, для чего часто требуется задействовать серверы на основе Linux и обращаться к приложениям через их интерфейсы;
- быть наставником для джуниор-разработчиков;
- яснее понимать, как программное обеспечение, которое вы разрабатываете, вписывается в бóльшую экосистему и технологический стек.

Надеемся, что теория, примеры и проекты из этой книги помогут вам как разработчикам вывести свои навыки на новый уровень.

Для кого эта книга

Эта книга предназначена для разработчиков программного обеспечения, которые пока еще плохо знакомы с Linux и командной строкой или которые давно не практиковались и хотели бы освежить свои навыки. Если вы чувствуете себя неуверенно, когда в два часа ночи смотрите на приглашение командной строки Linux на продакшен-сервере, то эта книга для вас. Она также пригодится, если вы хотите быстро подтянуть свои знания в области Linux, чтобы продвинуться по карьер-

ной лестнице. Наконец, книга будет полезна и тем, кто просто любознателен и кому интересно, как работать эффективнее, если добавить немного магии командной строки в конфигурацию и процедуры своей среды разработки.

Чего нет в этой книге

Мы стремились к тому, чтобы книга была максимально полезной для действующих специалистов, и поэтому очень тщательно продумывали, что в нее войдет. Мы постарались отказаться от всего, что напрямую не относится к профессиональной деятельности разработчиков или к фундаментальным понятиям системы Linux и ее важнейших абстракций. Иными словами, книга полезна именно благодаря тому материалу, *который в нее не попал*.

«Linux для разработчиков» — не исчерпывающий курс по Linux. Эта книга не предназначена для системных инженеров или разработчиков ядра Linux. Поэтому она не является фолиантом объемом в тысячу страниц, и вы вполне сможете освоить ее за несколько дней, например, во время спокойного рабочего спринта.

О чем эта книга

В главе 1 «Как устроена командная строка» вы получите представление о том, как работает интерфейс командной строки и что такое командная оболочка, и быстро овладеете элементарными навыками работы в Linux. После краткого теоретического введения вы начнете осваиваться в командной строке: научитесь находить нужные файлы и выполнять базовые операции с ними, а также узнаете, где искать справочную информацию, если окажетесь в затруднении. Эта глава адресована начинающим разработчикам, которым важно научиться обращаться с командной строкой. Даже если вы прочтете только одну эту главу, то уже будете лучше подкованы в Linux, чем раньше.

Глава 2 «Как работать с процессами» — это своего рода обзорная экскурсия по процессам в Linux. Из нее вы вынесете полезные практические навыки того, как обращаться с процессами в командной строке. Мы подробнее рассмотрим ряд факторов (например, права доступа), с которыми сталкиваются разработчики ПО и которые часто служат источником проблем, связанных с процессами. Кроме того, мы освоим эвристические методы, с помощью которых можно бороться с этими проблемами. Вас ожидает также введение в более продвинутые темы, их мы раскроем дальше по ходу книги.

Глава 3 «Как управлять службами с помощью systemd» опирается на то, что вы узнали о процессах из предыдущей главы, и вводит дополнительный уровень

абстракции — службу `systemd`. Вы узнаете, какую роль в операционной системе выполняет подсистема инициализации и почему это важно. Затем мы рассмотрим все команды, которые понадобятся, чтобы управлять службами в системе Linux.

Короткая глава 4 «История команд в оболочке Linux» охватывает некоторые полезные приемы, которые помогут вам взаимодействовать с командной строкой быстрее и эффективнее. В этих приемах используются сочетания клавиш, а также история команд, благодаря которой приходится меньше дублировать нажатия на клавиши.

Глава 5 «Файлы в Linux» знакомит с файлами — фундаментальной абстракцией, которая позволяет понять, как на самом деле устроен Linux. Вы узнаете об иерархии FHS (Filesystem Hierarchy Standard), которая, подобно карте местности, помогает ориентироваться в любой системе семейства Unix. Затем вы освоите практические команды для работы с файлами и каталогами в Linux, в том числе с некоторыми специальными типами файлов, о которых, возможно, даже не слышали. Вы также почувствуете всю мощь поиска по файлам и по их содержимому: это одна из самых впечатляющих возможностей Linux, которой стоит овладеть каждому разработчику.

Глава 6 «Как редактировать файлы из командной строки» посвящена двум текстовым редакторам с интерфейсом командной оболочки — nano и Vim. Вы освоите их на уровне уверенного пользователя, а также узнаете, какие ошибки чаще всего возникают при редактировании файлов и как их избежать.

В главе 7 «Пользователи и группы» вы узнаете, что такое пользователи и группы, почему эти понятия лежат в основе модели безопасности Unix и как они позволяют регулировать доступ к таким ресурсам, как файлы и процессы. Затем вы научитесь практическим командам, с помощью которых можно создавать и редактировать пользователей и группы.

Глава 8 «Владельцы ресурсов и права доступа» опирается на материал о пользователях и группах из предыдущей главы и рассказывает, как управлять доступом к ресурсам в Linux. Чтобы узнать, как настраивать владение ресурсами и права доступа, вам предстоит изучить сведения о файлах из вывода команды `ls` в расширенном формате. На этом примере мы рассмотрим часто используемые права доступа к файлам и каталогам, которые встретятся вам в продакшен-среде Linux, а затем разберемся с командами, которые позволяют назначать владельцев файлов и регулировать доступ к ним.

В главе 9 «Как устанавливать программное обеспечение» пойдет речь о том, как устанавливать приложения в различных дистрибутивах Linux (а также в macOS). Сначала мы познакомимся с системами управления пакетами — предпочительным

механизмом для того, чтобы управлять установленным ПО. Вы овладеете необходимой теорией и практическими командами, которые понадобятся вам как разработчикам, чтобы взаимодействовать с этими системами. Затем мы рассмотрим некоторые другие методы установки приложений, такие как загрузка установочных сценариев и проверенная временем традиция энтузиастов Unix компилировать свое собственное программное обеспечение локально из исходных файлов (это не так страшно, как кажется!).

Глава 10 «Как настраивать конфигурацию приложений» продолжает тему предыдущей главы об установке ПО; она научит вас конфигурировать приложения в системе Linux. Вы узнаете, к каким каталогам и файлам большинство программ обращаются по умолчанию за своими настройками (эта схема называется иерархией конфигурации). Эти знания не только пригодятся для того, чтобы отлаживать код в ночь перед выпуском, но и ощутимо помогут вам создавать более качественные программы. Мы рассмотрим аргументы командной строки, переменные окружения и конфигурационные файлы, а также узнаем, как все перечисленное работает в таких нестандарных средах Linux, как контейнеры Docker. «На закуску» вас ждет небольшой дополнительный проект: вы научитесь превращать произвольные программы в полноценные службы `systemd`.

Глава 11 «Конвейеры и перенаправление ввода-вывода» познакомит вас с техникой, которую многие считают «убойной функцией» Unix: это возможность объединять существующие программы в собственные решения с помощью конвейеров. В начале главы вы освоите необходимую теорию и практические навыки, которые относятся к файловым дескрипторам и перенаправлению ввода-вывода, а затем начнете сами конструировать составные команды с помощью конвейеров. Вы овладеете несколькими важнейшими инструментами командной строки и практическими шаблонами конвейеров, которыми будете пользоваться долгие годы после того, как расстанетесь с этой книгой.

Глава 12 «Как автоматизировать задачи с помощью сценариев командной оболочки» — это интенсивный курс по написанию сценариев для Bash. Благодаря ему вы сможете не просто вводить отдельные команды в интерактивной оболочке, но и создавать сценарии. Эта книга адресована действующим разработчикам, поэтому мы ограничимся кратким введением в важнейшие возможности языка и не будем тратить время на то, чтобы заново объяснять основы программирования. Вы познакомитесь с синтаксисом Bash, освоите эффективные приемы написания сценариев и узнаете, каких распространенных подводных камней стоит избегать.

Глава 13 «Безопасный удаленный доступ по протоколу SSH» рассказывает о протоколе SSH и о том, какие инструменты есть в вашем распоряжении, чтобы с ним работать. Вы узнаете, как устроены **криптографические системы**

с открытым ключом, — каждому разработчику стоит в них разобраться, прежде чем создавать собственные ключи SSH и авторизовываться в удаленных системах по защищенному сетевому каналу. Опираясь на эти знания, вы попрактикуетесь в том, как копировать файлы по сети и создавать ситуативные прокси-узлы или конфигурации VPN с помощью SSH, а также увидите примеры многих других задач, в которых данные передаются по зашифрованному тунелю SSH.

В главе 14 «Как управлять версиями с помощью Git» вы узнаете, как использовать Git — инструмент, с которым вы, вероятно, уже хорошо знакомы, — из командной строки, а не из IDE или графического клиентского интерфейса. Мы бегло повторим теоретические основы Git, а затем перейдем к командам, которые понадобятся вам в командной оболочке. Мы рассмотрим два мощных приема, владение которыми окупится с лихвой, — бисекцию и перебазирование, — а затем поделимся полезными техниками и псевдонимами командной оболочки, которыми сами пользуемся. Наконец, в разделе «GitHub на коленке» представлен небольшой, но вполне полезный проект, на котором вы сможете потренироваться и задействовать все, что к этому моменту узнали о Linux.

Глава 15 «Контейнерная виртуализация приложений с помощью Docker» снабдит вас теоретическими основами и практическими навыками, которые помогут использовать Docker в задачах разработки. Вы узнаете, какие проблемы решает Docker, познакомитесь с наиболее важными понятиями контейнерной виртуализации и узнаете о соответствующих рабочих процессах и командах, которые вам пригодятся. А поскольку мы подходим к этой теме с точки зрения разработки ПО и в контексте Linux, вы также получите хорошее представление о том, как контейнерная виртуализация работает на внутреннем уровне и чем она отличается от виртуальных машин.

Глава 16 «Журналы приложений» посвящена протоколированию в Unix и Linux. Мы покажем, как (и где) регистрируются журнальные сообщения в большинстве современных систем Linux с помощью `systemd`, а также как работают более традиционные подходы (в реальной практике вы столкнетесь и с тем и с другим). Вы отточите практические навыки работы с командной строкой, находя и просматривая журналы, и узнаете немного о том, как ведутся журналы в более крупных инфраструктурах.

Глава 17 «Балансировка нагрузки и протокол HTTP» охватывает основы протокола HTTP, актуальные для разработчиков, с особым упором на сложности, с которыми вы столкнетесь, когда будете работать с HTTP в более крупных инфраструктурах. Мы развеем некоторые распространенные заблуждения о состояниях HTTP, заголовках HTTP и версиях HTTP и о том, как приложения должны их обрабатывать. Мы также рассмотрим, как в реальном мире устроены балансировка нагрузки и проксирование и почему из-за этих факторов устранять

неполадки в среде эксплуатации приходится совсем не так, как в среде разработки на вашем компьютере. Здесь пригодятся многие из навыков работы в Linux, которыми вы овладели к этому моменту. Вы также познакомитесь с `curl` — новым инструментом, который помогает отлаживать широкий спектр неполадок, относящихся к HTTP.

Как извлечь максимальную пользу из книги

Чтобы воспроизводить на своем компьютере все примеры из этой книги, вам понадобится командная оболочка Linux, которую можно открыть, например, в Ubuntu, если установить эту систему в виртуальной машине или запустить как контейнер Docker.

Можно обойтись более скромными средствами: например, в Windows есть встроенная подсистема WSL, а macOS сама относится к семейству Unix, так что в этих системах будут запускаться «из коробки» почти все практические команды, которые вы изучите в этой книге (кроме тех, которые помечены как специфические для Linux). Впрочем, если вы хотите получить наиболее всесторонний опыт, лучше использовать полноценную ОС Linux.

Чтобы извлечь максимальную пользу из этой книги, вам будет достаточно только базовых компьютерных навыков, которыми вы уже обладаете как разработчик ПО, — редактировать текст, работать с файлами и каталогами, иметь некоторое представление о том, что такое операционная система, устанавливать программы и использовать интегрированную среду разработки. Всему, что выходит за эти рамки, мы вас научим.

Условные обозначения

Ниже перечислены условные обозначения, которые используются в этой книге.

`Код в тексте` обозначает фрагменты кода в основном тексте, имена таблиц в базе данных, имена файлов и каталогов, расширения файлов, пути к файлам, фиктивные URL, пользовательский ввод и имена Twitter. Например:

Длинная форма флага `-f` — `--follow`, а флага `-u` — `--unit`.

Код в командной строке выделяется так:

```
/home/steve/Desktop# ls
anotherfile documents somefile.txt stuff
/home/steve/Desktop# cd documents/
/home/steve/Desktop/documents# ls
contract.txt
```

Полужирным шрифтом выделяются новые термины, важные слова или текст, который вы видите на экране. В частности, так обозначаются надписи в меню или диалоговых окнах. Например:

> Если файл помечен как исполняемый, Unix приложит все усилия, чтобы его выполнить. Это завершится либо успехом — если файл соответствует формату **ELF (Executable and Linkable Format**, пожалуй, самый популярный сейчас формат исполняемых файлов), — либо неудачей.

ПРИМЕЧАНИЕ

Так обозначаются предупреждения и важные примечания.

СОВЕТ

Так обозначаются советы и полезные приемы.

1

Как устроена командная строка

Прежде чем изучать команды Linux на практических примерах, вам стоит получить общее представление о том, как работает командная строка. Этому посвящена текущая глава.

Начинающим разработчикам она поможет овладеть элементарными навыками взаимодействия с командной строкой Linux. Но и более опытные специалисты узнают кое-что ценное, например, чем отличается командная оболочка от командной строки. Иногда эта разница играет важную роль!

В этой главе мы рассмотрим такие темы:

- Что такое интерфейс командной строки, или CLI.
- Как устроены команды.
- Как применяются аргументы команд и как они выглядят, когда вы вводите команды и когда обращаетесь к справочным страницам.
- Что такое командная оболочка и чем она отличается от командной строки.
- Какими основными правилами руководствуется оболочка, когда обрабатывает команды.

Для начала разберемся, что представляет собой интерфейс командной строки. Мы поговорим о том, как он работает, и рассмотрим простой практический пример.

Вначале был REPL

Что такое **интерфейс командной строки**, он же **CLI** (command-line interface)? Это интерактивная оболочка, которая работает в текстовом режиме и позволяет взаимодействовать с компьютером, для чего осуществляет такие операции:

1. Принимает то, что ввел пользователь.
2. Выполняет вычисления, обрабатывая введенные данные.
3. Выводит на экран тот или иной результат.
4. Повторяет эту последовательность операций с самого начала.

Давайте разберем каждый шаг этой процедуры на примере конкретной команды `ls`, которую мы подробно рассмотрим далее в этой главе. Пока достаточно знать, что эта команда перечисляет содержимое каталога.

	Название операции	**Буквальный перевод**	**Что делает командная оболочка**	**Описание**
1	Read	Чтение	Принимает ввод	Вы вводите команду `ls` и нажимаете Enter
2	Evaluate	Вычисление	Обрабатывает команду	Командная оболочка ищет двоичный код команды `ls`, находит его и поручает компьютеру его выполнить
3	Print	Печать	Выводит результат	Команда `ls` производит текстовые данные — имена всех файлов и каталогов, которые она нашла. Затем командная оболочка выводит эти данные в окно терминала
4	Loop	Цикл	Переходит к шагу 1 (повторяет весь процесс)	После того как программа, которую вызвала команда, завершает работу, командная оболочка снова готова принимать данные от пользователя, и последовательность операций повторяется сначала

Цикл такого рода обозначается аббревиатурой REPL — по буквам, с которых начинаются названия четырех перечисленных операций в английском языке (Read — Eval — Print Loop[1]). Эта аббревиатура пришла из языков программирования, которые ввели в обиход и отладили соответствующую процедуру, таких как Lisp.

Цикл REPL можно записать в формате, более привычном для разработчиков, — на псевдокоде:

```
пока (истина) { // бесконечный цикл
    печать(вычисление(чтение()))
}
```

[1] Цикл «*чтение — вычисление — печать*». — *Примеч. пер.*

На самом деле в большинстве языков программирования достаточно всего нескольких строчек кода, чтобы создать REPL, который сможет выполнять элементарные вычисления. Например, вот однострочная программа командной оболочки, написанная на языке Perl:

```
perl -e 'while (<>){print eval, "\n"}'
1 + 2
3
```

Здесь код на Perl передается как параметр, а результаты вычислений выводятся в консоль в ответ на каждую строку, которую вводит пользователь. В конце вычислений программа выводит новую строку и завершает работу.

Это совсем крохотная программа, но ее достаточно для того, чтобы реализовать интерактивный цикл REPL в окружении командной строки — в **командной оболочке**. Оболочки, с которыми вы будете иметь дело в Linux и Unix, устроены гораздо сложнее, чем эта миниатюрная оболочка на Perl, но они работают по тому же принципу.

Вообще говоря, если вы разрабатываете ПО, то скорее всего, уже используете REPL, даже если не подозреваете об этом, потому что этот механизм поставляется почти со всеми современными языками сценариев. По сути, функции командной строки в Linux (а также в macOS или любой другой системе семейства Unix) работают так же, как «интерактивные оболочки» в интерпретируемых языках. Так что даже если вы не знакомы с REPL языка Lisp, то вышеприведенный сценарий на Perl наверняка заставит вас вспомнить простейшие оболочки для Ruby или Python.

Теперь, когда вы понимаете основы того, как работает интерфейс командной строки, с которым вы будете иметь дело в Linux, давайте попробуем применить первые команды. Для этого нужно знать, как устроен их синтаксис и как правильно использовать его в командной строке.

Синтаксис командной строки (чтение)

Работа любого REPL начинается с того, что он считывает данные, которые ввел пользователь. Чтобы оболочка принимала команды из командной строки Linux, они должны быть синтаксически правильными. Общая форма команды такова:

```
имя_команды параметры
```

С точки зрения программирования имя команды можно понимать как имя функции, а параметры — это набор из произвольного количества аргументов, которые ей передаются. Обратите внимание, что не существует универсального синтаксиса

са для любых параметров: каждая команда по-своему задает множество параметров, которые она готова принимать. Поэтому оболочка в общем случае не умеет проверять, правильно ли указаны параметры команды; она может только убедиться, что тому или иному имени команды соответствует исполняемый код.

ПРИМЕЧАНИЕ

В этой главе термины «программа» (program) и «команда» (command) употребляются как синонимы. Разница между ними незначительна: некоторые встроенные функции командной оболочки определены в ее коде и поэтому формально не являются отдельными программами. Но вам незачем вдаваться в эти подробности; оставьте такие нюансы ветеранам Unix.

Давайте посмотрим на более развернутый пример синтаксиса параметров команды, с которым вы будете часто иметь дело:

```
команда [-флаги,] [--example=foobar] [прочие параметры ...]
```

Этот общепринятый формат будет встречаться вам в документации, например, на справочных страницах (man-страницах), которые включены в большинство сред Linux, и этот синтаксис довольно прост:

- `команда` — это программа, которую вы запускаете;
- параметры в квадратных скобках не обязательны, а скобки с многоточием (например, `[xyz ...]`) обозначают, что в этом месте можно передать ноль или более аргументов;
- `-флаги` — это любые допустимые параметры программы (в Unix они называются флагами), например `-debug` или `-foobar`[1].

Некоторые программы также принимают короткие и длинные формы одних и тех же параметров: короткая форма начинается с одного дефиса, а длинная — с двух. (Например, параметры `-l` и `--long` могут обозначать одно и то же.) Так ведут себя не все команды, а только те, создатели которых позаботились о том, чтобы одни и те же параметры можно было задавать и в короткой, и в длинной форме.

Не все команды позволяют при вызове передавать настройки по такой схеме, но она охватывает самые распространенные случаи, которые вам встретятся.

---

[1] `foobar`, а также `foo`, `bar`, `baz` и некоторые другие похожие английские слова широко используются в компьютерной документации как универсальные имена-заместители. Они будут часто встречаться далее в книге. — *Примеч. науч. ред.*

По умолчанию пробел означает конец аргумента. Поэтому, как и в большинстве языков программирования, если в качестве аргумента передается строка, которая содержит пробелы, то ее нужно заключать в одинарные или двойные кавычки. Подробнее об этом вы прочтете в главе 12 «Как автоматизировать задачи с помощью сценариев командной оболочки».

Совсем скоро мы подробно проследим, как оболочка интерпретирует команду, которую вы вводите с помощью этого синтаксиса. Но сначала давайте четко обозначим разницу между двумя терминами, которые мы используем в этой главе и которые иногда употребляются как взаимозаменяемые: «командная строка» и «командная оболочка».

Командная строка и командная оболочка

Под **командной строкой** (command line) или **окружением командной строки** (command-line environment) в этой книге мы понимаем любую среду, которая работает в текстовом режиме и ведет себя как цикл REPL, позволяя взаимодействовать с операционной системой, интерпретатором языка программирования, базой данных и т. д. Интерфейсом командной строки называется общий принцип коммуникации с системой.

Но мы будем употреблять и более узкий термин — «командная оболочка».

Командная оболочка, **командный интерпретатор** или просто **оболочка** (shell) — это конкретная программа, которая реализует интерфейс командной строки и позволяет вводить команды в текстовом режиме. Существует много разных оболочек, которые обеспечивают похожее окружение командной строки в режиме REPL, но могут выполнять совершенно разные задачи, например:

- Bash[1] — это широко распространенная оболочка для взаимодействия с Linux и другими операционными системами семейства Unix.
- Все популярные системы управления базами данных, такие как PostgreSQL, MySQL и Redis, оснащены командной оболочкой для разработчиков, с помощью которой можно взаимодействовать с базами данных, запуская команды.
- Большинство интерпретируемых языков программирования поставляются с командными оболочками, которые позволяют ускорить разработку. В таких

[1] В названиях популярных оболочек (Bash, Zsh, tsch, fish и др.) буквы *sh* — сокр. *shell* (*командная оболочка*). Bash расшифровывается как *Bourne Again Shell* — это игра слов, которая подчеркивает как преемственность с классической оболочкой Bourne Shell, так и идею перерождения (*bourne again* созвучно с *born again* — *рожденный заново*). В свою очередь, Bourne Shell (также известная как просто sh) была названа по имени своего создателя Стивена Борна (Stephen Bourne). — *Примеч. пер.*

оболочках допустимые команды — это просто инструкции соответствующего языка. Возможно, вы встречали оболочку `irb` для Ruby, интерактивную оболочку для Python или другие подобные инструменты.

- Zsh — это альтернативная командная оболочка для операционных систем, похожая на Bash. Ее можно встретить на компьютерах тех разработчиков, кто специально настроил свою рабочую среду[1].

В этой книге под *оболочкой* мы будем понимать командную оболочку Unix (обычно это Bash) с интерфейсом командной строки, которая позволяет взаимодействовать с Linux или другой нижележащей операционной системой семейства Unix.

Каким образом оболочка узнаёт, какую программу запускать (вычисление)

После того, как командная оболочка *прочитала* команду, она должна *вычислить* ее, то есть выполнить ту или иную программу, предоставить определенную информацию или сделать еще что-то полезное.

ПРИМЕЧАНИЕ

Последующее чрезвычайно подробное описание того, как работает оболочка, может поначалу показаться занудным. Но мы обещаем, что если вы освоите эти подробности, то вам будет гораздо легче устранять неполадки в ситуации, когда нужной программы нет или когда для нее заданы неправильные права доступа.

Когда вы набираете в командной оболочке (например, Bash) команду вроде `foobar -option1 test.txt` и нажимаете Enter, происходит вот что:

1. Если для команды задан путь к файлу, оболочка обращается по этому пути. Путь может принимать разные формы:
   - Абсолютный путь — например, `/usr/bin/foobar` в команде `/usr/bin/foobar -option1 test.txt`.
   - Относительный путь — например, текущий рабочий каталог в команде `./foobar -option1 test.txt`. (Точка в начале команды обозначает текущий каталог; мы подробнее поговорим об этом позже в разделе «Абсолютные и относительные пути к файлам». По сути, эта команда означает «выполнить файл `foobar`, который находится в текущем каталоге».)

[1] Zsh является системной оболочкой по умолчанию в macOS, начиная с версии macOS Catalina (2019), а также в некоторых современных дистрибутивах Linux (например, Kali Linux). — *Примеч. науч. ред.*

- Путь может содержать особые конструкции двух типов:
  - переменные окружения — например, `$HOME` в пути `$HOME/foobar`;
  - специальные символы, которые поддерживает оболочка, — например, тильду (`~`) в пути `~/foobar` (символ `~` обозначает домашний каталог текущего пользователя).

2. Если путь к файлу не задан, оболочка проверяет, нет ли у нее информации о том, что такое `foobar`. Например, это может быть:
   - встроенная команда оболочки;
   - псевдоним выполнения, с помощью которого можно настраивать макросы или сокращенные имена команд.
3. Если ничего из этого не подошло, то оболочка обычно обращается к переменной окружения `$PATH`, которая перечисляет несколько каталогов, где следует искать команды: `/bin`, `/usr/bin`, `/sbin` и т. д. Этот перечень могут изменять как пользователи, так и различные программные продукты: диспетчеры версий для языков сценариев, виртуальные окружения для Python и многие другие программы. Оболочка просматривает каталоги в том порядке, в каком они следуют в переменной `$PATH`, пока не найдет в одном из них исполняемый файл с именем `foobar`.

Если после всего этого оболочка так ничего и не нашла, она возвращает ошибку вроде `-:foobar: command not found` [1].

С другой стороны, если на каком-нибудь этапе этой процедуры оболочка находит исполняемый файл с именем `foobar`, она запускает этот файл и передает ему в качестве аргументов `-option1` и `test.txt` (именно в таком порядке).

В этом случае оболочка знает, какая программа нужна, и запускает именно ее. По мере того как команда вычисляется, ее результаты выводятся на экран, и таким образом завершается третья стадия цикла REPL (печать). После этого командной оболочке остается только вернуться в начало цикла и повторить всю процедуру заново, приняв следующую команду от пользователя.

Процесс, который мы рассмотрели, помогает командной оболочке наилучшим образом угадать, какую программу пользователь хочет запустить, и избежать неоднозначности. Неоднозначность может вызывать проблемы и приводить к недоразумениям и ошибкам. Устраняя неполадки, часто требуется выяснить, какая команда на самом деле запущена. Для этого служит команда `which` *`имя_команды`*, которая выводит полный путь к исполняемому файлу команды и позволяет узнать, является ли она встроенной командой оболочки. В некоторых системах нет команды `which`, зато можно использовать более универсальную `command -v` — это экви-

[1] Команда foobar не найдена. — *Примеч. пер.*

валентная команда в соответствии со стандартом POSIX, о котором мы скоро поговорим[1]:

```
bash-3.2$ which ls
/bin/ls
bash-3.2$ command -v ls
/bin/ls
```

Краткое введение в POSIX

«Википедия» гласит, что **POSIX**[2] — это «семейство стандартов, утвержденное Компьютерным обществом IEEE для того, чтобы обеспечить совместимость между операционными системами». На практике это попытка определить некоторые общие стандарты, которым должны следовать системы семейства Unix, чтобы не получалось так, что в разных системах приняты совершенно разные наборы элементарных команд.

По сути, POSIX содержит требования такого рода: «В каждой системе, совместимой с POSIX, должна быть команда, которая перечисляет содержимое каталога и называется `ls`». В случае с `command -v` требование звучит так: «Каждая система, совместимая с POSIX, должна обеспечивать способ проверить, существует ли исполняемый файл, который соответствует заданному имени команды».

Если вы хотите, чтобы ваши сценарии были переносимы между разными операционными системами семейства Unix, имеет смысл ограничиться только командами, которые соответствуют POSIX. Но это не стопроцентная гарантия: многие широко распространенные дистрибутивы Linux отклоняются от POSIX в различных аспектах, причем некоторые отклонения трудно обнаружить, пока они не причинят вам неприятности.

---

[1] В этом и последующих листингах `bash-3.2$` и подобные конструкции — это *приглашение командной строки* (*prompt*). Когда оболочка готова принимать команды, она обычно отображает приглашение, а после него — мигающий курсор, который обозначает позицию ввода. Приглашение может принимать разные формы в зависимости от операционной системы, командной оболочки и пользовательских настроек: например, оно может содержать название и версию оболочки (как в этом примере), имя пользователя и компьютера, текущий каталог, специальные символы вроде → и т. д. Иногда приглашение может оформляться иначе, чем вводимые команды, например выделяться другим цветом или курсивом. Символ `$` в конце приглашения означает, что командная оболочка запущена с правами обычного пользователя (не администратора).

В листингах в этой книге фигурирует командная оболочка Bash 3.2, хотя на момент выхода англоязычного издания, а также на момент подготовки этого перевода наиболее свежей версией является Bash 5.2, выпущенная в 2022 году. — *Примеч. науч. ред.*

[2] Portable Operating System Interface (переносимый интерфейс операционной системы). — *Примеч. пер.*

POSIX — это последняя тема, которую нам нужно было затронуть, прежде чем начать на практике работать с командной строкой. К этому моменту мы с вами уже рассмотрели немало основных понятий:

- Вы узнали о том, как устроен цикл REPL («чтение — выполнение — печать») и как он воплощается в современных командных оболочках.
- Вы познакомились с общим синтаксисом команд, который вам предстоит использовать, работая в Linux.
- Вы разобрались, каким образом командная оболочка определяет, как принять вашу команду и правильно «вычислить» ее.
- Вы изучили важные понятия, которые будут часто встречаться на протяжении книги: «командная оболочка», «интерфейс командной строки», POSIX, а также еще несколько тем, владение которыми облегчит вашу профессиональную деятельность.

После того как вы освоили эти основы, можно перейти от теории к практике. В следующем разделе мы поговорим о специфическом для Linux контексте, в котором вы будете находиться, когда запускаете команды. Вы получите необходимые начальные сведения о файловой системе Linux и о том, как обращаться с разными типами путей к файлам и каталогам. После этого вся оставшаяся часть главы будет посвящена тому, как запускать различные команды Linux.

Элементарные навыки работы в командной строке

Чтобы эффективно работать с Linux, следует овладеть элементарными знаниями: как устроена файловая система, как просматривать ее структуру и перемещаться по ней, как просматривать и редактировать файлы. Все это мы рассмотрим в данном разделе, после чего вы будете уверенно ориентироваться в структуре файлов и каталогов в Linux.

Далее в книге мы подробнее углубимся в каждую тему и каждую команду, но сейчас важно, чтобы к концу главы вы получили минимальный набор практических навыков, которые можно применять в профессиональной деятельности.

Введение в файловую систему Unix

В графических пользовательских интерфейсах **каталоги** (в macOS они называются **папками**) представляются в виде значков. Возможно, вы привыкли наблюдать в своем домашнем каталоге ряды этих пиктограмм: «Рабочий стол», «Документы», «Видео» и т. д. Если дважды щелкнуть по значку каталога, откроется новое окно, в котором отображается содержимое этого каталога.

Когда мы говорим «файловая система», то имеем в виду именно это — совокупность каталогов и файлов, в которых содержатся все данные, существующие в системе. Интерфейс командной строки опирается на то же самое понятие файловой системы, просто она представлена немного по-другому.

В командной строке вы не увидите многочисленных окон и значков: все представляется в виде текста, а содержимое каталогов отображается только тогда, когда вы его запросите. Однако файлы и каталоги по-прежнему ведут себя привычным образом.

Поначалу может показаться сложным держать файловую систему в голове, когда вы работаете с файлами и каталогами, но когда вы к этому привыкнете, то убедитесь, что такой подход часто оказывается более эффективным, чем щелкать по значкам. Большинство пользователей, которые поработали в таком режиме хотя бы несколько дней, без труда удерживают в памяти подробное представление файловой системы, и им редко требуется его уточнять.

Абсолютные и относительные пути к файлам

Новичков в Linux часто сбивает с толку разница между **абсолютными** и **относительными** путями к файлам и каталогам. Если ее не уяснить, можно впустую потратить много времени, разглядывая сообщения об ошибках вроде такого:

```
No such file or directory[1]
```

Чтобы правильно запустить почти любую команду Linux, нужно понимать, как устроены пути к файлам, поэтому мы прежде всего остановимся на этом вопросе.

Абсолютный путь — это полный путь от корневого каталога до того или иного файла в файловой системе. Этот путь начинается с прямого слеша (`/`), обозначающего **корневой каталог** — каталог верхнего уровня, который лежит в основе файловой системы и содержит все остальные файлы и каталоги.

Вот несколько примеров абсолютных путей:

- `/home/dave/Desktop`
- `/var/lib/floobkit/`
- `/usr/bin/sudo`

Эти **абсолютные пути** подобны полному маршруту в навигаторе, который ведет вас по всем дорожным развязкам от известной отправной точки, например от вашего дома или, в случае системы семейства Unix, от корневого каталога.

---

[1] Такого файла или каталога не существует. — *Примеч. пер.*

Абсолютный путь можно однозначно узнать по тому, что он начинается с прямого слеша (/). Абсолютный путь работает одинаково независимо от того, в каком месте файловой системы вы находитесь, потому что он представляет собой полный и уникальный адрес файлового объекта.

Относительный путь — это частичный путь, и предполагается, что он отталкивается не от корневого, а от *текущего каталога*. Относительный путь отличается тем, что он *не* начинается со слеша.

Относительный путь — это как маршрут от того места, где вы сейчас находитесь. Если вы съехали на обочину, потому что заблудились и хотите уточнить дорогу, то вас интересует маршрут от вашего *текущего местоположения*, а не от дома. Это и есть относительный путь.

В результате относительные пути часто удобнее вводить, чем абсолютные. Например, если вы уже находитесь в каталоге `/home/Desktop`, то к файлу `mydocument.txt` проще обратиться по этому имени, чем по полному пути `/home/Desktop/mydocument.txt` (хотя если вы находитесь в указанном каталоге, то оба пути одинаково действительны). Разница между этими путями проявится, когда вы перейдете в другой каталог. Если переместиться на один уровень вверх — из каталога `/home/Desktop` в каталог `/home`, — то абсолютный путь будет попрежнему вести к тому же самому файлу, а относительный — нет. (В нашем случае, если теперь ввести `mydocument.txt`, это будет обозначать файл `/home/mydocument.txt`.)

Для примера рассмотрим фрагмент файловой системы. Предположим, что так выглядит структура каталогов и файлов внутри каталога `/home/dave/Desktop`:

```
Desktop                  # Рабочий стол[1]
├── anotherfile          # ├── другой файл
├── documents            # ├── документы
│   └── contract.txt     # │   └── договор
├── somefile.txt         # ├── какой-то файл
└── stuff                # └── разное
    ├── nothing          #     ├── неважно
    └── important        #     └── важно
```

Допустим, вы находитесь в каталоге `Desktop` («Рабочий стол»); иными словами, ваш текущий каталог — `/home/dave/Desktop` (этот путь можно просмотреть командой `pwd`).

Вот несколько примеров относительных и абсолютных путей к одним и тем же файлам в этом каталоге:

[1] Приблизительные эквивалентные имена файлов и каталогов на русском языке приведены для тех читателей, которым это поможет лучше понять пример. — *Примеч. пер.*

Относительный путь	Абсолютный путь
`anotherfile`	`/home/dave/Desktop/anotherfile`
`documents/contract.txt`	`/home/dave/Desktop/documents/contract.txt`
`stuff/important`	`/home/dave/Desktop/stuff/important`

Обратите внимание, что относительный путь — это то же самое, что абсолютный путь, из начала которого отсечена часть, ведущая к текущему рабочему каталогу.

Еще об абсолютных и относительных путях

Продолжая рассматривать наш фрагмент файловой системы, представьте, что вы работаете в командной оболочке и ваш текущий каталог — `Desktop`. Допустим, вам нужно вывести сведения о файле `contract.txt`, который находится в папке `documents`. Как обратиться к этому файлу? Есть два способа:

- `ls /home/dave/Desktop/documents/contract.txt` — абсолютный путь, который одинаково работает из любого каталога.
- `ls documents/contract.txt` — относительный путь, то есть путь относительно текущего каталога.

Как открыть командную оболочку

Чтобы в Ubuntu Linux или macOS открыть командную оболочку с интерфейсом командной строки, запустите приложение «Терминал» (Terminal).

Сведения о файлах и каталогах

Если вы еще плохо знакомы с командной оболочкой, то первое, что стоит сделать, когда вы ее открыли, — осмотреться в системе. В этом разделе мы перечислим важнейшие команды Linux, нужные для того, чтобы ориентироваться в файлах и каталогах и выводить сведения о них.

pwd — текущий каталог

Когда вы запускаете команду `pwd`[1] в терминале, оболочка выводит абсолютный путь к каталогу, в котором вы находитесь. Файловую систему Unix часто ассоциируют с деревом, но пока можно рассматривать ее как захламленный рабочий стол со множеством каталогов. Если представлять себе каждый каталог как комнату, то команда `pwd` позволяет узнать, в какой комнате сейчас находится ваше окружение командной строки[2].

---

[1] Аббр. *print working directory* (*вывести на печать рабочий каталог*). — *Примеч. пер.*

[2] Имена команд в оболочках Linux и других систем семейства Unix зависят от регистра. То есть чтобы получить имя текущего каталога, нужно ввести именно `pwd`, а не `Pwd` или `PWD`. — *Примеч. науч. ред.*

Новые сеансы командной оболочки обычно начинаются в вашем домашнем каталоге. Если вы запускаете команды в Linux, то увидите нечто подобное:

```
→ ~ pwd
/home/dave
```

В других системах семейства Unix вывод может выглядеть немного иначе. Например, вот что отображается в macOS:

```
→ ~ pwd
/Users/dave
```

Независимо от того, в каком месте файловой системы вы находитесь, можно обращаться к любому файлу в любом каталоге (см. раздел «Абсолютные и относительные пути к файлам» ранее в этой главе). Однако иногда удобнее перемещаться между каталогами. Далее в главе 5 мы подробнее рассмотрим, как устроена файловая система.

ls — содержимое каталога

Команда `ls`[1] выводит список файлов в заданном каталоге. Если запустить ее без аргументов, она просто перечисляет файлы и каталоги, которые находятся в текущем каталоге. Если в качестве аргумента передать путь к конкретному каталогу, то `ls` выведет его содержимое:

```
ls /var/log
```

Команда `ls` может принимать и другие аргументы — так называемые флаги. Их бывает много, но два самых распространенных — это `-l`[2] (подробные сведения) и `-h`[3] (размер файлов в «человекочитаемом» формате).

```
ls -l -h

#  Тот же результат; флаги можно объединять
ls -lh

# Выводит содержимое указанного каталога
ls -lh /usr/local/
```

Команда `ls` с флагом `-l` выводит результат в таком формате:

```
-rw-r--r-- 1 dcohen wheel 0 Jul   5 09:27 foobar.txt
```

---

[1] Сокр. *list* (*перечислить, вывести список*). — *Примеч. пер.*

[2] Название флага `-l` происходит от long (длинный формат вывода). В отличие от большинства флагов, у него нет длинной формы (наподобие `--long` или чего-то похожего). — *Примеч. науч. ред.*

[3] Короткому флагу `-h` соответствует эквивалентная длинная форма `--human-readable` (человекочитаемый формат). — *Примеч. науч. ред.*

Давайте разберем все поля этого вывода:

`-rw-r--r--`	Первый символ — тип файла, остальные — права доступа к нему: три группы по три бита, которые представляют права владельца, группы владельца и всех остальных пользователей системы соответственно
`1`	Количество жестких ссылок, которые ведут на этот файл
`dcohen`	Имя пользователя, который владеет файлом
`wheel`	Имя группы, которой принадлежит файл
`0`	Объем дискового пространства, которое занимает файл (в этом примере файл пуст). По умолчанию значения выражаются в байтах, но с флагом `-h` они выводятся в «человекочитаемом» формате, то есть в килобайтах, мегабайтах или гигабайтах, там, где это уместно
`Jul 5 09:27`	Дата и время последнего изменения файла
`foobar.txt`	Имя файла

В этом выводе фигурируют понятия, которые мы еще не рассматривали, — пользователи, группы и права доступа. Не волнуйтесь, мы вникнем во все это в главе 7 «Пользователи и группы».

Навигация по файловой системе

После того как вы изучили простейшие команды Linux, с помощью которых можно ориентироваться в файловой системе, давайте поговорим о том, как перемещаться по ней и находить нужные объекты.

cd — сменить текущий каталог

Команда `cd`[1] позволяет сменить текущий каталог на любой другой из тех, которые есть в файловой системе. Если продолжать аналогию с комнатами, то эта команда моментально телепортирует вас из текущей комнаты в какую-то другую.

После того как вы успешно смените каталог, команда `pwd` покажет ваше новое местоположение[2]:

```
# Перейти в каталог /etc/ssl
```

[1] Аббр. *change directory* (*сменить каталог*). — *Примеч. пер.*

[2] В оболочках систем семейства Unix команды, которые модифицируют состояние системы (например, меняют текущий каталог или копируют файлы), по умолчанию *не* выводят в командную строку никаких результатов, если операция завершилась успешно. Поэтому в этом примере сама по себе команда `cd` не сопровождается никаким выводом. — *Примеч. науч. ред.*

```
bash-3.2$ cd /etc/ssl

bash-3.2$ pwd
/etc/ssl

bash-3.2$ ls
README cert.pem certs misc openssl.cnf private

# Перейти в каталог /etc/ssl/certs
# Обратите внимание на относительный путь
bash-3.2$ cd certs

bash-3.2$ pwd
/etc/ssl/certs
```

find — поиск файлов

Команда `find`[1] предназначена для поиска файлов. Это одна из немногих команд, для которой не действует соглашение о том, что параметры в длинной форме начинаются с двух дефисов (например, `--name`). Флаги команды `find` начинаются с одного дефиса, например:

```
bash-3.2$ find / -type d -name home
/home
…
```

Эта команда ищет в каталоге `/` (то есть во всей файловой системе) каталоги (`-type d`), которые называются `home`. Учтите, что если вы запускаете `find` не с правами всемогущего администратора (пользователя `root`), то у команды не хватит полномочий, чтобы обратиться к содержимому многих каталогов. Поэтому, кроме списка найденных каталогов, вы увидите много сообщений вроде `find: '/root': Permission denied`[2].

Еще одна распространенная практика — запускать другие команды на основе того, что вывела `find`, например:

```
bash-3.2$ find . -exec echo {} \;
.
./foobar
```

Эта команда запускает команду `echo` для каждого имени файла, найденного в текущем каталоге (`.`), подставляя это имя вместо фигурных скобок (`{}`). Результат весьма похож на то, что выводит команда `ls`.

---

[1] *Найти. — Примеч. пер.*

[2] Команде `find` отказано в доступе к каталогу `/root`. — *Примеч. пер.*

Если нужно не запускать отдельную команду `echo` для каждого найденного файла, а передать все имена файлов в качестве аргументов одной команде `echo`, то обратный слеш (`\`) следует заменить на плюс (`+`):

```
bash-3.2$ find . -exec echo {} +;
. ./foobar
```

У `find` есть много других флагов — их конкретный набор зависит от того, какая версия этой команды установлена в операционной системе.

Вот несколько типичных примеров:

Команда	**Что ищет эта команда**
`find -iname foobar`	Файлы с именем `foobar` без учета регистра
`find -name "foobar*"`	Файлы, имя которых начинается с `foobar`
`find -name "*foobar"`	Файлы, имя которых оканчивается на `foobar`

Как просматривать содержимое файлов?

После того как вы научились находить нужные файлы, давайте посмотрим, как из командной строки просматривать их содержимое.

less — постраничный просмотр файла

Команда `less`[1] позволяет просматривать содержимое файла, отображая в терминале по одной «странице» в каждый момент времени (размер «страницы» зависит от размера окна терминала).

```
less имя_файла
```

Если запустить `less`, то в командной оболочке отобразится содержимое файла и вы сможете прокручивать его построчно (клавишами со стрелками ↑ и ↓) или постранично (вниз — клавишей пробела, вверх — клавишей `B`).

---

[1] *Less* дословно означает «*меньше*». Команду назвали таким образом в качестве игры слов, потому что она задумывалась как расширенная версия классической команды `more` («*больше*»). В отличие от своей предшественницы, `less` умела прокручивать текст не только вниз, но и вверх. В свою очередь, создатель команды `more` Дэн Хэлберт назвал ее так потому, что она выводила `--More--`, если содержимое файла не умещалось на экране. Это было шагом вперед по сравнению с командами еще более старшего поколения, которые в этом случае подавали звуковой сигнал. — *Примеч. науч. ред.*

Чтобы искать текст в файле, нажмите /[1], затем введите искомый текст и нажмите Enter. Перейти к следующему вхождению искомого текста можно клавишей N[2], а к предыдущему — Shift+N.

Чтобы выйти из режима просмотра файла, нажмите Q[3].

Как вносить изменения в файлы

Вы уже умеете находить файлы и просматривать их содержимое, так что теперь давайте научимся редактировать существующие файлы и создавать новые.

touch — создать пустой файл или обновить метки времени для существующего файла

Команда `touch` создает файл, поэтому в качестве аргумента ей нужно передать путь к этому файлу. Если файла с таким путем не существует, то команда создает пустой файл (если у вас есть нужные права).

Если файл по указанному пути уже есть, то дата и время его изменения и последнего доступа обновляются до текущего времени. Если вам нужно изменить только время доступа или только время изменения, используйте флаги `-a` или `-m` соответственно[4].

mkdir — создать каталог

Команда `mkdir`[5] создает каталог в соответствии с путем, который передан в качестве аргумента, например:

---

[1] Командная оболочка реагирует на то, какие текстовые символы вы вводите, а не на то, какие клавиши нажимаете. Например, если у вас включена стандартная русская раскладка клавиатуры (допустим, вы с помощью `less` хотите искать текст в файле на русском языке), то символ / понадобится вводить не той же клавишей, что в английской раскладке, а сочетанием клавиш Shift+\. — *Примеч. науч. ред.*

[2] От *next* (*следующий*). — *Примеч. пер.*

[3] От *quit* (*выйти*). Клавиша Q позволяет выйти из режима просмотра или редактирования не только в `less`, но и во многих других командах. — *Примеч. пер.*

[4] Основное назначение команды `touch` («*дотронуться*», «*прикоснуться*») первоначально состояло в том, чтобы с минимальными затратами ресурсов модифицировать время изменения файла и/или доступа к нему: это было полезно для процессов разработки, резервного копирования, управления версиями и других технических процедур. Ключи `-a` и `-m` — это сокр. *access* (*доступ*) и *modification* (*изменение*) соответственно; у обоих ключей нет длинных форм. Создание новых файлов было побочным эффектом команды, но со временем превратилось в основной сценарий ее использования. Чтобы `touch` не создавала новых файлов, ее можно запустить с ключом `-c` (или `--no-create`, «*не создавать*»). — *Примеч. науч. ред.*

[5] Сокр. *make directory* (*создать каталог*). — *Примеч. пер.*

```
bash-3.2$ mkdir foobar
bash-3.2$ ls
foobar
```

Чтобы создать сразу несколько каталогов, можно передать команде их имена как дополнительные аргументы:

```
bash-3.2$ mkdir foo bar baz
bash-3.2$ ls
foo
bar
baz
```

А чтобы создать несколько каталогов, которые вложены друг в друга (или просто убедиться, что они существуют), можно использовать флаг `-p`[1]:

```
bash-3.2$ mkdir -p /var/log/myapp/foobar
bash-3.2$ ls /var/log/myapp
foobar
```

Даже если каталога `/var/log/myapp` раньше не было, команда `mkdir` с флагом `-p` обеспечит создание этого каталога до того, как внутри него будет создан каталог `/var/log/myapp/foobar`. С другой стороны, если каталог, который вы передаете команде `mkdir`, уже существует, то благодаря флагу `-p` команда никак его не повредит, так что команду `mkdir -p` с одним и тем же именем каталога можно запускать много раз подряд (такие команды называются *идемпотентными*). Поэтому флаг `-p` широко используется в сценариях.

rmdir — удалить пустой каталог

Команда `rmdir`[2] удаляет пустые каталоги. Она работает только с пустыми каталогами, поэтому запускать ее относительно безопасно. Впрочем, большинство пользователей Linux предпочитают команду `rm`, которая делает то же самое, но более универсальна.

rm — удалить файл или каталог

Чтобы удалить файл, вызовите команду `rm`[3]:

```
rm имя_файла
```

На практике большинство пользователей применяют `rm` также для того, чтобы удалять каталоги, потому что, в отличие от `rmdir`, эта команда работает и с не-

---

[1] Длинная форма — `--parents` (*родительские каталоги*). — *Примеч. пер.*

[2] Сокр. *remove directory* (*удалить каталог*). — *Примеч. пер.*

[3] Сокр. *remove* (*удалить*). — *Примеч. пер.*

пустыми каталогами. Чтобы выполнить команду *рекурсивно* (то есть для всех каталогов, которые находятся внутри удаляемого), вам понадобится флаг `-r`, а чтобы принудительно удалить все файлы и каталоги, не запрашивая подтверждения, — флаг `-f`[1]:

```
rm -rf /путь/к/каталогу
```

ПРИМЕЧАНИЕ

С командой `rm -rf` нужно обращаться крайне осторожно, потому что с ее помощью Linux позволяет удалять в том числе те каталоги, без которых вся система не сможет работать. Например, `rm -rf /` обозначает, что нужно удалить корневой каталог, который содержит вообще все, что есть в системе.

Некоторые дистрибутивы Linux и другие системы семейства Unix предлагают изобретательные подходы к этой проблеме. Например, в Ubuntu у команды `rm` есть параметр `--no-preserve-root`[2], который фактически спрашивает: «Вы *действительно* этого хотите?». Solaris использует облегченную интерпретацию правил о том, как должна вести себя команда `rm`, чтобы не удалять корневой каталог. Впрочем, на практике эти предосторожности легко обойти. Будьте бдительны, когда запускаете `rm`, и не вставляйте бездумно в свою оболочку команды, которые вы нашли в интернете!

mv — переместить или переименовать файл или каталог

Команда `mv`[3] устроена довольно хитроумно: она умеет выполнять две разные операции с помощью одного и того же кода. Эта команда позволяет либо переместить файлы из одного каталога в другой, либо переименовать файл, оставив его в том же каталоге.

Давайте сначала с помощью `touch` создадим файл `foobar.txt`:

```
bash-3.2$ touch foobar.txt
bash-3.2$ ls
foobar.txt
```

---

[1] Длинные формы флагов `-r` и `-f` соответственно `--recursive` (*рекурсивно*) и `--force` (*принудительно*). Для рекурсивного удаления также служит флаг `-R`. — *Примеч. науч. ред.*

[2] «*Не предохранять корневой каталог*». Без этого параметра команда `rm` не будет удалять корневой каталог. Отметим, что стандарт POSIX, начиная с версии 7 (2018), явно запрещает команде `rm` удалять корневой каталог. — *Примеч. науч. ред.*

[3] *Move* (*переместить*). — *Примеч. пер.*

Затем переименуем этот файл в `foobarbaz.txt` на месте:

```
bash-3.2$ mv foobar.txt foobarbaz.txt
bash-3.2$ ls
foobarbaz.txt
```

Учтите, что если файл `foobarbaz.txt` уже существовал, эта команда его перезапишет, так что будьте осторожны с переименованием.

Чтобы переместить файл в другой каталог, мы создадим этот каталог, а затем перенесем туда файл:

```
bash-3.2$ mkdir targetdir
bash-3.2$ mv foobarbaz.txt targetdir/
bash-3.2$ ls targetdir/
foobarbaz.txt
```

Обе операции можно совмещать. Вот как можно поступить, если вы хотите переместить файл в новый каталог *и* одновременно с этим переименовать этот файл:

```
bash-3.2$ mv foobarbaz.txt targetdir/renamed.txt
bash-3.2$ ls targetdir/
renamed.txt
```

Как пользоваться справочными страницами

Во всех командных оболочках, кроме самых минималистичных, есть встроенные справочные страницы (так называемые man-страницы) — официальная документация, с помощью которой можно научиться использовать программы, доступные из командной строки (или хотя бы вызубрить их).

Чтобы получить справку о команде, запустите `man`[1] *`имя_команды`*. Например, `man ls` выведет что-то наподобие этого[2]:

```
LS(1)                                                Команды общего назначения
Справочное руководство                               LS(1)

ИМЯ
```

[1] Сокр. *manual* (*справочное руководство*). — *Примеч. пер.*

[2] Справочные страницы в Linux и других системах семейства Unix выводятся на английском языке. Здесь и далее фрагменты справочных страниц приведены в русском переводе, чтобы читателю было проще получить представление о том, какого рода информация на них содержится.

Хотя во всех операционных системах семейства Unix команда `ls` выводит содержимое каталога, в разных системах ее поведение может различаться в деталях, и также различается содержание и формат справочных страниц. Этот пример в книге соответствует macOS и некоторым родственным системам. — *Примеч. науч. ред.*

```
     ls – перечисляет содержимое каталога

ОБЗОР
     ls [-@ABCFGHILOPRSTUWabcdefghiklmnopqrstuvwxy1%,] [--color=когда]
        [-D формат] [файл ...]

ОПИСАНИЕ
     Для каждого операнда, который соответствует файлу любого типа, кроме
     каталога, ls отображает его имя, а также любую связанную информацию,
     которая была запрошена. Для каждого операнда, который соответствует
     файлу типа каталога, ls отображает имена файлов, которые содержатся
     в этом каталоге, а также любую связанную информацию, которая была
     запрошена.

     Если операнды не предоставлены, отображается содержимое текущего каталога.
     Если предоставлено более одного операнда, то сначала обрабатываются
     операнды, которые не являются каталогами.  Операнды, которые являются
     каталогами и не являются ими, сортируются отдельно друг от друга
     и в лексикографическом порядке.

     Доступны следующие параметры:

     -@      Отобразить расширенные ключи атрибутов и размеры в длинном (-l)
             формате вывода.

     -A      Включить элементы, имена которых начинаются с точки ('.'),
             за исключением . и ...  Автоматически предоставить вывод
             для суперпользователя, если не указан флаг -I.
```

Справочные страницы автоматически открываются в приложении постраничного просмотра, поэтому для прокрутки, поиска и выхода можно использовать те же сочетания клавиш, которые уже знакомы вам по работе с командой `less`.

Имейте в виду, что `man` — это старая утилита, которая пытается имитировать традиционную книгу-справочник, где разные разделы (главы) посвящены разным темам. В предыдущем примере команда обозначалась как `ls(1)`, где цифра `1` в скобках указывала, в каком разделе справочника мы находимся.

Иногда бывает, что в разных разделах есть справочная страница с одним и тем же именем. Чтобы вызвать страницу из нужного раздела, добавьте его номер перед именем команды. Например, команда `man 1 ls` выведет тот же результат, который вы только что видели.

В большинстве операционных систем семейства Unix есть такие разделы справочных страниц:

1. Команды общего назначения, то есть команды, которые обычно запускаются из командной строки.
2. Системные вызовы.

3. Библиотечные функции, которые охватывают стандартную библиотеку C.
4. Специальные файлы (как правило, это устройства, которые находятся в `/dev`) и драйверы.
5. Форматы файлов и соглашения, которые с ними связаны. Сюда входят конфигурационные файлы.
6. Игры и экранные заставки.
7. Прочее.
8. Команды и демоны для системного администрирования.

Таким образом, если вы захотите глубже познакомиться с той или иной темой из этой книги, то вам, скорее всего, понадобится начать с разделов 1, 5 и 8 справочных страниц.

Если вы не знаете точно, как называется нужная вам справочная страница, ее можно найти с помощью команды `apropos` *`ключевое_слово`* или `man -k` *`ключевое_слово`*[1]. Эти команды выводят список всех справочных страниц, имя или описание которых содержит указанное слово.

Автозавершение команд

Если вы работаете в интерактивном сеансе командной оболочки (то есть не запускаете готовый сценарий или не создаете Docker-файл), то можете использовать **автозавершение команд** (оно же **автодополнение** или **автозавершение по Tab**), которое позволяет составлять команды так, чтобы нажимать меньше клавиш и допускать меньше опечаток.

Чтобы воспользоваться автозавершением, начните набирать имя файла или каталога и нажмите Tab. Оболочка будет постепенно сужать ваш выбор, отображая возможные соответствия под строкой, в которую вы вводите команду. Если для вашего ввода остался всего один подходящий вариант, то оболочка автоматически подставит соответствующую команду или аргумент, так что вам останется только нажать Enter. Давайте рассмотрим пример.

Если вы находитесь в своем домашнем каталоге в настольной системе Linux, его содержимое может выглядеть так:

```
→ ~ pwd
/home/dave
```

[1] *Apropos* — «*относящийся к*». Флаг `-k` — первая буква слова слова *keyword* (*ключевое слово*). Короткому флагу `-k` соответствует длинная форма `--apropos`. — *Примеч. пер.*

```
→ ~ ls
Desktop
Documents
Downloads
Library
Movies
Music
Pictures
Public
code
go
```

Если вы хотите переместиться в каталог `Documents`, это можно сделать командой `cd`:

```
→ ~ cd Documents
```

Посмотрим, как ввести эту команду с помощью автозавершения. Сначала наберите `cd D` и нажмите Tab:

```
→ ~ cd D
Desktop/    Documents/    Downloads/
```

Как видите, оболочка сузила выбор с десяти до трех возможных вариантов. Наберите еще одну букву и снова нажмите Tab. Вы увидите, что теперь подходят только два варианта:

```
→ ~ cd Do
Documents/    Downloads/
```

Если набрать еще одну букву, а именно `c`, то выбор сузится до одного варианта, и очередное нажатие Tab приведет к тому, что подходящее имя каталога автоматически подставится в командную строку:

```
→ ~ cd Documents/
```

После того как автозавершение подставит имя каталога, полученную команду можно запустить как обычно, нажав Enter, а можно дальше дополнять путь с помощью автозавершения — теперь уже внутри этого каталога. Например, если вы снова нажмете Tab, то автозавершение продолжится для имен файлов и каталогов, которые вложены в каталог `Documents`: часть аргумента `Documents/` останется без изменений, а подходящие имена будут подставляться после слеша. Текущий каталог не изменится, пока вы не составите допустимый путь и не нажмете Enter.

Этот простой прием сэкономит вам *огромный* объем работы по набору на клавиатуре. Чем раньше вы начнете им пользоваться, тем лучше!

Итоги

В этой главе вы изучили основы, которые нужны для эффективной работы с командной строкой. Вы увидели практические примеры кода и получили представление о том, как большинство команд принимает аргументы.

Мы также ввели понятие командной оболочки и разобрались, как она ищет исполняемые файлы после того, как вы набрали команду и нажали клавишу Enter. Как ни удивительно, многие продвинутые пользователи не до конца понимают эти две темы, и это мешает им быстро и эффективно работать с интерфейсом командной строки.

Наконец, вы освоили самые важные из простейших команд, которые позволяют взаимодействовать с системой из командной строки. Эти команды вам предстоит использовать почти каждый раз, когда вы будете работать на компьютере под управлением Linux; это необходимый минимум знаний, которыми важно овладеть каждому, кто собирается освоить Linux. Вы узнали и первый прием, который позволяет сэкономить время, — автозавершение команд.

Если вы практиковались по ходу этой главы и запускали все изученные команды в реальной оболочке Linux (а вам *не мешало бы* их запускать!), поупражняйтесь еще немного и закрепите полученные знания перед тем, как переходить к следующей главе. Мы будем опираться на этот материал на протяжении всей книги.

2

Как работать с процессами

Вы — разработчик, а это означает, что вы уже на интуитивном уровне знакомы с процессами. Они представляют собой не что иное, как плоды вашего труда: после того как вы написали код и отладили его, ваша программа в конце концов запускается, превращаясь в великолепный процесс под управлением операционной системы!

Процессом в Linux может быть и «долгоиграющее» приложение, и разовая команда оболочки (например, `ls`), и любой другой объект, который порождается ядром, чтобы выполнить какую-нибудь работу в системе. Если в Linux что-то происходит, значит, за это отвечают один или несколько процессов. Браузер, текстовый редактор, утилита поиска уязвимостей в системе и даже операции чтения файлов или команд, которые вы изучили к этому моменту, — все эти компоненты были запущены как процессы.

Процесс Linux — это важнейшая абстракция, на которую опираются все команды и инструменты, с которыми вам предстоит работать, поэтому важно понимать, как устроена модель процессов в Linux. Мы отвлечемся от конструкций, с которыми вы привыкли иметь дело как разработчики, таких как переменные, функции или потоки, и будем считать, что все они инкапсулированы в нечто, что называется «процессом». Теперь в вашем распоряжении будет новый, внешний набор элементов управления и индикаторов: идентификатор процесса, его состояние, потребление ресурсов и остальные атрибуты процесса, о которых мы поговорим в этой главе.

Прежде всего мы подробнее рассмотрим абстракцию процесса как таковую, а затем перейдем к практическим приемам работы с процессами в Linux. Хотя мы будем ориентироваться в первую очередь на практику, нам предстоит также небольшой теоретический обзор факторов (таких, как права доступа), из-за которых часто возникают проблемы, а кроме того, вы узнаете несколько эвристических правил, которые помогают устранять неполадки.

Эта глава охватывает такие темы:

- Что такое процесс Linux и как узнать, какие процессы выполняются в системе прямо сейчас.
- Какие атрибуты бывают у процесса и какие сведения о процессах можно собрать, чтобы устранять неполадки.
- Популярные команды, с помощью которых можно просматривать и искать процессы.
- Более продвинутые темы; они пригодятся, если вы будете писать программы, которые должны запускаться как процессы Linux: что такое сигналы и межпроцессная коммуникация, как устроена виртуальная файловая система `/proc`, как просматривать дескрипторы открытых файлов с помощью команды `lsof` и как создаются процессы в Linux.

Вас также ждет практическая работа, в ходе которой вы будете устранять неполадки, применяя теоретические знания и команды, изученные в этой главе. Так что давайте для начала разберемся, что же на самом деле представляет собой процесс Linux.

Введение в процессы Linux

Когда говорят о «процессе» в Linux, то имеют в виду внутреннюю модель операционной системы, которая отражает, *что конкретно* представляет собой выполняющаяся программа. Системе Linux необходима универсальная абстракция, которая распространялась бы на *любые* программы и инкапсулировала бы факторы, которыми управляет ОС. Процесс — это именно такая абстракция, и она позволяет операционной системе отслеживать важные элементы контекста, который окружает запущенные программы, а именно:

- потребление памяти;
- процессорное время;
- потребление других ресурсов (доступ к диску, сетевой трафик);
- коммуникация между процессами;
- сопутствующие процессы, которые инициирует программа (например, если она запускает команду оболочки).

Чтобы просмотреть список всех системных процессов (точнее, тех, для просмотра которых у вас есть права), можно запустить программу `ps` с флагами `aux`[1]:

[1] Сокр. *process status* (*состояние процессов*). Здесь `aux` — не один параметр, а три разных флага, которые в совокупности означают «отображать наиболее полный список про-

```
root@localhost  :~# ps aux
USER  PID  %CPU  %MEM     VSZ    RSS  TTY  STAT  START  TIME  COMMAND
root    1   0.2   0.3  167308  12816   ?   Ss    18:55  0:01  /sbin/init
root    2   0.0   0.0       0      0   ?   S     18:55  0:00  [kthreadd]
root    3   0.0   0.0       0      0   ?   I<    18:55  0:00  [rcu_gp]
root    4   0.0   0.0       0      0   ?   I<    18:55  0:00  [rcu_par_gp]
root    5   0.0   0.0       0      0   ?   I<    18:55  0:00  [slub_flushwq]
root    6   0.0   0.0       0      0   ?   I<    18:55  0:00  [netns]
root    8   0.0   0.0       0      0   ?   I<    18:55  0:00  [kworker/0:0H
                                                               -events_highpri]
root   10   0.0   0.0       0      0   ?   I<    18:55  0:00  [mm_percpu_wq]
root   11   0.0   0.0       0      0   ?   S     18:55  0:00  [rcu_tasks_rude_]
root   12   0.0   0.0       0      0   ?   S     18:55  0:00  [rcu_tasks_trace]
root   13   0.0   0.0       0      0   ?   S     18:55  0:00  [ksoftirqd/0]
root   14   0.0   0.0       0      0   ?   I     18:55  0:00  [rcu_sched]
root   15   0.0   0.0       0      0   ?   S     18:55  0:00  [migration/0]
root   16   0.0   0.0       0      0   ?   S     18:55  0:00  [idle_inject/0]
root   17   0.0   0.0       0      0   ?   I     18:55  0:00  [kworker/0:1
                                                               -cgroup_destroy]
root   18   0.0   0.0       0      0   ?   S     18:55  0:00  [cpuhp/0]
root   19   0.0   0.0       0      0   ?   S     18:55  0:00  [cpuhp/1]
root   20   0.0   0.0       0      0   ?   S     18:55  0:00  [idle_inject/1]
root   21   0.0   0.0       0      0   ?   S     18:55  0:00  [migration/1]
root   22   0.0   0.0       0      0   ?   S     18:55  0:00  [ksoftirqd/1]
root   23   0.0   0.0       0      0   ?   I     18:55  0:00  [kworker/1:0
                                                               -events_unbound]
```

Далее в этой главе мы рассмотрим атрибуты, с которыми вам придется чаще всего иметь дело при разработке ПО.

Из чего состоит процесс Linux

С точки зрения операционной системы процесс — это просто структура данных, которая обеспечивает удобный доступ к сведениям такого рода:

- **Идентификатор процесса (Process ID**, или `PID` в предыдущем листинге**)**. Процесс с идентфикатором 1 порождает все остальные процессы и соответствует подсистеме инициализации, которая запускает всю систему. Когда ядро ОС начинает работу, оно создает этот процесс одним из первых. Когда порождается новый процесс, ему присваивается следующий доступный идентификатор в порядке возрастания. Поскольку без процесса инициализации система не может нормально функционировать, его нельзя принудительно завершить, даже с правами пользователя `root`. В разных ОС семейства Unix бывают разные

цессов в расширенном формате» (подробнее см. `man ps`). В отличие от большинства флагов, эти флаги употребляются без начального дефиса. Тот же результат можно получить, если указать флаги в любом порядке и/или через пробел, например `ps uxa` или `ps x a u`. — *Примеч. науч. ред.*

подсистемы инициализации: например, большинство дистрибутивов Linux используют `systemd`, в macOS действует `launchd`, а во многих других Unix'ах — SysV[1].

ПРИМЕЧАНИЕ

При контейнерной виртуализации процессы объединяются в пространства имен: например, в среде эксплуатации у всех процессов одного и того же контейнера может быть PID 3210, который отображается на множество процессов с идентификаторами от 1 до n внутри контейнера (n — общее количество процессов, которые выполняются в контейнере). Эту схему можно проследить извне контейнера, но не изнутри.

- **Идентификатор родительского процесса** (**Parent Process ID**, или `PPID`). Каждый процесс порождается другим процессом, который называется родительским по отношению к порожденному. Если родительский процесс завершается, когда дочерний еще выполняется, то дочерний процесс становится «сиротой» и переподчиняется подсистеме инициализации (PID 1)[2].
- **Состояние процесса** (`STAT`). Команда `man ps` покажет вам обзор возможных состояний:

`D`	Ожидание без возможности прерывания (обычно процесс ввода-вывода)
`I`	Простаивающий поток ядра
`R`	Процесс выполняется или готов к выполнению (в очереди выполнения)
`S`	Ожидание с возможностью прерывания (процесс ожидает, пока наступит то или иное событие)
`T`	Процесс приостановлен сигналом диспетчера задач
`t`	Процесс приостановлен отладчиком в ходе трассировки
`X`	Мертвый процесс (не должен наблюдаться в исправной системе)
`Z`	Процесс-«зомби», который выведен из эксплуатации, но не ликвидирован родительским процессом

[1] В английском издании этой книги, как и в большинстве англоязычных источников, подсистема инициализации называется *init* независимо от того, какой конкретно программой она реализуется — `init`, `systemd` или какой-то еще. — *Примеч. пер.*

[2] Для самóй подсистемы инициализации родительским процессом обычно считается процесс с фиктивным PID 0, который не является настоящим процессом Linux, а имитируется диспетчером подкачки на уровне ядра. — *Примеч. науч. ред.*

- **Приоритетность (Priority)** — мера «уступчивости», которая показывает, насколько процесс готов уступать ресурсы другим процессам.
- **Владелец** процесса (`USER`) — идентификатор пользователя, которому принадлежит процесс.
- **Идентификатор группы** (`EGID`), которой принадлежит процесс.
- Распределение **адресного пространства** памяти процесса.
- Сведения о том, какие **ресурсы** потребляет процесс: открытые файлы, сетевые порты и т. д. (Например, в предыдущем листинге столбцы `VSZ` и `RSS` показывают потребление памяти.)

Давайте ближе познакомимся с отдельными атрибутами процессов, которые особенно важны для того, чтобы разрабатывать программное обеспечение и устранять неполадки.

Идентификатор процесса (Process ID, PID)

У каждого процесса есть идентификатор (PID) — уникальное целое положительное число, которое присваивается процессу, когда он порождается. Операционная система Linux отслеживает состояние каждого процесса по его PID; это во многом похоже на то, как в реляционных базах данных каждая строка идентифицируется уникальным ключом.

PID — это самый важный атрибут из тех, которые вам пригодятся, чтобы взаимодействовать с процессами.

Эффективные идентификаторы пользователя (EUID) и группы (EGID)

Эти атрибуты указывают, от имени какого пользователя и какой группы процесс запущен в системе. Права доступа пользователя и группы в совокупности определяют, какие операции процессу разрешено выполнять.

Как вы увидите в главе 5 «Файлы в Linux», для каждого файла задано, какому пользователю и какой группе он принадлежит, а от этого зависит, на кого распространяются права доступа к нему. Если представить себе, что владение файлом и права доступа к нему — это замóк, то процесс с подходящими правами пользователя и группы — это ключ, который отпирает замок и открывает доступ к файлу. Мы подробнее поговорим об этом позже, когда займемся правами доступа.

Переменные окружения

Скорее всего, вы уже использовали переменные окружения в своих приложениях: они позволяют операционной среде, которая запустила процесс, передавать ему

нужные данные. К ним относятся, в частности, директивы конфигурации (например, `LOG_DEBUG=1`) и конфиденциальные ключи (например, `AWS_SECRET_KEY`), и в каждом языке программирования есть тот или иной способ извлечь эти данные из контекста программы.

Например, этот сценарий на Python извлекает путь к домашнему каталогу пользователя из переменной окружения `HOME`, а затем выводит этот путь на экран:

```
import os
home_dir = os.environ['HOME']
print("Домашний каталог текущего пользователя:", home_dir)
```

Когда мы запускаем эту программу в сеансе REPL интерпретатора `python3` на компьютере под управлением Linux, она выводит такой результат:

```
Домашний каталог текущего пользователя: /home/dcohen
```

Рабочий каталог

С любым процессом, так же как и с командной оболочкой (которая, естественно, тоже является процессом), связан текущий рабочий каталог. Если запустить в оболочке команду `pwd`, вы увидите текущий рабочий каталог оболочки; аналогичный каталог есть у каждого процесса. Впрочем, пока процесс существует, его текущий рабочий каталог может меняться, так что на него не стоит чересчур полагаться.

На этом мы заканчиваем обзор основных атрибутов процессов. В следующем разделе мы перейдем от теории к практике и рассмотрим несколько команд, с помощью которых можно начать работать с процессами прямо сейчас.

Команды для работы с процессами Linux

Перечислим команды, которые пригодятся вам в первую очередь.

ps — сведения о процессах

Команда `ps`, которая уже встречалась ранее в этой главе, выводит сведения о процессах, которые запущены в системе. От флагов зависит, какие атрибуты процессов будут показаны в качестве столбцов. Эта команда обычно используется в совокупности с фильтрами, которые ограничивают вывод: например, составная команда `ps aux | head -n 10` сокращает вывод до первых 10 строк. Вот еще несколько полезных примеров:

- `ps -eLf` выводит информацию о потоках, которые связаны с процессом:

```
root@40086047ef36:/# ps -eLf
UID  PID  PPID  LWP  C  NLWP  STIME  TTY    TIME     CMD
```

```
root   1     0    1  0     1  22:30  pts/0  00:00:00  /bin/bash
root   9     1    9  0     1  22:30  pts/0  00:00:00  ps -eLf
```

- `ps -ejH` наглядно отображает иерархические отношения между процессами (дочерние процессы показаны с отступами относительно родительских):

```
root@40086047ef36:/# ps -ejH
PID    PGID    SID    TTY        TIME    CMD
  1       0      0      ?    00:00:00    init
  7       7      7      ?    00:00:00      init
  8       7      7      ?    00:00:00        init
  9       9      9  pts/0    00:00:00          bash
247     247      9  pts/0    00:00:00            nano
292     292      9  pts/0    00:00:00            ps
```

pgrep — найти процесс по имени

Команда `pgrep`[1] ищет идентификаторы процессов по их имени и может принимать в качестве аргументов регулярные выражения, например:

Процессы, в имени которых есть строка `nginx`

```
root@localhost:~# pgrep nginx
1589
1592
1593
```

Процессы, имена которых начинаются с `n`

```
root@localhost:~# pgrep ^n
6
584
1589
1592
1593
```

top — системный монитор

Команда `top`[2] — это интерактивная программа, которая опрашивает все процессы (по умолчанию ежесекундно) и выводит их список, отсортированный по потреблению тех или иных ресурсов (их набор можно настроить). Она также отображает количество ресурсов, потребляемых системой в целом. Чтобы выйти из режима просмотра, нажмите Q или Ctrl+C. Пример использования этой команды встретится вам далее в этой главе.

---

[1] В названии команды `pgrep` буква `p` — от слова *process* (*процесс*), а `grep` — название другой классической команды Unix, которая ищет строки в текстовых данных и подробно рассматривается в главе 5. — *Примеч. науч. ред.*

[2] «*Верхний*», «*находящийся на первых строчках рейтинга*». Команда называется так потому, что по умолчанию выводит только самые ресурсоемкие процессы. — *Примеч. науч. ред.*

iotop — монитор ввода-вывода

Команда `iotop`[1] подобна `top`, но отслеживает дисковый ввод-вывод и отлично помогает находить процессы с самым интенсивным вводом-выводом. Далеко не во всех системах она установлена по умолчанию, но ее можно установить с помощью большинства диспетчеров пакетов.

Вот пример того, что выводит `iotop`[2]:

```
Total DISK READ:       0.00 B/s | Total DISK WRITE:      0.00 B/s
Current DISK READ:     0.00 B/s | Current DISK WRITE:    0.00 B/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
    1  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  ini.t
    2  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [kthreadd]
    3  be/0  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [rcu_gp]
    4  be/0  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [rcu_par_gp]
    5  be/0  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [slub_flushwq]
    6  be/0  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [netns]
    8  be/0  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [kworker~_highpri]
   10  be/0  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [mm_percpu_wq]
   11  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [rcu_tasks_rude_]
   12  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [rcu_tasks_trace]
   13  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [ksoftirqd/0]
   14  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [rcu_sched]
   15  rt/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [migration/ 0]
   16  rt/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [idle_inject/0]
   18  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [cpuhp/0]
   19  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [cpuhp/1]
   20  rt/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [idle_inject/1]
   21  rt/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [migration/1]
   22  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [ksoftirqd/1]
   23  be/4  root        0.00 B/s    0.00 B/s  0.00 %  0.00 %  [kworker~ercpu_wq]
 keys:  any: refresh  q: quit  i: ionice  o: active  p: procs  a: accum
 sort:  r: asc  left: SWAPIN  right: COMMAND  home: TID  end: COMMAND
```

nethogs — монитор сетевого трафика

Команда `nethogs`[3] — еще один аналог `top`, однако она отслеживает сетевой трафик. Эта команда особенно полезна тем, что группирует сетевую активность по процессам. Она доступна в большинстве диспетчеров пакетов.

---

[1] Буквы `io` в названии команды — от *input/output* (*ввод-вывод*). — *Примеч. пер.*

[2] Возможно, самые востребованные столбцы в этом выводе — TID (Thread ID, то есть *идентификатор потока*), disk read (*скорость чтения с диска*) и disk write (*скорость записи на диск*). — *Примеч. науч. ред.*

[3] В английском языке основное значение слова *hog* — свинья, боров, однако на профессиональном сленге оно также означает программу (процесс, поток, устройство и т. д.), которая слишком интенсивно потребляет тот или иной ресурс. Название программы `nethogs` — сокращение от *network hogs* («*пожиратели трафика*»). — *Примеч. пер.*

Вот пример того, что выводит `nethogs`:

```
NetHogs version 0.8.6-3
  PID  USER  PROGRAM                    DEV      SENT    RECEIVED
 2064  root  curl                       eth0     0.580   8.730 KB/sec
  719  root  sshd: root@pts/0           eth0    11.067   0.412 KB/sec
    ?  root  45.33.83.163:445-201.91..           0.021   0.024 KB/sec
    ?  root  45.33.83.163:6379-47.57..           0.011   0.014 KB/sec
    ?  root  45.33.83.163:7777-139.1..           0.000   0.000 KB/sec
 2062  root  curl                       eth0     0.000   0.000 KB/sec
    ?  root  45.33.83.163:9010-79.12..           0.000   0.000 KB/sec
 2060  root  curl                       eth0     0.000   0.000 KB/sec
    ?  root  45.33.83.163:50153-66.7..           0.000   0.000 KB/sec
    ?  root  45.33.83.163:36198-35.2..           0.000   0.000 KB/sec
    ?  root  45.33.83.163:37464-35.2..           0.000   0.000 KB/sec
    ?  root  45.33.83.163:2443-66.29..           0.000   0.000 KB/sec
    ?  root  45.33.83.163:57240-23.2..           0.000   0.000 KB/sec
    ?  root  unknown TCP                         0.000   0.000 KB/sec
  TOTAL                                         11.679   9.181 KB/sec
```

Продвинутые понятия и инструменты для работы с процессами

Здесь начинается углубленный раздел этой главы. Чтобы эффективно работать с процессами в Linux, вам не обязательно в совершенстве овладевать всем материалом из данного раздела, но эти знания могут принести огромную пользу. Если у вас найдется несколько минут времени, мы советуем потратить их на то, чтобы освоить приведенный ниже материал.

Сигналы

Каким образом `systemctl` заставляет веб-сервер заново подгрузить свои конфигурационные файлы? Как вежливо попросить процесс завершиться и убрать за собой? А как экстренно прервать неисправный процесс, который мешает работать вашему приложению в подакшен-среде?

В Unix и Linux все это делается с помощью сигналов. **Сигналы** — это числовые сообщения, которые можно передавать от одних программ к другим. Сигналы позволяют процессам отправлять и получать определенные сообщения и таким образом коммуницировать друг с другом и с операционной системой.

С помощью сообщений процессу можно передать информацию самого разного рода, например оповестить о том, что произошло определенное событие или что требуется то или иное действие.

Как сигналы используются на практике

Давайте рассмотрим на примерах, какую практическую пользу приносит механизм сигналов. С их помощью можно реализовать межпроцессную коммуникацию: например, один процесс может направить сигнал другому, чтобы оповестить о том, что процесс-отправитель закончил ту или иную задачу и процесс-получатель может начать свою работу. Это позволяет процессам координировать действия друг с другом и работать согласованно и эффективно подобно потокам выполнения в языках программирования (хотя и без совместного использования памяти).

Сигналы также нередко помогают обрабатывать программные ошибки. Например, представим себе процесс, который разработан специально для того, чтобы перехватывать сигнал `SIGSEGV`, обозначающий ошибку сегментации. Когда процесс получает этот сигнал, он может перехватить его и принять меры, чтобы занести ошибку в журнал, сформировать дамп ядра для отладки или корректно освободить все используемые ресурсы перед тем, как безопасно завершить работу.

Сигналы могут пригодиться и для того, чтобы реализовать само безопасное завершение. Например, когда система завершает работу, она может направить всем процессам сигнал, чтобы дать им возможность сохранить свое состояние и освободить ресурсы.

Перехват сигналов

Во многих ситуациях процессы, которым адресованы сигналы, могут их перехватывать. Это делается по тому же принципу, по какому объекты в языках программирования перехватывают и обрабатывают исключения.

Если у процесса-получателя есть функция-обработчик полученного сигнала, то она запускается. Благодаря этому программы могут заново читать свои конфигурационные файлы без перезапуска, а также завершать запись в базу данных и закрывать свои файловые дескрипторы, когда получают сигнал завершения.

Команда kill

Однако сигналы можно передавать не только от одних процессов другим. С помощью программы с пугающим (и, формально говоря, некорректным) названием `kill`[1] пользователи тоже могут передавать сигналы процессам.

Один из самых типичных вариантов использования команды `kill` — прервать процесс, который завис и не отвечает. Например, если процесс застрял в беско-

---

[1] «*Убить*». Ранние версии команды `kill` действительно позволяли только уничтожить процесс, но впоследствии ее функциональность расширилась: теперь с помощью `kill` можно передать процессу любой сигнал, хотя название команды осталось прежним. — *Примеч. науч. ред.*

нечном цикле, то с помощью `kill` ему можно послать сигнал, который заставит процесс прекратиться.

Эта команда принимает в качестве аргумента PID процесса, которому передается сигнал. Например, если вы хотите завершить процесс с PID 2600, введите такую команду:

```
kill 2600
```

Команда передаст процессу сигнал 15 (`SIGTERM`, или «завершить работу»), после чего у процесса будет возможность перехватить его и корректно завершиться.

ПРИМЕЧАНИЕ

Как видно из последующей таблицы, по умолчанию `kill` передает сигнал типа «Завершить работу» (с кодом 15), а не «Уничтожить процесс» (сигналу `SIGKILL` соответствует код 9). Программа `kill` позволяет не только уничтожать процессы, но и передавать им любые другие сигналы. Ее название действительно сбивает с толку; к сожалению, это одна из причуд Unix и Linux, к которой придется привыкнуть.

Если вы хотите передать сигнал не с кодом 15 по умолчанию, а с другим кодом, его можно указать после дефиса. Например, чтобы передать тому же процессу сигнал `SIGHUP`, запустите такую команду:

```
-kill -1 2600
```

Вот перечень распространенных сигналов в Linux, сгруппированный по тому, как процесс по умолчанию должен реагировать на сигнал[1].

Сигнал	Код	Описание
Уничтожить процесс		
`SIGHUP`	1	Контролирующий терминал завершил соединение, или контролирующий процесс уничтожен
`SIGINT`	2	Прерывание от клавиатуры
`SIGKILL`	9	Принудительное уничтожение процесса
`SIGPIPE`	13	Неисправный конвейер
`SIGALRM`	14	Сигнал таймера от `alarm`

[1] Во всех современных системах семейства Unix основные сигналы работают одинаково, однако второстепенные особенности поведения команды `kill` могут различаться в зависимости от дистрибутива. Подробную справку о сигналах в вашей системе можно вызвать командой `man signal`. — *Примеч. науч. ред.*

Сигнал	Код	Описание
SIGTERM	15	Сигнал на завершение работы
SIGSTKFLT	16	Ошибка стека сопроцессора
SIGVTALRM	26	Исчерпано процессорное время
SIGPROF	27	Сигнал от профилирующего таймера
SIGIO SIGPOLL	29	Возможен ввод-вывод
SIGPWR SIGINFO	30	Ошибка энергоснабжения
SIGEMT	–	Перехват эмулятора
Уничтожить процесс и сформировать дамп ядра		
SIGQUIT	3	Запрос на выход от клавиатуры
SIGILL	4	Недопустимая инструкция
SIGTRAP	5	Перехват трассировки или точки останова
SIGABRT SIGIOT	6	Сигнал на прерывание процесса от `abort`
SIGBUS	7	Ошибка шины
SIGFPE	8	Исключение вычислений с плавающей точкой
SIGSEGV	11	Недопустимая ссылка на память
SIGXCPU	24	Исчерпано предельное процессорное время
SIGXFSZ	25	Исчерпан предельный размер файла
SIGSYS SIGUNUSED	31	Недопустимый системный вызов
Приостановить процесс		
SIGSTOP	19	Безусловная приостановка процесса
SIGTSTP	20	Запрос на приостановку процесса
SIGTTIN	21	Ввод в терминале для фонового процесса
SIGTTOU	22	Вывод в терминале для фонового процесса
Игнорировать сигнал		
SIGCHLD SIGCLD	17	Дочерний процесс приостановлен или завершен
SIGURG	23	Экстренная ситуация сокета
SIGWINCH	28	Изменение размера окна

Сигнал	Код	Описание
Прочее		
SIGCONT	18	Возобновление приостановленного процесса
SIGUSR1	10	Сигналы, определяемые пользователем; по умолчанию процесс уничтожается
SIGUSR2	12	

Вам будет особенно полезно (в том числе для интервью с работодателями) ориентироваться в самых важных сигналах:

Сигнал	Код	Описание
SIGHUP	1	«Положить трубку» (*hangup*). Многие приложения (например, nginx) интерпретируют этот сигнал как «Обновить свою конфигурацию, потому что в нее внесены изменения»
SIGINT	2	«Прерывание» (*interrupt*); этот сигнал вежливо просит, чтобы процесс корректно завершился
SIGTERM	15	«Завершение» (*terminate*); обычно интерпретируется так же, как SIGINT[1]
SIGUSR1 SIGUSR2	10 12	Обычно используются, чтобы передавать пользовательские сообщения между приложениями. Например, SIGUSR1 заставляет nginx заново открыть файлы журналов, в которые он вносит записи: это полезно, если вы только что провели ротацию журналов
SIGKILL	9	Процесс не может перехватить и обработать этот сигнал. Если он передан программе, то операционная система немедленно ее завершит. При этом не выполняется никакая очистка — в частности, не освобождаются файлы, которые были открыты для записи, и не запускается безопасное завершение работы. Таким образом, сигнал SIGKILL может повредить данные и поэтому считается крайней мерой

Если вам интересно глубже изучить процессы в Linux, обратите внимание на каталог /proc[2]. Эта тема точно не относится к элементарным понятиям, однако в этом каталоге содержится поддерево файловой системы, где каждому процессу соот-

[1] Как правило, SIGINT поступает от пользователя (с помощью клавиатуры или другого устройства ввода), а SIGTERM — от других процессов или от операционной системы. — *Примеч. науч. ред.*

[2] Сокр. *process* (*процесс*). — *Примеч. пер.*

ветствует свой каталог и файлы в нем отражают актуальную информацию о процессе.

На практике каталог `/proc` может пригодиться, чтобы устранять неполадки, когда вы обнаружили неисправный (или непонятный) процесс и хотите в режиме реального времени проследить, что он делает.

Можно многое узнать о процессе, если просто покопаться в его подкаталоге внутри каталога `/proc` и навести справки в поисковой системе.

Многие инструменты, которые вы видели в этой главе, на самом деле обращаются за сведениями о процессе к каталогу `/proc` и показывают вам только небольшую часть того, что в нем находится. Если вы действительно хотите получить сразу всю информацию и самостоятельно отфильтровать нужные данные, то `/proc` всегда к вашим услугам.

lsof — дескрипторы файлов, которые открыл процесс

Команда `lsof`[1] выводит перечень всех файлов, которые процесс открыл для чтения и/или записи. Эти сведения иногда бывают ценными, потому что всего один мелкий дефект в программе может привести к утечке файловых дескрипторов (ссылок на файлы, к которым программа запросила доступ). А это может вызвать перерасход ресурсов, повреждение файлов и много других неприятностей.

К счастью, легко узнать, какие файлы открыл тот или иной процесс. Просто запустите команду `lsof` и передайте ей флаг `-p` с идентификатором процесса (скорее всего, эту команду понадобится запускать от имени `root`[2]). Вы увидите список файлов, которые открыл процесс (в данном случае — процесс с PID 1589).

```
→ ~ lsof -p 1589
COMMAND PID USER  FD TYPE DEVICE SIZE/OFF   NODE  NAME
nginx  1589 root cwd  DIR    8,0     4096      2  /
nginx  1589 root rtd  DIR    8,0     4096      2  /
nginx  1589 root txt  REG    8,0  1240136  78096  /usr/sbin/nginx
nginx  1589 root mem  REG    8,0   184904 262503  /usr/1ib/nginx/modules/
                                                   ngx_stream_module.so
nginx  1589 root mem  REG    8,0   112264 262494  /usr/1ib/nginx/modules/
                                                   ngx_mail_module.so
nginx  1589 root mem  REG    8,0   149760    867  /usr/1ib/x86_64-linux-
                                                   gnu/libgpg-error.so.0.32.1
nginx  1589 root mem  REG    8,0   125488   1035 /usr/1ib/x86_64-linux-
                                                  gnu/libgcc_s.so.1
```

---

[1] Сокр. *list open files* (*перечислить открытые файлы*). — *Примеч. пер.*

[2] Чтобы запустить команду от имени `root`, укажите перед ней `sudo`, например: `sudo lsof -p 1589`. Возможно, после этого вам придется ввести пароль вашего пользователя, чтобы команда сработала. Подробнее о `sudo` пойдет речь далее в главе 4. — *Примеч. науч. ред.*

```
nginx  1589 root mem  REG    8,0  2252096    837 /usr/1ib/x86_64-linux-
                                                  gnu/libstdc++.so.6.0.30
nginx  1589 root mem  REG    8,0 29476472   8427 /usr/1ib/x86_64-linux-
                                                  gnu/libicudata.so.70.1
nginx  1589 root mem  REG    8,0  1296312    840 /usr/1ib/x86_64-linux-
                                                  gnu/libgcrypt.so.20.3.4
nginx  1589 root mem  REG    8,0  2062664   8432 /usr/1ib/x86_64-linux-
                                                  gnu/libicuuc.so.70.1
nginx  1589 root mem  REG    8,0    96416  17716 /usr/1ib/x86_64-linux-
                                                  gnu/libexslt.so.0.8.20
nginx  1589 root mem  REG    8,0   264632  17717 /usr/1ib/x86_64-linux-
                                                  gnu/libxslt.so.1.1.34
nginx  1589 root mem  REG    8,0  1967384   5818 /usr/1ib/x86_64-linux-
                                                  gnu/libxml2.so.2.9.13
nginx  1589 root mem  REG    8,0    27672 262485 /usr/1ib/nginx/modules/ngx_
                                                  http_xslt_filter_module.so
```

В предыдущем листинге показано, что выводит `lsof` для процесса веб-сервера nginx. В первой строке фигурирует текущий рабочий каталог процесса — в данном случае корневой каталог (`/`). Как видите, к процессу также относится дескриптор исполняемого файла самого́ nginx (`/usr/sbin/nginx`) и дескрипторы различных библиотек в каталоге `/usr/lib`.

Если просматривать вывод `lsof` дальше, то можно заметить более интересные пути к файлам:

```
COMMAND PID USER   FD TYPE             DEVICE  SIZE  NODE  NAME
                                                /OFF
nginx  1589 root DEL   REG                0,1          11  /dev/zero
nginx  1589 root   0u  CHR                1,3  0t0      5  /dev/null
nginx  1589 root   1u  CHR                1,3  0t0      5  /dev/null
nginx  1589 root   2w  REG                8,0   76 262197  /var/log/nginx/
                                                  error.log
nginx  1589 root   3u unix 0xffff8da007507300  0t0  23207  type=STREAM
nginx  1589 root   4w  REG                8,0 2910 262196  /var/log/nginx/
                                                  access.log
nginx  1589 root   5w  REG                8,0   76 262197  /var/log/nginx/
                                                  error.log
nginx  1589 root   6u IPv4              23965  0t0    TCP  *:http (LISTEN)
nginx  1589 root   7u IPv6              23966  0t0    TCP  *:http (LISTEN)
nginx  1589 root   8u unix 0xffff8da007505540  0t0  23208  type=STREAM
nginx  1589 root   9u unix 0xffff8da007504000  0t0  29795  type=STREAM
nginx  1589 root  10u unix 0xffff8da007505dc0  0t0  29796  type=STREAM
```

В этом листинге упоминаются файлы журналов, в которые nginx ведет запись (их имена оканчиваются на `.log`), а также файлы сокетов (Unix, IPv4 и IPv6), которые процесс открыл для чтения и записи. В Unix и Linux сетевые сокеты — это просто особый тип файлов, поэтому один и тот же набор базовых инструментов можно использовать в самых разных ситуациях. Если практически все в программном

окружении представлено в виде файлов, то инструменты для работы с файлами оказываются невероятно мощными.

Наследование

За исключением самого первого процесса (в нашем примере это процесс `init` с PID 1), каждый процесс порождается родительским процессом, который фактически создает свою копию по принципу ветвления (с помощью системного вызова `fork()`). Новый процесс обычно наследует права доступа, переменные окружения и другие атрибуты своего родителя.

Хотя такое поведение по умолчанию можно изменить, это влечет риски для безопасности, потому что программное обеспечение, которое вы запускаете вручную, получает права доступа текущего пользователя (или даже права `root`, если вы применяете `sudo`). Эти права наследуют все дочерние процессы, которые порождает этот процесс (например, во время установки или компиляции программы).

Представьте себе процесс веб-сервера, который запущен с привилегиями `root` (чтобы присоединиться к сетевому порту), а в его переменных окружения хранятся ключи для аутентификации в облачном хранилище (чтобы извлекать данные из облака). Если этот основной процесс порождает дочерний процесс, которому не нужны ни права `root`, ни конфиденциальные переменные окружения, то вы создадите неоправданный риск для безопасности, если позволите передать эти атрибуты новому процессу. В результате обычной практикой стало «обнулять» права доступа и очищать переменные окружения, когда те или иные службы порождают дочерние процессы.

С точки зрения безопасности об этом важно помнить, чтобы предотвратить утечку таких чувствительных данных, как пароли или доступ к конфиденциальным файлам. Подробное обсуждение того, как обеспечивать безопасность данных, выходит за рамки этой книги, однако об этой проблеме не стоит забывать, если вы пишете программное обеспечение, которое будет запускаться на компьютерах под управлением Linux.

Практическая работа: сеанс устранения неполадок

Давайте рассмотрим пример, в котором нам предстоит устранять неполадки в системе. Пока мы не приступили к делу, нам известно лишь то, что некий сервер под управлением Linux работает невероятно медленно.

Для начала попробуем посмотреть, что происходит в системе. Совсем недавно вы узнали, что интерактивная команда `top` позволяет просматривать «живой» снимок текущих процессов. Давайте же ее запустим.

```
top - 19:06:27 up 10 min,  2 users, load average: 0.08, 0.03, 0.01
Tasks: 108 total,  3 running,  105 sleeping,  0 stopped,  0 zombie
%Cpu(s): 46.8 us,  2.0 sy,  0.0 ni,  48.8 id,  2.3 wa,  0.0 hi,  0.0 si, 0.0 st
MiB Mem:  3924.0 total,   3260.3 free,   145.5 used,    518.2 buff/cache
MiB Swap:  512.0 total,    512.0 free,     0.0 used,   3552.6 avail Mem
  PID  USER  PR  NI    VIRT   RES  SHR  S %CPU %MEM TIME+ COMMAND
 1763  root  20   0   10168  7820 1436  R 94.7 0.2 0:03.14 bzip2
 1761  root  20   0    7324  3100 2872  S  3.0 0.1 0:00.11 tar
    1  root  20   0  167308 12816 8300  S  0.0 0.3 0:01.82 systemd
    2  root  20   0       0     0    0  S  0.0 0.0 0:00.00 kthreadd
    3  root   0 -20       0     0    0  I  0.0 0.0 0:00.00 rcu_gp
    4  root   0 -20       0     0    0  I  0.0 0.0 0:00.00 rcu_par_gp
    5  root   0 -20       0     0    0  I  0.0 0.0 0:00.00 slub_flushwq
    6  root   0 -20       0     0    0  I  0.0 0.0 0:00.00 netns
    8  root   0 -20       0     0    0  I  0.0 0.0 0:00.00 kworker/
                                                          0:0H-events_highpri
   10  root   0 -20       0     0    0  I  0.0 0.0 0:00.00 mm_percpu_wq
   11  root  20   0       0     0    0  S  0.0 0.0 0:00.00 rcu_tasks_rude_
   12  root  20   0       0     0    0  S  0.0 0.0 0:00.00 rcu_tasks_trace
   13  root  20   0       0     0    0  S  0.0 0.0 0:00.02 ksoftirqd/0
   14  root  20   0       0     0    0  R  0.0 0.0 0:00.03 rcu_sched
   15  root  rt   0       0     0    0  S  0.0 0.0 0:00.00 migration/0
   16  root -51   0       0     0    0  S  0.0 0.0 0:00.00 idle_inject/0
   17  root  20   0       0     0    0  I  0.0 0.0 0:00.33 kworker/
                                                          0:1-mm_percpu_wq
   18  root  20   0       0     0    0  S  0.0 0.0 0:00.00 cpuhp/0
   19  root  20   0       0     0    0  S  0.0 0.0 0:00.00 cpuhp/1
```

По умолчанию `top` сортирует процессы по потреблению ресурсов центрального процессора (столбец `%CPU`), так что первый процесс, который мы видим в таблице, и будет проблемным. Оказывается, этот процесс потребляет 94 % доступного процессорного времени.

В результате запуска `top` мы узнали кое-что полезное:

- Проблема заключается именно в чрезмерной нагрузке на ЦП, а не в нехватке какого-то другого ресурса.
- Проблемный процесс имеет PID 1763, и соответствующая ему команда (согласно столбцу `COMMAND`) — это `bzip2`, которая предназначена для сжатия данных.

Мы делаем вывод, что активный процесс `bzip2` нам сейчас не нужен и его стоит остановить. С помощью команды `kill` мы просим процесс завершиться:

```
kill 1763
```

Подождав несколько секунд, мы проверяем, продолжает ли работать этот (или любой другой) процесс `bzip2`:

```
pgrep bzip2
```

К сожалению, мы убеждаемся, что процесс с тем же самым PID все еще выполняется. Пришло время принимать радикальные меры:

```
kill -9 1763
```

Эта команда поручает операционной системе уничтожить процесс, не позволяя ему перехватить сигнал (и, возможно, проигнорировать его). Сигнал `SIGKILL` (с кодом 9) просто убивает процесс на месте.

После того как вы уничтожили проблемный процесс, сервер работает как по маслу. Теперь можно разыскать того разработчика, который решил, что сжимать гигантские каталоги с исходным кодом — это хорошая идея.

В этом примере мы на практике рассмотрели наиболее распространенную схему устранения неполадок в реальной системе:

- Мы проанализировали потребление ресурсов. Здесь мы запустили для этого команду `top`, но могли бы использовать и любой другой изученный инструмент в зависимости от того, с какими системными ресурсами наблюдаются проблемы.
- Мы нашли PID процесса, который вызывает неполадки.
- Мы приняли меры. В нашем примере не потребовалось изучать ситуацию глубже, и мы передали процессу сигнал, чтобы его завершить, — сначала `SIGTERM` (с кодом 15), а затем `SIGKILL` (с кодом 9).

Итоги

В этой главе мы подробно изучили абстракцию процесса, которая в Linux представляет выполняющиеся программы. Вы узнали, какими общими компонентами обладают все процессы, и освоили основные команды для того, чтобы искать и исследовать запущенные процессы. Эти инструменты позволят вам не только узнавать о том, что какой-то процесс вызывает неполадки, но и выяснять, *какой именно* процесс.

3

Как управлять службами с помощью systemd

В предыдущей главе вы узнали, как устроены процессы в Linux. Теперь настало время разобраться, как эти процессы «упаковываются» в еще один уровень абстракции — службу `systemd`.

Команды, которые вы изучали до сих пор, — `ls`, `mv`, `rm`, `ps` и др. — выполняются на переднем плане и привязываются к сеансу командной оболочки. Вы запускаете программу, она делает свою работу и завершается. Однако так ведут себя не все программы.

Службы, которые также часто называются *демонами*[1], — это продолжительные процессы, которые выполняются в фоновом режиме. К ним относятся, например,

---

[1] Называть фоновые процессы словом *daemon* начали в 1960-х годах программисты, которые работали над проектом MAC в Массачусетском технологическом институте. Это название было вдохновлено «демонами» из научной терминологии, такими как демон Максвелла в термодинамике или демон Лапласа в классической механике. (В научных работах *демонами* иногда называются воображаемые сверхъестественные сущности, которые выполняют те или иные «волшебные» функции в мысленных экспериментах.) Англоязычные компьютерные специалисты используют термин *daemon* (или *dæmon*), чтобы подчеркнуть преемственность с *даймонами*, или *даймониями*, — духами-покровителями из древнегреческой мифологии. (При этом демон как инфернальное мифическое существо по-английски называется *demon*.) С другой стороны, иногда программы-«демоны» ассоциируются и с хрестоматийными исчадиями ада: например, талисман операционных систем семейства BSD — «мультяшное» изображение красного чертёнка в кроссовках. В русскоязычной сфере IT устоялось название *демон*, а не *даймон*. — *Примеч. науч. ред.*

системы управления базами данных и веб-серверы, а также обычные системные службы вроде диспетчера сетевых соединений, рабочего стола и т. д. Эти «долгоиграющие» фоновые службы обычно запускает и координирует подсистема инициализации, например `systemd`.

Подсистемой инициализации (init) называется первый процесс, который запускается ядром операционной системы и отвечает за то, чтобы запускать все остальные процессы.

Службами `systemd` можно управлять с помощью утилиты командной строки, которая называется `systemctl`. Она позволяет запускать и останавливать службы, например перезапустить службу, если та не отвечает или если нужно обновить ее конфигурацию.

Если вы изучаете книгу не по порядку и еще не прочли предыдущие главы, эта глава все равно будет вам полезна. Под *процессом* здесь можно понимать любую команду, приложение или службу, которые фактически выполняются в системе. Если вам захочется подробнее узнать о том, как устроены процессы, перечитайте главу 2 «Как работать с процессами».

В этой главе вы узнаете:

- Как управлять службами `systemd` с помощью команды `systemctl`.
- Как работает подсистема инициализации и как именно в этом качестве выступает программа `systemd`.
- Как эффективно работать с контейнерными окружениями (например, контейнерами Docker), в которых нет надежного механизма управления службами, подобного тому, который описывается в этой главе.

ПРИМЕЧАНИЕ

Значительная часть материала этой главы относится только к Linux, потому что в macOS и Windows (и даже в других системах семейства Unix) службами управляют другие инструменты. Даже в разных дистрибутивах Linux за это отвечают разные механизмы, хотя `systemd` — самый распространенный из них. Поэтому разработчикам особенно важно знать, как обращаться со службами в современных системах Linux.

Введение в службы Linux

Службы Linux — это фоновые процессы, которые функционируют в системе Linux и выполняют различные задачи. Они подобны службам в Windows или демонам в macOS.

В большинстве сред Linux, которые работают не в контейнерах, службами управляет программа `systemd`[1]. Взаимодействовать с ней вам помогут два инструмента:

- `systemctl` — управляет службами (в номенклатуре `systemd` они называются *модулями*).
- `journalctl` — позволяет работать с системными журналами.

В этой главе мы рассмотрим `systemctl`, а программе `journalctl` посвящена глава 16 «Журналы приложений»[2].

`systemd` — подсистема инициализации и диспетчер служб для Linux. Это стандартное решение, с помощью которого можно управлять службами в Linux. Сейчас `systemd` широко используется как подсистема инициализации по умолчанию в большинстве дистрибутивов Linux. Раньше многие дистрибутивы опирались на подсистемы SysV, которые произошли из Unix и до сих пор используются во многих современных операционных системах семейства Unix. В отдельных дистрибутивах, таких как Alpine или Gentoo Linux, в качестве подсистемы инициализации выступает OpenRC. Есть и много других подсистем инициализации, однако подавляющее большинство дистрибутивов Linux перешло на `systemd`. Эта подсистема позволяет запускать службы, останавливать их, включать и отключать их автозапуск при загрузке системы, а также просматривать их состояние. Каждой службе соответствует файл модуля, в котором точно описано, как `systemd` должна управлять этой службой.

Чтобы распоряжаться службами с помощью `systemd`, можно запускать такие элементарные команды (мы подробнее рассмотрим каждую из них далее в этой главе):

Команда	Назначение
`systemctl start` *`имя_службы`*	Запустить службу
`systemctl stop` *`имя_службы`*	Остановить службу
`systemctl restart` *`имя_службы`*	Перезапустить службу
`systemctl status` *`имя_службы`*	Вывести текущее состояние службы

[1] Названия служб Linux традиционно оканчиваются на *d* (сокращение от *daemon*). Название `systemd` можно интерпретировать как *system daemon*, то есть *системная служба*. — *Примеч. пер.*

[2] `ctl` — сокращение от *control* (*управление*). На `ctl` обычно оканчиваются названия программ, которые позволяют управлять той или иной службой или другим активом. — *Примеч. пер.*

Не забывайте, что управлять системными службами с помощью `systemd` можно только с правами `root` (например, если использовать `sudo`).

Подсистема инициализации

Давайте уделим чуть больше внимания распространенному термину, который будет вам часто встречаться. **Подсистема инициализации**, или **инициализатор**, в Linux — это первый процесс, который запускается, когда система загружается. Неудивительно, что ему присваивается PID 1. Инициализатор управляет процессом загрузки и отвечает за то, чтобы запустить все остальные процессы и службы, которые должны функционировать в системе. Также он обслуживает «осиротевшие» процессы (чей родительский процесс прекратил существование): он переподчиняет их самому себе, чтобы они продолжали нормально работать.

Как это свойственно практически всем компонентам в Linux, в роли подсистемы инициализации могут выступать разные конкретные программы, которые выполняют функции начальной загрузки, инициализации и координации. Как уже упоминалось, среди популярных инициализаторов Linux — подсистемы System V (SysV), OpenRC и `systemd`. Большинство современных систем Linux перешли на `systemd`, и поэтому здесь идет речь именно о ней.

От того, какую подсистему инициализации вы используете, зависит, как настраивать службы и управлять ими, так что имейте в виду, что весь материал этой главы относится именно к `systemd`.

Процессы и службы

Стоит хорошо понимать тонкое различие между процессами и службами. Службу можно рассматривать как обертку для программного компонента, которая предназначена для того, чтобы этим компонентом было удобнее управлять, когда он выполняется как процесс.

Служба добавляет полезные возможности, с помощью которых система обслуживает ту или иную программу (и процесс, который она порождает). Например, служба позволяет определить зависимости между разными процессами, задать последовательность запуска, добавить переменные окружения, с которыми запускается процесс, ограничить потребление ресурсов, регулировать права доступа и т. д. Кроме того, служба обеспечивает простое имя, по которому можно ссылаться на вашу программу. В главе 10 «Как настраивать конфигурацию приложений» вы научитесь создавать свои собственные службы, но до конца текущей главы мы ограничимся тем, что будем управлять существующими службами.

Команды systemctl

Рассмотрим практические примеры того, как с помощью `systemctl` управлять службами, которые функционируют в системе.

Как проверить состояние службы

Команда `systemctl status` *`имя_службы`* проверяет состояние службы и выводит набор данных, которые могут пригодиться, если придется устранять неполадки. Например, для службы веб-сервера nginx вывод этой команды выглядит так:

```
root@localhost:~# systemctl status nginx
• nginx.service - A high performance web server and a reverse proxy server
    Loaded: loaded (/lib/systemd/system/nginx.service; enabled;
            vendor preset: enabled)
    Active: active (running) since Mon 2023-02-06 21:47:43 UTC; 3min 54s ago
      Docs: man:nginx(8)
   Process: 1503 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on;
            master_process on; (code=exited, status=0/SUCCESS)
   Process: 1504 ExecStart=/usr/sbin/nginx -g daemon on;
            master_process on; (code=exited, status=0/SUCCESS)
  Main PID: 1598 (nginx)
     Tasks: 3 (limit: 4575)
    Memory: 5.4M
       CPU: 20ms
    CGroup: /system.slice/nginx.service
            ├─1598 "nginx: master process /usr/sbin/nginx -g daemon on;
            │       master_process on;"
            ├─1601 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" ""
            └─1602 "nginx: worker process" "" "" "" "" "" "" "" "" "" "" "" ""

Feb 06 21:47:43 localhost systemd[1]: Starting A high performance web server
                                      and a reverse proxy server...
Feb 06 21:47:43 localhost systemd[1]: Started A high performance web server
                                      and a reverse proxy server.
```

Давайте разберем этот плотный массив информации строка за строкой:

Компонент состояния	Обозначение в листинге	Описание
Имя службы	`nginx.service - A high performance web server and a reverse proxy server`	Имя службы в том виде, как оно определено в ее файле модуля
Состояние загрузки	`Loaded`	Успешно ли загружен файл модуля службы и готов ли он к запуску

Компонент состояния	Обозначение в листинге	Описание
Текущее состояние	`Active`	Текущее состояние службы: выполняется (*active*), приостановлена (*inactive*) или неисправна (*failed*), а также как долго служба находится в этом состоянии
Документация	`Docs`	Ссылка на основную страницу, где находится документация на службу (если она установлена)
Основной идентификатор процесса и дочерние процессы	`Main PID`	Идентификатор (PID) основного процесса, которому соответствует служба, а также дополнительные записи для всех дочерних процессов, которые были запущены
Потребление ресурсов	`Memory` `CPU`	Объем потребляемой памяти (Memory) и процессорное время (CPU)
Контрольная группа	`CGroup`	Сведения о контрольной группе, в которую входит процесс
Предпросмотр журналов	`Feb 06 21:47:43 localhost systemd[1] …`	Избранные записи журнала службы, которые дают представление о том, как она функционирует

Таким образом, этот вывод представляет собой подробную сводку сведений о службе и ее состоянии и может пригодиться, чтобы отлаживать службу или проверять ее работоспособность.

Если служба неисправна, то в выводе команды обычно содержится информация о сути проблемы, — например, код завершения или описание ошибки.

Как запустить и остановить службу

Вот простейшие команды, с помощью которых можно управлять службой:

Команда	Назначение	Примечания
`systemctl start` *`имя_службы`*	Запустить службу	Если служба уже запущена, эта команда не вызывает эффекта
`systemctl stop` *`имя_службы`*	Остановить службу	Если служба не запущена, эта команда не вызывает эффекта
`systemctl restart` *`имя_службы`*	Перезапустить службу	Эта команда эквивалентна двум последовательным командам: `systemctl stop` *`имя_службы`* `systemctl start` *`имя_службы`*

ПРИМЕЧАНИЕ

Будьте осторожны с командой `systemctl restart`. Если с тех пор, как служба была запущена, ее конфигурационный файл на диске изменился и в нем появился дефект, который мешает успешно запустить службу, то подкоманда `restart` благополучно остановит действующую службу, но не сможет запустить ее обратно.

Уже много лет разработчики то и дело страдают от этого закономерного, но потенциально нежелательного эффекта. Чтобы избежать страданий, прежде чем перезапускать службу, проверяйте, исправны ли ее конфигурационные файлы.

У многих популярных программ есть встроенные средства для того, чтобы проверять конфигурацию. Например, для nginx это можно сделать такой командой:

```
nginx -t
```

Как перезагрузить службу

Чтобы перезагрузить службу, укажите подкоманду `reload`:

```
systemctl reload имя_службы
```

Эту подкоманду поддерживают не все службы: чтобы она работала, нужно, чтобы ее реализовал специалист, который настраивает конфигурацию службы. Если у службы есть параметр `reload`, он обычно более безопасен, чем `restart`.

Как правило, `reload` выполняет такие операции:

- Перепроверяет конфигурационные файлы на диске, чтобы убедиться, что они исправны.
- Заново считывает конфигурацию в память, по возможности не прерывая действующий процесс.
- Перезапускает процесс только после того, как команда проверила конфигурацию и убедилась, что процесс успешно запустится после остановки.

Как это часто бывает в Linux, это скорее традиционные соглашения, чем обязательные требования. Вам вполне может встретиться программное обеспечение, которое им не следует, например:

- Не поддерживает подкоманду `reload`.
- Не реализует меры безопасности, перечисленные выше (проверку конфигурации и пр.).

- Делает с помощью `reload` вообще что-то другое, потому что разработчик или составитель пакета решил, что это хорошая идея.

В общем случае, когда вы обновляете конфигурационные файлы приложения, особенно в среде реальной эксплуатации, лучше перезапускать службы в режиме `reload`, чем `restart`.

Как управлять автозапуском служб

Команда `systemctl enable` *`имя_службы`* приводит к тому, что служба автоматически запускается при загрузке системы. Наоборот, команда `systemctl disable` *`имя_службы`* отключает автозапуск (если он был включен) и переводит службу в режим ручного управления.

Подкоманды `enable` и `disable` отличаются от `start` и `stop` тем, что две последние оказывают немедленный эффект: они обеспечивают запуск или остановку службы *прямо сейчас*. В отличие от этого, `enable` и `disable` влияют на то, как служба будет вести себя при следующих запусках системы, и не меняют текущее состояние службы на тот момент, когда запускается команда.

Разработчики часто совершают ошибку, когда предполагают, будто подкоманда `enable` запустит службу. На самом деле это не так. Например, если вы хотите запустить веб-сервер nginx прямо сейчас, а также настроить, чтобы он автоматически запускался при каждой перезагрузке системы, нужно применить две команды:

```
systemctl start nginx
systemctl enable nginx
```

Чтобы облегчить эту процедуру, подкоманды `enable` и `disable` обычно могут принимать необязательный флаг, который немедленно запускает службу (или останавливает ее в случае `disable`). Например, вместо предыдущей последовательности из двух команд можно обойтись одной командой:

```
systemctl enable --now nginx
```

Службы Linux и контейнеры Docker

Хотя `systemctl` стала стандартным инструментом для того, чтобы управлять службами в традиционных системах Linux, эта программа обычно не используется в контейнерах Docker из-за их изолированной и замкнутой природы.

В идеале в контейнере Docker выполняется всего один процесс, и поэтому ему не требуется сложная стадия загрузки или диспетчер процессов. По сути, контейнер — *это и есть процесс*, и у него нет доступа к подсистеме инициализации управляющей системы (в том числе к `systemd`).

Хотя технически можно сделать так, чтобы в контейнерах запускались команды из этой главы, обычно нежелательно заводить в них какие-либо системы управления службами.

Мы не рекомендуем настраивать контейнеры Docker так, чтобы в них выполнялись множественные процессы или применялись особые механизмы управления службами. Перефразируя классика, можно сказать, что все удачные образы Docker похожи друг на друга, а каждая неудачная конфигурация Docker неудачна по-своему.

Итоги

Из этой главы вы узнали о том, как устроены службы в Linux и какие практические команды позволяют ими управлять. Теоретическая часть этой главы поможет вам ориентироваться в понятиях, с которыми вам предстоит иметь дело в реальных системах: что такое подсистема инициализации, какую роль в Linux играет `systemd` и какие команды нужны, чтобы с ней взаимодействовать.

В следующей главе мы продемонстрируем несколько полезных приемов работы с командной оболочкой и историей команд, благодаря которым вы сэкономите немало времени и почувствуете себя могущественным волшебником из своего любимого фильма. (Помимо этого вы научитесь быстрее и эффективнее справляться с повседневной работой, просто эта перспектива звучит не так привлекательно.)

4

История команд в оболочке Linux

Чтобы в совершенстве овладеть командной строкой, нужно регулярно практиковаться. В освоении Linux нет коротких путей к успеху, но есть несколько чрезвычайно полезных приемов, которые стоит взять на вооружение как можно раньше, чтобы сэкономить немало времени и нервов. Чем скорее вы закрепите эти приемы в мышечной памяти, тем лучше.

Из этой главы вы узнаете, как пользоваться историей команд, чтобы не приходилось утомительно перепечатывать те, которые вы уже запускали. Кроме того, вы научитесь настраивать поведение и внешний вид оболочки на своем компьютере с помощью конфигурационного файла. Наконец, вы познакомитесь с наиболее полезными сочетаниями клавиш для того, чтобы редактировать команды в командной строке. Если вы прочно овладеете материалом этой главы, то сможете работать в командной строке с феноменальной скоростью.

Мы рассмотрим такие темы:

- История команд в оболочке.
- Как вызывать предыдущие команды с помощью восклицательного знака (!).
- Как перемещаться в начало или конец строки.

Давайте начнем с того, что такое история команд.

История команд в командной оболочке

В большинстве командных оболочек сохраняется история команд, которые в ней выполнялись. История позволяет заново помещать в командную строку любые

команды, которые вы раньше уже успешно запускали. Для этого служат клавиши со стрелками: ↑ позволяет перейти на одну команду назад, а ↓ — на одну команду вперед. Такая прокрутка истории заметно облегчает работу, особенно если вы часто запускаете идентичные или похожие команды.

Обратите внимание, что команды, которые вы прокручиваете таким образом, можно модифицировать: просто нажмите ← или →, чтобы в командной строке появился курсор, и редактируйте команду как обычно.

Если вы модифицировали команду, она добавляется в конец истории, а исходная команда при этом не меняется и остается на прежнем месте.

Эти приемы в сочетании друг с другом позволяют легко перемещаться по списку команд и заново вызывать или редактировать любые из них.

Конфигурационные файлы оболочки

Для некоторых практических задач из этой главы требуется вносить изменения в конфигурационный файл оболочки. Обычно это делается так:

1. Отредактировать конфигурационный файл, изменив нужный параметр.
2. Сохранить файл.
3. Запустить новый сеанс командной оболочки, чтобы увидеть изменения.
4. Чтобы обновить конфигурацию в текущем сеансе, запустите команду `source`[1], которая заставит оболочку заново прочитать свой конфигурационный файл:

```
source ~/путь/к/конфигурационному/файлу
```

Вот где хранят свою конфигурацию две популярные командные оболочки:

- **Bash** использует два конфигурационных файла:
  - `~/.bashrc`[2] — для интерактивных сеансов: например, если вы открываете новое окно терминала в графической среде. Именно этот файл нужен практически всегда, когда вы собираетесь изменить конфигурацию на своем рабочем компьютере. Слово «интерактивный» здесь означает, что команды в терминал вводите вы сами (чаще всего с клавиатуры), а не программный сценарий (например, если его автоматически запустило задание `cron`). Будем считать, что вы находитесь в интерактивной оболочке, если перед

[1] *Источник. — Примеч. пер.*

[2] Буквы `rc` в названиях подобных файлов происходят от исторического сокращения `runcom`, которое, в свою очередь, означало *run commands* (*запускать команды*). В наше время `rc` иногда расшифровывают также как *run configuration* (*запустить конфигурацию*) или *run control* (*запустить управление*). — *Примеч. науч. ред.*

вами открыт терминал с приглашением командной строки, куда можно вручную вводить команды.

- `~/.bash_profile` — для оболочек входа в систему. К таким оболочкам относится как механизм, с помощью которого пользователь вручную вводит имя и пароль на локальном компьютере, так и средства аутентификации по SSH.

- В оболочке **Zsh** конфигурационный файл это `~/.zshrc`.

Файлы истории

Разные оболочки хранят свои файлы истории в разных местах, и обычно эти места можно настроить. В оболочке Bash, с которой вам, скорее всего, предстоит особенно активно работать, история хранится в файле `~/.bash_history`.

Если вы не знаете точно, где искать файл истории, во многих оболочках вам поможет конфигурационная переменная `HISTFILE`, которая содержит эту информацию.

Например, вот как можно узнать, где находится файл истории команд в оболочке Zsh:

```
% echo $HISTFILE
/home/dcohen/.zsh_history
```

В Bash есть две настройки, благодаря которым файл истории не разрастается до бесконечности. Они ограничивают его до такого размера, когда его удобно поддерживать, а поиск по истории выполняется быстро:

- `HISTSIZE` — максимальное количество записей истории команд, которое хранится в памяти.
- `HISTFILESIZE`[1] — максимальное количество записей, которое сохраняется в файле истории команд между сеансами.

Если вам нужно увеличить эти ограничения[2], внесите соответствующие изменения в конфигурационный файл оболочки. Для этого откройте этот файл (например, `~/.bashrc`) в текстовом редакторе и добавьте в конец стро́ки такого вида:

```
export HISTSIZE=1000
export HISTFILESIZE=5000
```

---

[1] Во фрагментах кода в этом разделе `HIST` — от *history* (*история*, *хронология*), `SIZE` — *размер*, `FILESIZE` — от *file size* (*размер файла*). — *Примеч. пер.*

[2] По умолчанию в Bash значения `HISTSIZE` и `HISTFILESIZE` составляют 500 записей. — *Примеч. науч. ред.*

Поиск по истории команд

Вам часто понадобится находить команды, которые вы запускали неделю (или месяц) назад. Скорее всего, такие команды зарыты уже очень глубоко в истории, и вы впустую потратите время, если будете нажимать ↑ сотни раз, чтобы добраться до них. Если вы *хоть как-то* представляете себе, что ищете, то вам поможет интерактивный поиск по истории команд. Вот как им воспользоваться:

1. Нажмите Ctrl+R, чтобы запустить утилиту поиска `reverse-i-search`[1].
2. Введите часть команды, которую вы ищете.
3. Оболочка постарается сопоставить символы, которые вы вводите, с историей команд и предложить самое точное и самое свежее совпадение.
4. Если нужно перейти к предыдущему совпадению, снова нажмите Ctrl+R. Это сочетание клавиш можно нажимать многократно, чтобы перебирать совпадения от более новых к более старым.
5. Если вы случайно проскочили мимо нужной команды, нажмите Ctrl+Shift+R, чтобы вернуться на одно совпадение назад.
6. Когда вы добрались до нужной команды, нажмите Enter, чтобы запустить ее.
7. Чтобы выйти из режима поиска, нажмите Esc.

Исключения поиска

В зависимости от командной оболочки и ее текущей конфигурации из описанной процедуры бывают исключения.

Некоторые оболочки не запоминают команды, которые завершились с ошибкой (то есть с кодом завершения, отличным от `0`). Также многие оболочки не вносят в историю команды, которые начинаются с символа пробела, — соответственно, такие команды не будут учитываться при поиске. Однако в любом случае по-прежнему можно нажать ↑, чтобы немедленно переместиться к предыдущей команде, не выполняя никаких других команд.

Как запустить предыдущую команду с помощью восклицательного знака (!)

Восклицательные знаки позволяют повторно запускать предыдущие команды. Посмотрим, как по-разному использовать эту возможность.

---

[1] От *reverse intelligent search* (*интеллектуальный обратный поиск*). Поиск называется *обратным*, потому что по умолчанию перебирает совпадения в обратном хронологическом порядке. — *Примеч. пер.*

Как повторно запустить команду с прежними аргументами

Если добавить один восклицательный знак (`!`) перед именем команды, то запустится последняя команда с этим именем и с прежними аргументами. Например, если ввести в командную строку `!ssh`, то оболочка найдет предыдущую команду `ssh`, которую вы запускали, и запустит ее снова с теми же аргументами. Это полезно для повтора часто используемых команд, например, чтобы быстро переподключиться к серверу SSH, с которым вы работаете каждый день.

Как включить предыдущую команду в состав текущей

Если в команде есть два восклицательных знака (`!!`), то оболочка подставит на их место всю предыдущую команду целиком. Это может показаться чем-то экзотическим, но такая возможность *невероятно* полезна в ситуациях, когда вы запустили команду, которая требует прав доступа `root`, но забыли приписать перед ней `sudo`:

```
apt-get install nginx  # ошибка прав доступа
sudo !!
# выполняется такая команда:
sudo apt-get install nginx
```

Это означает, что после того как команду не удалось выполнить из-за недостаточных прав, можно просто ввести `sudo !!`, и та же команда запустится в режиме `sudo`.

ПРИМЕЧАНИЕ

Из соображений безопасности не превращайте это в бездумную привычку. Всегда убеждайтесь, что вы понимаете, зачем команде нужны повышенные права, и задумывайтесь о том, достаточно ли вы ей доверяете, чтобы позволить вытворять в системе буквально что угодно. Если чересчур легкомысленно применять `sudo`, немудрено испортить что-то важное или впустить в систему злоумышленника.

Как перейти в начало или конец текущей строки

Когда вы вводите команду, нередко требуется перейти в начало строки, например, чтобы исправить опечатку или добавить обязательный аргумент. Для этого существует сочетание клавиш Ctrl+A.

Подобным образом, чтобы перейти в конец строки, нажмите Ctrl+E.

Думаем, что два этих сочетания клавиш будут выручать вас довольно часто[1].

[1] Чтобы полностью очистить командную строку, нажмите Ctrl+U. — *Примеч. науч. ред.*

Итоги

Работая в командной оболочке Linux, приходится много печатать на клавиатуре. Если вы научитесь набирать команды хотя бы немного быстрее и делать немного меньше ошибок, это может вызвать грандиозный эффект: вместо того чтобы сетовать, что элементарные задачи отнимают уйму времени, вы будете чувствовать себя волшебником и повелителем Linux'а!

В этой главе мы рассмотрели самые популярные и мощные сочетания клавиш, которые могут пригодиться в повседневной работе. Когда вы объедините свои новые навыки поиска по истории командной оболочки с изученными сочетаниями клавиш для ввода и редактирования команд, вы почувствуете себя гораздо увереннее и будете работать быстрее и эффективнее.

5

Файлы в Linux

В Linux любой объект является файлом (или может быть представлен как файл). Файлы организованы в файловую систему: это не что иное, как иерархия файлов и каталогов (а каталоги — это просто специальный тип файлов). Практически все, чем вы занимаетесь в Linux как разработчик, требует разбираться в файлах: эти знания нужны, чтобы писать и копировать исходный код, создавать образы Docker, налаживать журналы приложений, настраивать зависимости и выполнять многие другие задачи.

Эта глава посвящена тому, как устроены файлы в Linux. Вы узнаете, в чем разница между текстовыми и двоичными файлами, — это два основных типа содержимого файлов, с которыми вам предстоит работать. Вы увидите, как из файлов складывается «дерево» файловой системы Linux, а затем познакомитесь с командами, которые позволяют создавать, редактировать, перемещать и удалять файлы. После того как вы овладеете этими основами, вас ждет практическая работа, где вы будете редактировать файлы с помощью наиболее распространенных текстовых редакторов с интерфейсом командной строки.

Впрочем, в этой главе мы не ограничимся элементарными знаниями. Файлы в Linux — это одна из тем, владение которыми на продвинутом уровне окупается с лихвой. В конце концов, работа с файлами — это одно из главных занятий разработчиков: вам понадобится очень часто записывать и читать файлы с исходным кодом и конфигурацией, искать в файлах то или иное содержимое, копировать и перемещать файлы журналов и выполнять другие файловые операции. Чем увереннее вы освоите необходимые основы, тем востребованнее вы будете как грамотный специалист, который не перелопачивает поисковые системы по поводу каждой команды Linux и которому не грозит неловко замешкаться в консольном текстовом редакторе посреди рабочего видеосозвона для устранения неполадок.

Мы поговорим о том, как искать файлы в файловой системе и как искать внутри файлов определенное содержание или совпадения с шаблонами. Затем мы рас-

смотрим специальные файлы и альтернативные файловые системы, с которыми вы можете встретиться: вы узнаете, как эффективно с ними работать.

К концу главы вы овладеете такими темами:

- Какие типы файлов встречаются в Linux и зачем они нужны.
- С какими важнейшими типами содержимого файлов вам предстоит иметь дело.
- Как устроена файловая система Linux и какие команды позволяют с ней работать.
- Как редактировать файлы.
- Какие распространенные проблемы связаны с файлами и как их избежать.

В этой главе много материала, и он станет одним из краеугольных камней вашего мастерства в области Linux. Убедитесь, что вы хорошо усвоили каждую предыдущую тему, прежде чем переходить к следующей. Если вы читаете эту главу впервые, вам не обязательно заучивать все подряд, но постарайтесь по ходу чтения как можно больше практиковаться в своей среде Linux. Ваши усилия оправдаются сполна, когда вам придется устранять неполадки в реальной ситуации или проходить собеседование.

Самое главное о файлах в Linux

Чтобы разбить объемную тему про файлы в Linux на удобоваримые порции, давайте рассмотрим самые базовые понятия, с которыми вы наверняка уже знакомы в той или иной степени: текстовые и двоичные файлы. Заодно вы узнаете, как бороться с распространенной ошибкой, которая может возникнуть, если вы переносите файлы из Windows в Linux и наоборот.

Текстовые файлы

Текстовые файлы (более официально — файлы в формате обычного текста) — это один из простейших форматов файлов, с которыми вам доводится работать. Хотя исторически это были файлы в кодировке ASCII[1], сейчас они обычно представлены в UTF-8. Возможно, вам встретятся текстовые файлы в других кодировках, но это редкость, потому что остальные кодировки считаются устаревшими.

---

[1] Кодировка ASCII поддерживала только латинский алфавит, а если в текстовых файлах требовались альтернативные алфавиты, эти файлы представлялись в других кодировках (совместимых с ASCII в диапазоне обычной латиницы). Например, для русского языка в Linux и других системах семейства Unix была особенно популярна кодировка KOI8-R, а в Windows — кодировка Windows-1251. — *Примеч. науч. ред.*

Двоичные файлы

В отличие от многих других операционных систем, Unix не делает различий между двоичными и текстовыми файлами. Любые файлы можно передавать по конвейеру, редактировать и дополнять. Файл — это просто последовательность байтов. Если файл помечен как исполняемый, Unix приложит все усилия, чтобы его выполнить. Это завершится либо успехом — если файл соответствует формату **ELF (Executable and Linkable Format**[1], пожалуй, самый популярный сейчас формат исполняемых файлов), либо неудачей (например, если попытаться «выполнить» изображение или аудиозапись).

Этот простой механизм открывает потрясающие возможности. Например, исполняемый файл можно передать по конвейеру утилите сжатия, затем отправить по сетевому туннелю (например, SSH), а после этого разархивировать и снова записать в файл, и все это одной командой, без промежуточных временных файлов.

Однако из этого вытекает, что стоит принимать меры предосторожности, чтобы не создать ситуацию, когда система попытается исполнить первый попавшийся файл — например, который загрузили или изменили пользователи сайта (это относится и к файлам журналов!). Если это не проконтролировать, то можно столкнуться с серьезными проблемами безопасности.

Переводы строк

Хотя в системах семейства Unix файлы (в том числе текстовые) ведут себя примерно так же, как в других операционных системах, обратите внимание, что в Windows (как ранее в DOS) перевод строки обозначается иными символами[2]. Эта разница иногда вызывает ошибки в программах, которые оперируют текстовыми файлами. Хотя эта проблема проявляется, только если файлы создаются в одной системе, а затем переносятся в другую (например, из Linux в Windows), об этом риске не стоит забывать.

Разнобой с переводами строк сложился по историческим причинам, причем многие современные инструменты (например, Git и различные текстовые редакторы) автоматически распознают и правильно обрабатывают переводы строк в любых форматах. Тем не менее в особых случаях вам может понадобиться преобразовывать файлы вручную. Для этого существуют классические команды вроде `dos2unix`, но в большинстве систем семейства Unix их нужно специально устанавливать.

---

[1] *Исполняемый и компонуемый формат. — Примеч. пер.*

[2] В Unix, Linux и современных версиях macOS перевод строки обозначается символом LF (U+000A, Line Feed), а в классической Mac OS (до версии 9) — символом CR (U+000D, Carriage Return). В Windows (как ранее в DOS) перевод строки обозначается последовательностью из двух символов — CR и LF. — *Примеч. науч. ред.*

Вот как можно преобразовывать переводы строк из формата Windows в формат Unix с помощью встроенных утилит:

- С помощью `sed`: `sed 's/^M$//' файл_windows > файл_unix`.
- С помощью `tr`[1]: `tr -d '\r' < файл_windows > файл_unix`.
- С помощью `perl`: `perl -pi -e 's/\r\n/\n/g' файл` (эта команда не создает новый файл, а преобразовывает переводы строк прямо в существующем файле).

После того как мы рассмотрели важнейшие понятия, связанные с файлами в системах семейства Unix, давайте поговорим о контексте, в котором все эти файлы фактически существуют: о файловой системе Linux.

Дерево файловой системы

Общепринятую структуру каталогов в системах, подобных Unix, описывает стандарт **FHS (Filesystem Hierarchy Standard**[2]). Linux соответствует этому стандарту, так что FHS можно считать официальной иерархией каталогов в Linux. Это стандартизованная древовидная структура, где все файлы и каталоги «произрастают» из корневого каталога, имя которого — просто символ «слеш»: `/`. Эта иерархия крайне важна: хотя конечным пользователям ничто не мешает изобретать свою собственную структуру каталогов, у каждого каталога внутри `/` (корневого каталога) есть свое официальное назначение.

Выучить общую структуру файловой системы не так уж сложно, и если вы сейчас потратите на это несколько минут, то сможете интуитивно представлять себе, куда поместить те или иные файлы, будь то исполняемые файлы приложений, журналы, файлы с данными или внешние устройства, к которым вашему коду нужен доступ. Иными словами, стандартная структура облегчает как разработку, так и устранение неполадок: если вы знаете, где *должен быть* каждый файл или каталог, то в случае неприятностей будете меньше блуждать в потемках и быстрее поймете, на какой участок файловой системы обращать внимание. Кроме того, эти знания понадобятся вам для того, чтобы писать свои собственные сценарии и выполнять простейшие задачи системного администрирования, без которых не обходится ни один серьезный разработчик.

---

[1] `tr` — историческая команда Unix, которая передает файл из стандартного ввода в стандартный вывод, заменяя или удаляя заданные символы. Название команды — сокращение от *translate* (*переводить*, *преобразовывать*) или *transliterate* (*транслитерировать*). — *Примеч. науч. ред.*

[2] *Стандарт иерархии файловой системы. — Примеч. пер.*

Вот некоторые важные места файловой системы, на которые ссылаются во многих источниках и с которыми вам самим предстоит работать[1]:

`/etc`	Конфигурационные файлы операционной системы и прикладного ПО, организованные во множество каталогов
`/bin` `/sbin`	Системные исполняемые файлы. Не трогайте эти каталоги, если не знаете, что делаете
`/usr/bin` `/usr/local/bin`	Установленное вами ПО и ваши собственные исполняемые файлы; их может просматривать и запускать любой пользователь системы
`/var/log` `/var/lib`	В каталоге `/var` находятся изменчивые данные, которые могут модифицироваться во время работы системы, например журналы приложений (`/var/log`) и динамические библиотеки (`/var/lib`), а также другие ресурсы, которые отражают состояние запущенных приложений
`/var/lib/systemd`	Одно из нескольких мест файловой системы, где хранится конфигурация подсистемы инициализации `systemd`
`/etc/systemd/system`	Подходящее место для пользовательских файлов подсистемы инициализации, если вы создаете собственные службы
`/dev`	Выделенная ветвь файловой системы, которая представляет аппаратные устройства
`/proc`	Выделенная ветвь файловой системы, которая представляет процессы и позволяет просматривать или редактировать состояние системы

Основные операции с файлами и каталогами

Настало время более детально изучить ключевые команды Unix, которые каждому разработчику приходится запускать особенно часто. Эти команды охватывают широкий диапазон основных задач, с которыми вам предстоит работать в любой

[1] В именах каталогов из этой таблицы: *etc* — от английского выражения латинского происхождения *et cetera* (*и так далее*), сейчас также считается сокращением *editable text configurations* (*редактируемые файлы конфигурации в текстовом формате*); *bin* — сокр. *binaries* (*исполняемые файлы*), *s* в *sbin* — сокр. *system* (*система*) или *superuser* (*суперпользователь*; в каталоге `/sbin` обычно находятся ключевые системные программы, которые запускаются с правами `root`); *usr* — от *user* (*пользователь*); *local* — *местный*, *локальный* (в `/usr/local` обычно расположены программы, которые не входят в дистрибутив ОС); *log* — *журнал*, *lib* — сокр. *library* (*библиотека*), *dev* — сокр. *devices* (*устройства*). — *Примеч. пер.*

системе. После того как вы изучите и закрепите на практике команды из этой главы, вы сможете:

- отслеживать журналы своего приложения в режиме реального времени;
- исправлять конфигурационные файлы, когда из-за ошибок в них приложение не запускается;
- перемещать файлы из одного каталога в другой в репозитории Git в своей локальной среде разработки на macOS.

Давайте начнем с того, как просмотреть содержимое каталога. Убедитесь, что вы успешно вошли в систему Linux или Unix (например, прекрасно подойдет Ubuntu или macOS) и открыли приложение «Терминал», чтобы воспроизводить примеры из этой главы.

ls — содержимое каталога

Команда `ls` выводит сведения о файле или о содержимом каталога. Это похоже на то, как вы открываете папку в графическом пользовательском интерфейсе. По умолчанию команда вызывается для текущего каталога и перечисляет все, что в нем находится:

```
/home/steve# ls
my_document.txt
```

В этом примере командная оболочка вызывается в текущем каталоге `/home/steve`, где расположен всего один файл `my_document.txt`.

Команда `ls` может вывести содержимое любого каталога в системе: для этого путь к каталогу нужно передать в качестве аргумента:

```
/home/steve# ls /var/log/
alternatives.log  apt  bootstrap.log  btmp  dpkg.log  faillog  lastlog
wtmp
```

Чтобы вывод стал информативнее, можно добавить параметр `-l`, который заставляет `ls` выделять для каждого файла или каталога отдельную строку и выводить дополнительные сведения.

```
# ls -l /var/log/
total 296
-rw-r--r-- 1 root root   4686 Jun 24 02:31 alternatives.log
drwxr-xr-x 2 root root   4096 Jun 24 02:31 apt
-rw-r--r-- 1 root root  64547 Jun 24 02:06 bootstrap.log
-rw-rw---- 1 root utmp      0 Jun 24 02:06 btmp
-rw-r--r-- 1 root root 177139 Jun 24 02:31 dpkg.log
-rw-r--r-- 1 root root  32032 Oct 28 14:26 faillog
-rw-rw-r-- 1 root utmp 296296 Oct 28 14:26 lastlog
-rw-rw-r-- 1 root utmp      0 Jun 24 02:06 wtmp
```

Таким образом, команда `ls` позволяет «осматриваться» в файловой системе Unix.

pwd — текущий каталог

Команда `pwd` показывает, где вы находитесь в файловой системе, то есть какой каталог сейчас является текущим для сеанса командной оболочки. Если я вошел в систему Linux как пользователь `steve` и нахожусь в своем домашнем каталоге, то `pwd` выводит такой результат:

```
pwd
/home/steve
```

cd — сменить текущий каталог

Команда `cd` меняет текущий каталог командной оболочки. Команды, которые вы запускаете после `cd`, будут выполняться в новом местоположении.

Рассмотрим для примера каталог, с которым мы уже встречались:

```
Desktop
├── anotherfile
├── documents
│   └── contract.txt
├── somefile.txt
└── stuff
    ├── nothing
    └── important
```

Если вы находитесь в каталоге `Desktop`, но затем запускаете команду `cd documents`, чтобы перейти в каталог `documents`, то после этого команда `ls` выведет содержимое второго каталога. Убедитесь сами:

```
/home/steve/Desktop# ls
anotherfile documents somefile.txt stuff

/home/steve/Desktop# cd documents
/home/steve/Desktop/documents# ls
contract.txt
```

Итак, мы умеем осматриваться в файловой системе (`ls`), перемещаться по ней (`cd`) и узнавать, где мы находимся (`pwd`). Давайте теперь разберемся, как вносить изменения в файловую систему, создавая и редактируя файлы.

touch — создать пустой файл или обновить метки времени для существующего файла

Эта команда записывается в формате `touch` *`путь_к_файлу`* и выполняет одну из двух операций в зависимости от того, существует ли уже файл по этому пути:

1. Если файла по указанному пути не существует, `touch` создает его:

```
→ /tmp touch файл
→ /tmp ls -l файл
-rw-r--r--  1 dcohen  wheel  0 Aug  7 16:02 файл
```

2. Если файл по указанному пути уже есть, `touch` обновляет дату и время последнего доступа и последнего изменения файла:

```
→ /tmp touch файл
→ /tmp ls -l файл
-rw-r--r--  1 dcohen  wheel  0 Aug  7 16:03 файл
```

Обратите внимание, что в показанных свойствах файла изменилось только время последнего изменения.

less — постраничный просмотр файла

Команда `less` — это так называемый *пейджер*, то есть программа, которая позволяет просматривать содержимое файла по одному экрану (странице) за раз:

```
less /etc/hosts
```

Просмотр работает в интерактивном режиме, где вы можете:

- прокручивать содержимое файла построчно вверх или вниз с помощью колесика мыши или клавиш со стрелками ↑ и ↓;
- прокручивать вниз постранично с помощью пробела;
- искать содержимое в файле: /, затем шаблон для поиска, затем Enter;
- перейти к следующему совпадению: N;
- выйти из пейджера: Q.

Не поленитесь потратить пару минут, чтобы поупражняться во всех этих операциях.

tail — просмотреть последние строки файла

Команда `tail`[1] позволяет просматривать несколько последних строк файла[2].

```
tail путь_к_файлу
```

---

[1] Букв. *хвост*, перен. *заключительная часть чего-л. — Примеч. пер.*

[2] По умолчанию `tail` выводит 10 последних строк файла. Чтобы вывести другое количество строк, используйте параметр `-n` (или `--lines`), например: `tail -n 25 путь_к_файлу`. *— Примеч. науч. ред.*

Параметр `-f` (или `--follow`[1]) особенно полезен для того, чтобы выводить дополняемые файлы журналов в режиме реального времени:

```
tail -f /var/log/some.log
```

Чтобы выйти из режима просмотра, нажмите Q.

mv — переместить или переименовать файл или каталог

С помощью команды `mv` можно перемещать и/или переименовывать файлы и каталоги.

Перемещение

Допустим, у вас есть файл `somefile.txt`:

```
→  ls -alh somefile.txt
-rw-r--r--  1 dcohen wheel   0B Aug  7 11:02 somefile.txt
```

Если вы находитесь в том же каталоге, что и этот файл, то следующей командой его можно переместить в каталог `/var/log`, не переименовывая:

```
mv somefile.txt /var/log/
```

Переименование

А теперь посмотрим, как переименовать тот же файл `somefile.txt` в `foobar.txt`:

```
mv /var/log/somefile.txt /var/log/foobar.txt
```

Вот и все[2]!

cp — копировать файлы и каталоги

Чтобы копировать файлы и каталоги, используйте команду `cp`[3].

Команда `cp` *`файл путь_к_каталогу`* копирует файл с именем *`файл`* в каталог, которому соответствует *`путь_к_каталогу`*. Каталоги обычно копируются с флагом `-r`

---

[1] *Следовать, сопровождать. — Примеч. пер.*

[2] Команда `mv` также позволяет одновременно переместить файл и переименовать его. Например, две описанные выше операции можно выполнить одной командой: `mv somefile.txt /var/lob/foobar.txt`. — *Примеч. науч. ред.*

[3] Название команды `cp` — сокр. *copy* (*копировать*). Обратите внимание, что эта команда, как и все прочие команды Unix и Linux, записывается английскими буквами, в данном случае *c* («си») и *p* («пи»), а не русскими *с* («эс») и *р* («эр»). — *Примеч. пер.*

(`--recursive`[1]): это позволяет скопировать не только сам каталог, но и все его содержимое.

```
cp -r /home/dave /storage/userbackups/
```

mkdir — создать каталог

Чтобы создать новый пустой каталог с именем *`каталог`*, запустите такую команду:

```
mkdir каталог
```

Популярный параметр `-p` позволяет создавать вложенные каталоги одной командой. Например, чтобы в текущем каталоге создать каталог `Документы`, в который вложен каталог `учеба`, в который вложен каталог `задания`, можно запустить такую команду[2]:

```
mkdir -p Документы/учеба/задания
```

rm — удалить файл или каталог

Команда `rm` удаляет файлы и каталоги, например:

`rm` *`файл`*	Удалить файл с именем *`файл`*
`rm -r` *`каталог`*	Удалить каталог с именем *`каталог`*, а также рекурсивно удалить все файлы и каталоги внутри него

Существует также отдельная команда `rmdir`, которая удаляет пустые каталоги, но ее обычно используют только в сценариях командной оболочки, когда разработчики заботятся о том, чтобы максимально сузить «зону поражения», если сценарий начнет удалять не те файлы, которые надо.

Как редактировать файлы

Если вы обновляете конфигурационные файлы, настраиваете новые службы Linux или делаете заметки, когда устраняете неполадки, в этих и многих других случаях вам требуется редактировать файлы с помощью командной строки. Мы под-

---

[1] *Рекурсивный. — Примеч. пер.*

[2] Этот пример показывает, что в именах файлов и каталогов в Linux можно использовать не только латиницу, но и кириллицу, а также другие алфавиты. Однако общепринятая практика заключается в том, чтобы по умолчанию ограничиваться обычными латинскими буквами, а прочие символы применять только при крайней необходимости. — *Примеч. науч. ред.*

робно рассмотрим эту тему в главе 6 «Как редактировать файлы из командной строки», но сейчас предложим вашему вниманию краткий обзор.

Если в вашем распоряжении есть только интерфейс командной строки, в нем можно использовать несколько специализированных текстовых редакторов, например:

nano	Установлен или легко устанавливается практически везде; прост в использовании
vi	Установлен практически везде; понадобится немного практики, чтобы с ним освоиться
vim	Легко устанавливается в любой системе; более функционален, чем `vi`

Если какого-то из этих редакторов нет в вашей системе, его можно установить с помощью службы управления пакетами. Например, если вы работаете в Ubuntu Linux, то можете запустить команду `sudo apt-get install nano` (или указать `vim` вместо `nano`). (На том, как управлять пакетами, мы подробнее остановимся в главе 9 «Как устанавливать программное обеспечение».) Независимо от того, какой редактор вы предпочтете, открыть файл для редактирования можно командой *редактор имя_файла*, например:

```
vi имя_файла
vim путь/к/файлу
nano путь/к/файлу
```

Эта команда приводит к одному из трех результатов:

- Если указанный файл существует, он откроется для редактирования в указанном редакторе.
- Если файла не существует, но существует указанный каталог, то файл будет создан в этом каталоге после того, как вы первый раз сохраните файл в редакторе.
- Если указанного каталога не существует, то вы все равно сможете редактировать файл, но редактор не сохранит его в файловой системе без некоторых дополнительных мер.

В следующей главе 6 «Как редактировать файлы из командной строки» мы во всех подробностях представим практические навыки, с помощью которых вы сможете редактировать файлы из командной строки Linux. Впрочем, если вам понадобилось во что бы то ни стало отредактировать файл до того, как вы освоите эту главу, просто запустите команду `nano путь/к/файлу` и следуйте подсказкам в нижней части экрана, чтобы сохранить файл и выйти из редактора. Тем временем давайте узнаем, с какими типами файлов вам как разработчикам придется встретиться в Linux.

Типы файлов

Мы уже говорили про «обычные» файлы, такие как простые текстовые или файлы с двоичными данными (например, изображения или исполняемые программы). Но бывают и другие типы файлов, о которых важно знать, чтобы успешно работать с ними в Linux. Когда вы подключаете флешку или новую клавиатуру, создаете ссылку на файл или разбираетесь, какие сетевые сокеты открыл процесс вебслужбы, вам приходится иметь дело со всеми этими типами файлов.

Вот какие типы файлов бывают в Linux и их назначение:

- **Обычный файл.** Это самый распространенный тип файлов, который содержит текстовые или двоичные данные. Программисты сталкиваются с обычными файлами почти в каждой задаче разработки: когда пишут код, редактируют конфигурационные файлы или запускают программы. Вот типичный пример — файл с исходным кодом, который может встретиться вам в расширенном выводе команды `ls`. Дефис (`-`), с которого начинается строка, обозначает, что это обычный файл:

```
-rw-r--r-- 1 dave dave 210 Jan 04 09:30 main.c
```

- **Каталог.** Каталоги — это особые файлы, с помощью которых можно структурировать другие файлы и каталоги. Если вы работали с Windows или macOS, то уже знакомы с каталогами: в этих системах они называются *папками* и содержат другие файлы и папки. Вот как может выглядеть каталог в перечне файлов и каталогов; обратите внимание на букву `d`[1] в самом начале:

```
drwxr-xr-x 5 root root 4096 Jan 04 09:21 /etc
```

- **Блочное устройство.** Этот специальный тип файлов обеспечивает буферизованный доступ к оборудованию, что бывает особенно полезно для аппаратных устройств вроде жестких дисков, где обмен данными происходит крупными блоками фиксированного размера. Вам редко понадобится обращаться к таким файлам напрямую, кроме ситуаций, когда вы монтируете файловую систему. В качестве примера рассмотрим раздел жесткого диска; блочное устройство обозначается буквой `b`[2] в начале строки:

```
brw-rw---- 1 root disk 8, 2 Jan 19 11:00 sda2
```

- **Символьное устройство.** Этот тип файлов подобен блочному устройству, но обеспечивает небуферизованный потоковый доступ к оборудованию, в первую очередь к таким устройствам, как клавиатура или мышь, которые не группируют данные в блоки. Чаще всего вам не придется иметь дело с такими файла-

---

[1] От *directory* (*каталог*). — *Примеч. пер.*

[2] От *block* (*блок*). — *Примеч. пер.*

ми, хотя иногда они встречаются в работе, — скажем, `/dev/urandom`, `/dev/null` или `/dev/zero`. Например, вот как выглядит в перечне такое символьное устройство, как терминал (строка начинается с буквы `c`[1]):

```
crw-rw-rw- 1 root tty 5, 1 Jan 19 22:00 /dev/tty1
```

- **Именованный канал** (или **именованный конвейер**)[2]. Такие типы файлов (не путайте их с безымянными конвейерами, которые часто применяются в командных оболочках) используются для коммуникации между процессами. Вам почти никогда не придется работать с этими файлами, хотя вы будете часто встречаться с их безымянными сородичами в главе 11 «Конвейеры и перенаправление ввода-вывода». В перечне файлов и каталогов именованный канал обозначается буквой `p`[3]:

```
prw-r--r-- 1 user user 0 Jan 21 10:00 mynamedpipe
```

- **Ссылки.** Ссылки — это своего рода «ярлыки», которые ведут от одних файлов к другим. Ссылки бывают двух видов — *жесткие* и *символические* («мягкие»). С жесткими ссылками вы вряд ли будете иметь дело, однако символические ссылки могут пригодиться, чтобы настроить удобные пути к часто используемым файлам или предусмотреть, чтобы к одному и тому же файлу вели разные пути. Чуть позже мы поговорим об этом подробнее. Символическая ссылка может выглядеть так (буква `l`[4] в начале строки):

```
lrwxrwxrwx 1 user user 7 Jan 21 10:30 versions/latest -> bin/app-3.1
```

  В этом примере показана ссылка на файл с именем `app-3.1`, которая называется `latest`.

- **Сокеты.** Подобно именованным каналам, сокеты в Unix используются для коммуникации между процессами. Файлы сокетов могут встретиться, если вы устраняете неполадки служб, которые должны обмениваться данными друг с другом. («Почему nginx не может достучаться до моего сервера приложения?») Вот как может выглядеть файл сокета, через который коммуницируют nginx и PHP-FPM, чтобы функционировало приложение WordPress (сокет обозначается буквой `s`[5] в начале):

```
srwxrwx--- 1 root socket 0 Jan 23 11:31 /run/wordpress.sock
```

---

[1] От *character* (*символ*). — *Примеч. пер.*

[2] В английском языке именованные конвейеры часто обозначаются аббревиатурой *FIFO* (*first in, first out*, то есть *первый вошел — первый вышел*) в соответствии с принципом их работы. — *Примеч. науч. ред.*

[3] От *pipe* (здесь: *канал*, *конвейер*). — *Примеч. пер.*

[4] От *link* (*ссылка*). — *Примеч. пер.*

[5] От *socket* (*сокет*). — *Примеч. пер.*

Мы перечислили несколько дополнительных, особых типов файлов, с которыми вы можете столкнуться, и постарались дать представление о том, когда (и почему) их можно встретить в реальных системах. Чтобы овладеть полезными практическими навыками, вам стоит особенно тщательно разобраться с отдельными типами файлов, которые мы сейчас рассмотрим, — речь идет прежде всего о ссылках.

Символические ссылки

Символические ссылки, которые еще называются «мягкими» ссылками или «симлинками», — это тип файла, который указывает на другой файл или каталог. В отличие от жесткой ссылки символическая ссылка может вести на ресурсы в других файловых системах и поддерживает собственный индексный дескриптор[1], который не совпадает с дескриптором целевого файла или каталога.

Чтобы создать символическую ссылку, используйте команду `ln`[2]:

```
ln -s существующий_файл символическая_ссылка
```

Например, если в текущем каталоге есть файл `file1.txt` и вы хотите создать символическую ссылку на этот файл, которая называется `link1`, запустите такую команду:

```
ln -s file1.txt link1
```

Теперь, если вывести содержимое каталога в расширенном формате (`ls -l`), вы увидите, что файл `link1` обозначен как символическая ссылка на файл `file1.txt`:

```
ls -l
total 0
-rw-r--r-- 1 root root 0 Oct 28 16:08 file1.txt
lrwxrwxrwx 1 root root 9 Oct 29 17:20 link1 -> file1.txt
```

Когда вы обращаетесь к `link1`, например, если хотите вывести на экран содержимое файла командой `cat link1`[3], — система автоматически разыменует ссылку и выведет содержимое файла `file1.txt`. Если переместить, переименовать или удалить этот целевой файл, то символическая ссылка не обновится сама по себе и будет указывать на несуществующий файл (так называемая «битая ссылка»).

---

[1] Индексный дескриптор (inode) — специальная структура данных, которая содержит метаданные о файле или каталоге, в том числе тип файла, сведения о владельце и время последнего доступа. — *Примеч. науч. ред.*

[2] От *link* (*ссылка*). Параметр `-s` (от *symbolic* — *символический*) позволяет создать именно символическую ссылку (а не жесткую). — *Примеч. пер.*

[3] Команда `cat` (от *concatenate* — *сцеплять*) выводит содержимое одного или нескольких файлов в стандартный вывод. — *Примеч. науч. ред.*

Символические ссылки особенно полезны для того, чтобы создавать ярлыки, структурировать файлы и каталоги, а также поддерживать гибкую и логичную организацию файловой системы.

Жесткие ссылки

Жесткая ссылка — это другое название для файла, который находится в той же файловой системе, что и исходный файл, и фактически ведет себя как его псевдоним. У исходного файла и жесткой ссылки один и тот же индексный дескриптор, благодаря чему изменения, которые вносятся в один из этих файлов, автоматически отражаются на другом. Жесткие ссылки, в отличие от символических, не могут вести на ресурсы в другой файловой системе или на каталоги. Если удалить исходный файл, то жесткая ссылка будет по-прежнему хранить те же данные. Чтобы создать жесткую ссылку на файл `file1.txt`, которая называется `link1`, запустите такую команду:

```
ln file1.txt link1
```

Команда file

Утилита `file` позволяет исследовать тип файла. Использовать ее проще простого: введите в командную строку `file`, а затем имя файла, например:

```
file mysecret.txt
```

Эта команда может вывести: `mysecret.txt: ASCII text`, то есть `mysecret.txt` — файл в формате обычного текста.

Если вы изучаете двоичный файл, например откомпилированную программу `mybinary`[1], то команда `file mybinary` выведет что-то вроде `mybinary: ELF 64-bit LSB executable`, то есть «`mybinary` — двоичный исполняемый файл».

Если передать команде `file` каталог, например `file /home/user`, то она, скорее всего, выведет `/home/user: directory`, то есть «`/home/user` — это каталог».

Команда `file` — мощный инструмент для того, чтобы быстро узнать тип файла, с которым вы работаете. Это особенно полезно для незнакомых или подозрительных файлов.

Если вас тянет на эксперименты, попробуйте с помощью `file` изучить такие файлы:

- `/bin/sh`
- `/dev/zero`

[1] *Мой двоичный файл. — Примеч. пер.*

- `/dev/urandom`
- `/dev/sda1`
- `~/.bashrc`
- `/bin/ls`
- `/home`
- `/proc/1/cwd`

Продвинутые операции с файлами

Когда вы работаете с файлами в операционных системах семейства Unix, вам часто требуется выполнять те или иные операции над файлами или их содержимым, но при этом не редактировать их непосредственно в редакторе. Например, вам может понадобиться:

- узнать, есть ли в файле то или иное содержимое;
- выяснить, какие файлы изменялись в определенное время;
- безопасно переместить файл в другую файловую систему, а не просто переложить его с места на место на локальном компьютере с помощью `mv`.

Порой даже требуется совместить все три задачи в одной операции! Знание о том, как это сделать, пригодится и при устранении неполадок (чтобы найти в журнале определенный тикет или код ошибки), и в разработке (чтобы узнать, какие файлы с исходным кодом недавно менялись), и если вы решили заняться тестированием (чтобы скопировать обновленный исходный код приложения в тестовую среду).

Предлагаем вашему вниманию краткий обзор этих файловых операций и соответствующих инструментов и команд.

Поиск по содержимому файлов с помощью grep

Чтобы находить совпадения с шаблоном в тексте, традиционно используется команда `grep`[1]. Возможно, на своем домашнем или рабочем компьютере вы предпочтете установить `ag` или `rg`[2], которые более дружественны к программистам

---

[1] Название команды `grep` происходит от команды `g/re/p`, которая выполняла ту же функцию в раннем текстовом редакторе `ed`. В свою очередь, эта команда расшифровывается как *global / regular expression search / print*, то есть «глобально искать совпадения с регулярным выражением и вывести строки с совпадениями». — *Примеч. пер.*

[2] Название команды `ag` — химическое обозначение серебра (*Ag*); ее автор Джефф Грир таким образом намекнул на метафору «серебряной пули» и одновременно подшутил над названием конкурирующей команды `ack`. Название команды `rg` — сокращение от *rip grep*, то есть «покойся с миром, `grep`». — *Примеч. науч. ред.*

и работают быстрее (например, `ag` устанавливается командой `sudo apt apt-get install silversearcher-ag`). Но в продакшен-средах обычно недоступно ничего, кроме `grep`.

Вот как можно искать совпадения с шаблоном *`шаблон`* в файле *`путь/к/файлу`*:

```
grep "шаблон"[1] путь/к/файлу
```

Конечно, с помощью `grep` можно искать совпадения со строковым литералом наподобие этого, однако прославленная мощь `grep` связана с тем, что она позволяет задавать шаблоны в виде регулярных выражений. Например, такая команда выведет строки, которые начинаются с буквы `A`:

```
grep ^A путь/к/файлу
```

А такая команда — строки, которые оканчиваются на `Я`:

```
grep Я$ путь/к/файлу
```

Регулярные выражения невероятно полезны, и каждому разработчику и пользователю Linux стоит освоить их хотя бы на начальном уровне.

С помощью `grep` с флагом `-r`[2] можно рекурсивно искать по заданному каталогу, то есть по всем файлам внутри этого каталога и всех его подкаталогов[3]:

```
root@c7f1417df8d2:/tmp# grep -r -i "hello world" /tmp
/tmp/secret/dontlook.key:hello world
/tmp/hi.txt:hello world
/tmp/hi.txt:HeLlO WoRlD! Как ты меня нашел?
```

Но что, если вам нужно найти не строки внутри файла, а сами файлы, которые соответствуют определенному условию?

Как искать файлы с помощью find

Команда `find` умеет искать файлы и каталоги по имени, по времени последнего изменения или другим атрибутам. По сути, она выполняет поиск в ширину по дереву файловой системы, что весьма полезно для задач вроде таких:

---

[1] Кавычки вокруг шаблона в простых случаях не обязательны; они полезны, когда шаблон содержит пробелы и/или специальные символы. — *Примеч. науч. ред.*

[2] Как и во многих других командах, флаг `-r` имеет длинную форму `--recursive` и применяет команду рекурсивно к заданному каталогу и всем его подкаталогам. — *Примеч. науч. ред.*

[3] Флаг `-i` в этом примере означает поиск без учета регистра (его длинная форма — `--ignore-case`, то есть *игнорировать регистр*). Поэтому команда обнаружила строки, в которые входит не только `hello world`, но и `HeLlO WoRlD`. — *Примеч. науч. ред.*

- найти все файлы журналов приложений, которые были созданы или изменены за последние сутки;
- обнаружить все тестовые файлы с исходным кодом, имена которых оканчиваются на `_test.go`;
- найти все файлы `php.ini`, которые остались после стажера-программиста, чтобы удалить их.

В последующих примерах */путь/поиска* означает ветвь файловой системы, в которой вы ищете нужные файлы. Если вас интересует поиск в текущем каталоге и всех его подкаталогах, можно использовать точку (`.`), например: `find . -name 'file.txt'`.

- Найти файлы, имена которых оканчиваются на `.ext`:

```
find /путь/поиска -name '*.ext'
```

- Найти файлы, которые соответствуют нескольким шаблонам для пути и для имени:

```
find /путь/поиска -path '**/путь/**/*.ext' -or -name '*шаблон*'
```

- Найти каталоги, имена которых соответствуют заданному шаблону без учета регистра:

```
find /путь/поиска -type d -iname '*lib*'
```

- Найти файлы, имена которых соответствуют указанному шаблону, исключив из поиска определенные каталоги:

```
find /путь/поиска -name '*.py' -not -path '*/site-packages/*'
```

- Найти файлы, размер которых находится в заданном диапазоне:

```
find /путь/поиска -size +500k -size -10M
```

Как копировать файлы между локальным и удаленным узлом с помощью rsync

`rsync`[1] — это чрезвычайно полезная утилита, которая позволяет копировать файлы и каталоги в пределах одного узла и между узлами. Она похожа на `cp`, но работает и тогда, когда один или оба узла являются удаленными.

По сути, `rsync` совмещает в себе `cp` (чтобы копировать данные) и `ssh` (чтобы безопасно передавать зашифрованные данные). Если вы плохо знакомы с командой `ssh`, постарайтесь изучить, как она работает, а также настроить свои собственные ключи и конфигурацию доступа, прежде чем запускать `rsync`.

---

[1] Сокр. *remote sync* (*удаленная синхронизация*). — *Примеч. пер.*

Вот несколько характерных примеров вызова `rsync`, которые опираются на материалы проекта tldr (github.com/tldr-pages/tldr):

- Передать файл с локального узла на удаленный:

```
rsync путь/к/локальному_файлу удаленный_узел:путь/к/удаленному_каталогу
```

- Передать файл с удаленного узла на локальный:

```
rsync удаленный_узел:путь/к/удаленному_файлу путь/к/локальному_каталогу
```

- Передать файл в архивном режиме (`-a` — чтобы сохранить атрибуты) и в сжатом виде (`-z`), отображать ход выполнения (`-P`) в подробном (`-v`) и человекочитаемом (`-h`) формате:

```
rsync -azvhP путь/к/локальному_каталогу удаленный_узел:путь/к/удаленному_
каталогу
```

Последний пример мы сами использовали сотни раз, чтобы быстро и в автоматическом режиме создавать резервные копии файлов.

Как совмещать find, grep и rsync

О том, как связывать команды с помощью вертикальной черты (`|`), мы подробно поговорим в главе 11 «Конвейеры и перенаправление ввода-вывода», но сейчас представим краткий обзор.

Допустим, мы хотим объединить примеры, которые только что рассмотрели, чтобы сделать резервные копии всех файлов из каталога `/tmp`, которые содержат строку `hello world` и были изменены за последнюю неделю. Это можно сделать одной элегантной командой:

```
find /tmp -type f -mtime -7 -exec grep -l "hello world" {} \; | xargs -I _
backupscript.sh _ backup@backupserver.local:/backups_
```

В этом примере мы сначала запускаем `find`, чтобы найти файлы, которые изменялись не более 7 дней назад. Флаг `-exec` команды `find` нужен для того, чтобы выполнить команду `grep` с флагом `-l`, которая просто возвращает имя файла, где обнаружено совпадение. Затем мы передаем имена файлов по конвейеру команде `xargs`, которая применяет указанную операцию к каждой строке ввода, полученного от предыдущей команды. В данном случае операция заключается в том, чтобы запустить сценарий резервного копирования `backupscript.sh` для каждого из подходящих файлов, а также передать этому сценарию целевой путь, по которому нужно разместить резервные копии (он начинается с `backup@backupserver.local:/backups`).

Если в нашем распоряжении есть те же файлы, что в недавнем разделе «Поиск по содержимому файлов с помощью `grep`», то вся эта сложносочиненная команда запустит две такие команды:

```
backupscript.sh /tmp/secret/dontlook.key backup@backupserver.local:/backups/
tmp/secret/dontlook.key

backupscript.sh /tmp/hi.txt backup@backupserver.local:/backups/tmp/hi.txt
```

Они делают именно то, чего мы добивались: запускают сценарий резервного копирования *только* для тех двух файлов, которые содержат искомый текст `hello world` *и* были изменены за последние 7 дней.

Хотя для того, чтобы составлять подобные команды, вам придется потратить несколько минут времени (и воспользоваться помощью поисковых систем), в долгосрочной перспективе это может сэкономить многие часы. В этом и заключается мощь командной строки, дополненная небольшими специализированными инструментами Unix, которые можно сочетать друг с другом так, как вам нужно.

В главе 11 «Конвейеры и перенаправление ввода-вывода» вы узнаете гораздо больше о конвейерах в Unix и команде `xargs`. Сейчас мы продемонстрировали этот пример, чтобы дать представление о том, каких фантастических результатов можно добиться, если объединять друг с другом простые команды, с которыми вы уже познакомились.

Расширенные возможности файловой системы в реальной практике

К этому моменту вы уже познакомились с разными типами файлов в Linux и на практике поработали с самыми распространенными из них. Давайте теперь уделим внимание более специализированным темам, в которых полезно ориентироваться, чтобы эффективно обращаться с файловой системой в Linux.

Эти темы пригодятся, когда вы будете:

- устранять неполадки в своем первом приложении Docker, с которым смонтированы тома накопителя;
- разрабатывать приложение, которое обменивается данными с промышленными контроллерами, системами видеонаблюдения и другим внешним оборудованием;
- писать код, которому нужно получать случайные числа, чтобы генерировать безопасные пароли или маркеры API.

Один из специальных типов файлов, с которыми вы встретитесь, — **блочные устройства**, то есть оборудование, которое в той или иной степени похоже на диск, где данные читаются и записываются блоками.

Традиционные дисковые накопители — это блочные устройства, и в файловой системе им обычно соответствуют такие пути:

- `/dev/hdX`
- `/dev/sdX`
- `/dev/nvmeN`

Здесь `X` и `N` — это буквенные или цифровые индексы накопителей, полный путь к которым может выглядеть как `/dev/sda` или `/dev/nvme0`[1]. **Разделы** файловой системы выглядят так же, как диски, но с добавочной буквой или цифрой: например, первый раздел первого диска может обозначаться как `/dev/sda0`.

Обратите внимание, что даже когда операционная система распознаёт новый накопитель и подключает его в одно из этих расположений (вместе с любыми обнаруженными разделами), вам все равно понадобится специально монтировать соответствующую файловую систему с помощью команды `mount`. Это не самая насущная операция для разработчиков, поэтому подробности мы опустим.

Бывают также специальные **программные устройства**, которые не соответствуют никакому физическому оборудованию, например:

- `/dev/null`, куда часто перенаправляется вывод в командах наподобие *команда* `> dev/null`;
- `/dev/random` и `/dev/urandom`, которые порождают случайные наборы байтов. Скорее всего, именно из этих устройств ваш любимый язык программирования получает случайные числа для безопасных криптографических операций.

Еще один важный каталог — `/proc`, который был популяризирован ОС Plan 9, хотя предусматривался еще на заре Unix. Как можно догадаться по названию, этот каталог представляет процессы в виде файлов: в нем содержатся каталоги, имена которых совпадают с идентификаторами процессов (PID), а файлы в каждом из этих каталогов отражают состояние соответствующего процесса. В Linux каталог `/proc` оснащен дополнительными интерфейсами, которые, в частности, позволяют конфигурировать драйверы ядра, получать сведения об оборудовании и сигналы датчиков и даже взаимодействовать с BIOS и UEFI.

---

[1] Аббревиатура *sd* исторически обозначала устройства SCSI, но теперь в `/dev/sdX` размещается большинство накопителей, включая контроллеры IDE и SATA. В `/dev/hdX` (*hd* — от *hard drive* или *hard disk*, то есть *жесткий диск*) раньше находились традиционные жесткие диски, подключенные по интерфейсу IDE, и до сих пор эти разделы файловой системы иногда используются для устаревших устройств. В `/dev/nvmeN` находятся контроллеры устройств, подключенных по более современному интерфейсу NVMe. — *Примеч. науч. ред.*

FUSE: новые горизонты файловых систем Unix

Как вы уже убедились, многие объекты в Linux можно интерпретировать как файлы. В основе такого подхода лежит идея о том, что редактировать файлы — очень распространенная задача, поэтому команды и языки программирования, которые умеют оперировать файлами, тем самым обеспечивают знакомый и понятный интерфейс. FUSE[1] — это интерфейс прикладного программирования (API), который позволяет любому разработчику реализовывать новые файловые системы Unix, не углубляясь в программирование на уровне ядра. Другими словами, поскольку многие программные объекты умеют взаимодействовать с файлами, иногда полезно имитировать API файловой системы Unix для сущностей, которые не относятся к «обычным» файлам, то есть байтам, хранящимся на локальном диске и традиционно называемым файлами. Если это звучит немного странно, присмотритесь к некоторым решениям, которые написаны на FUSE. Например, с помощью этого API реализованы многие драйверы классических файловых систем, в частности, NTFS, благодаря чему на компьютере под управлением Linux можно работать с файловыми системами Windows. Впрочем, FUSE оказался достаточно гибким и доступным для того, чтобы реализовать на нем и более экзотические файловые системы.

- `sshfs` позволяет смонтировать на локальном компьютере каталог, который находится на другом узле и доступен по SSH.
- Существуют файловые системы FUSE, с помощью которых можно смонтировать удаленное облачное хранилище (например, Amazon S3) как локальный каталог.
- Некоторые еще более необычные решения позволяют смонтировать «Википедию» как каталог с файлами или представить в виде файловых систем такие протоколы, как IRC, и такие ресурсы, как API метеорологических служб.

Технология FUSE проявила себя настолько удачно, что ее внедрили многие операционные системы семейства Unix помимо Linux и она доступна даже в Windows. Про FUSE стоит знать не только потому, что это инновационное воплощение абстракции файлов в Unix, но и потому, что этот API может оказаться чрезвычайно полезным, когда вы имеете дело с данными, которые хранятся в таких расположениях, куда нельзя обратиться через классический API на уровне приложения. В любом популярном языке программирования есть стандартная библиотека, которая позволяет взаимодействовать с файлами в Unix, и с помощью FUSE можно наладить соответствующий интерфейс для любых источников данных.

---

[1] Сокращение от *Filesystem in Userspace* (*файловая система в пользовательском пространстве*). — *Примеч. пер.*

Итоги

В этой главе мы познакомились с основными понятиями, а также с некоторыми продвинутыми возможностями файлов и файловой системы в Linux. Вы узнали, чем отличаются текстовые файлы от двоичных, увидели, как устроено дерево файловой системы Linux, и изучили все основные команды, которые нужны, чтобы работать с файлами. Если вы добросовестно осваивали материал, то наверняка уделили время тому, чтобы поработать в своей среде Linux и закрепить навыки редактирования файлов в командной строке.

От элементарных понятий мы перешли к наиболее важным продвинутым темам, которые вам пригодятся. Вы узнали, как искать файлы и содержимое в них, а также получили представление о специальных типах файлов.

Все это вместе должно вооружить вас важнейшими знаниями и навыками, которые помогут использовать Linux, чтобы решать задачи из реальной практики. Надеемся, что вам понравился этот головокружительный обзор!

6

Как редактировать файлы из командной строки

Разработчикам часто требуется редактировать файлы из командной строки: ведь в средах реальной эксплуатации, как правило, нет графического интерфейса. Однако если вы умеете эффективно редактировать текстовые файлы в командной оболочке, это дает немало преимуществ не только в терминале, но даже там, где в вашем распоряжении есть графические текстовые редакторы или IDE.

Например, многие полнофункциональные текстовые редакторы и IDE поддерживают приемы, о которых вы узнаете в этой главе, а значит, вы сможете перенести навыки быстрой и эффективной работы на другие инструменты. Сочетания клавиш, которые вы здесь изучите, применимы в самых разных ситуациях: например, когда нужно срочно найти и заменить текст в одном или нескольких файлах или исправить опечатку посередине длинной команды оболочки.

Вы наверняка обнаружите, что похожие сочетания клавиш интегрированы в ваши любимые инструменты — иногда с помощью дополнений. Например, поисковая система охотно подскажет вам множество клиентов электронной почты, браузерных надстроек и веб-приложений с поддержкой сочетаний клавиш Vim, которые вы изучите в этой главе.

Если вы привыкнете к эффективным методам работы, которые свойственны минималистичным, чисто текстовым интерфейсам, это поможет вам также более продуктивно выполнять повседневные операции и не тратить время на то, чтобы постоянно щелкать по графическим меню или выбирать варианты в пошаговом мастере. Вместо этого вы будете достигать нужных результатов всего несколькими нажатиями клавиш.

ПРИМЕЧАНИЕ

Из эксплуатационных образов чрезвычайно минималистичных (или чрезвычайно защищенных) сред часто исключают текстовые редакторы — хотя это не повышает безопасность, потому что если понадобится, из команд `cat`, `echo`, `mv` и перенаправления ввода-вывода можно соорудить работоспособный редактор. Скорее всего, как любые компьютерные специалисты, вы за свою карьеру установите немало экземпляров nano или Vim в контейнерах Docker.

В этой главе вы изучите основы работы с двумя текстовыми редакторами. Один из них — nano, его, по нашему мнению, легче всего освоить для начала, а другой — Vim, владеть которым, как нам кажется, особенно полезно для профессионального успеха в долгосрочной перспективе. Вы получите общее представление о том, как работают текстовые редакторы в командной оболочке Linux, детально познакомитесь с nano и Vim и, наконец, научитесь избегать самых частых ошибок при редактировании файлов. Также мы покажем, как настроить оболочку, чтобы она автоматически открывала файлы в выбранном вами текстовом редакторе, когда это возможно.

Nano

Nano[1] — это облегченный и простой в использовании текстовый редактор с интерфейсом командной строки. Возможно, самая заметная его особенность состоит в том, что все время, пока вы редактируете текст в терминале, в нижней части экрана перед вами отображается шпаргалка по сочетаниям клавиш. Она особенно полезна, если вы пока еще чувствуете себя неуверенно и не привыкли редактировать текст из командной строки.

Nano в целом хорош, хотя вы вряд ли встретите его в минималистичных окружениях, например в контейнерах Docker или виртуальных машинах в среде реальной эксплуатации. Также учтите, что nano часто норовит автоматически создавать резервные копии файлов (`~имя_файла.txt`) и может засорить ими файловую систему.

[1] Название *nano* — *нано-*, приставка системы СИ (10^{-9}). Это очередной каламбур разработчиков: nano создавался как замена редактору Pico, а приставка *пико-* означает 10^{-12}, то есть в 1000 раз меньше, чем *нано-*. — *Примеч. науч. ред.*

Как установить nano

Во всех популярных дистрибутивах Linux, с которыми вам предстоит работать, пакет текстового редактора nano называется nano. Чтобы его установить, используйте стандартный диспетчер пакетов своей операционной системы. Например, вот как установить nano в Ubuntu:

```
apt-get install nano
```

Шпаргалка по сочетаниям клавиш nano

Официальная и наиболее актуальная шпаргалка по сочетаниям клавиш nano находится по адресу nano-editor.org/dist/latest/cheatsheet.html. Далее в этом разделе перечислены некоторые из самых востребованных сочетаний клавиш.

Операции с файлами

Ctrl+S	Сохранить текущий файл
Ctrl+O	Запрос на запись файла («Сохранить как»)
Ctrl+R	Вставить файл в текущий файл
Ctrl+X	Закрыть буфер, выйти из nano

Редактирование

Ctrl+K	Вырезать текущую строку в буфер обмена
Ctrl+U	Вставить содержимое буфера обмена
Alt+3	Закомментировать или раскомментировать строку или блок
Alt+U	Отменить последнее действие
Alt+E	Повторить последнее отмененное действие

Поиск и замена

Ctrl+Q	Начать обратный поиск
Ctrl+W	Начать прямой поиск
Alt+Q	Найти ближайшее следующее вхождение
Alt+W	Найти ближайшее предыдущее вхождение
Alt+R	Начать сеанс замены

vi и Vim

vi — это еще один текстовый редактор с интерфейсом командной строки; его часто можно встретить под именами ex-vi или nvi[1]. Vim (vi iMproved[2]) — расширенная версия vi, которую многие используют как полноценную IDE. vi и Vim поддерживают один и тот же набор основных команд и сочетаний клавиш, поэтому если вы его освоите, то сможете благополучно работать и в старинных, и в современных системах.

Мы должны предупредить, что Vim отличается непростым интерфейсом, и потребуются определенные усилия, чтобы его изучить. Возможно, чтобы начать уверенно обращаться с Vim, вам придется практиковаться в свободное время в течение одного-полутора месяцев. Это можно сравнить с трудозатратами на то, чтобы настроить свой первый веб-сервер под управлением Linux или написать первую программу на 500 строк.

Важное преимущество Vim состоит в том, что на удаленном сервере, где нет графического интерфейса, с помощью этой программы вы сможете редактировать файлы точно так же, как на локальном компьютере, причем и там и там вы будете работать весьма эффективно. Правда, для этого важно в определенной степени перестроить свое мышление, а также освоить ключевые понятия Vim — *команды* и *режимы*, о которых мы вскоре поговорим.

Как и всегда, когда вы овладеваете новым навыком, вам понадобится специально практиковаться в работе с Vim, а затем регулярно использовать его, иначе не получится по-настоящему освоить редактор и уверенно с ним обращаться. Возможно, поначалу вы столкнетесь с некоторыми затруднениями и не во всем сразу разберетесь; мы очень надеемся, что это вас не остановит!

Vim — модальный («режимный») редактор: это означает, что одни и те же клавиши выполняют разные функции в зависимости от того, в каком режиме вы находитесь. Например, если вы работаете в режиме вставки, то нажатия клавиш будут просто записываться в файл (или буфер), который вы редактируете, подобно тому, как это происходит в вашей IDE или в Microsoft Word. Однако в режиме по умолчанию (так называемом нормальном режиме) те же самые клавиши будут вместо этого выполнять команды, которые им назначены. Как только вы

---

[1] ex-vi — классический редактор vi, перенесенный на современные системы Unix с минимальными дополнениями (вроде поддержки Unicode). nvi (*new vi*, то есть *новый vi*) — современная версия vi для систем семейства BSD с обновленными функциями (помимо Unicode, nvi поддерживает расширенные регулярные выражения, горизонтальную прокрутку и некоторые другие возможности). — *Примеч. науч. ред.*

[2] *Улучшенный vi*. — *Примеч. пер.*

привыкнете к этому принципу **модального редактирования**, вам останется лишь немного попрактиковаться, чтобы освоить vi или Vim.

Например, если вы запустите Vim и дважды нажмете строчную i, то первая i переключит редактор в режим вставки, а вторая i введет букву *i* в активное окно (буфер), где вы редактируете файл. Если это пока не очень понятно — не волнуйтесь. Даже если Vim никогда не станет вашей рабочей IDE, к концу этой главы вы будете гораздо лучше ориентироваться в том, как им пользоваться.

ПРИМЕЧАНИЕ

Стоит упомянуть, что во время написания этой книги стал набирать популярность еще один редактор, подобный Vim,— он называется Nvim (Neovim[1]). Бóльшая часть всего, что вы узнаете о Vim, также относится к Nvim, поэтому неважно, с каким из двух упомянутых редакторов вы начнете работать. Основные различия между ними касаются разработки дополнений, так что вы ничего не потеряете, если впоследствии перейдете с Vim на Neovim, как сделали мы.

Команды vi и Vim

Далее перечислены некоторые элементарные команды vi и Vim. Перед тем как использовать любую из них, нажмите Esc, чтобы убедиться, что вы находитесь в нормальном режиме.

Режимы

v	Переключиться в визуальный режим. Этот режим есть только в Vim (и отсутствует в vi), и начинающие пользователи часто им злоупотребляют, потому что он наиболее привычен по опыту работы с другими редакторами
Esc	Выйти из любого режима, в котором вы находитесь, и переключиться в нормальный режим, где можно вводить команды

Командный режим

В командный режим можно переключиться из нормального (Esc), если нажать двоеточие :. Для ясности мы включили этот символ в команды, которые перечисляются ниже.

[1] *Новый Vim. — Примеч. пер.*

Подсказки[1]

`:set number`	Отображать номера строк
`:set paste`	Это может пригодиться, если вы хотите сохранить отступы, когда вставляете текст или код в Vim. Чтобы отключить эту настройку, введите команду `:set nopaste`

Выход из редактора[2]

`:q`	Выйти
`:q!`	Выйти без сохранения (принудительный выход)
`:w`	Сохранить файл
`:wq`	Сохранить файл и выйти
`:wqa`	Сохранить все открытые файлы и выйти (работает только в Vim). Это полезно, когда открыто несколько панелей: например, если надстройка открывает диспетчер файлов на боковой панели

Нормальный режим

Нормальный режим — это режим по умолчанию, в котором Vim начинает работать до того, как вы что-либо ввели. Вернуться в нормальный режим можно в любой момент, если нажать Esc.

Перемещение по документу[3]

k	Вверх
j	Вниз
l	Вправо
h	Влево

[1] *Number* — здесь *номер* (строки), *paste* — *вставить*. — *Примеч. пер.*

[2] Многие сочетания клавиш Vim происходят от английских слов, которые обозначают соответствующие операции, и это помогает запомнить эти сочетания тем, кто владеет языком. Здесь `q` — от *quit* (*выйти*), `w` — от *write* (*записать*), `a` — от *all* (*все*). В терминологии vi и Vim операция сохранения файла обозначается словом *write*, а не *save* (*сохранить*), как в большинстве современных редакторов. — *Примеч. науч. ред.*

[3] В отличие от многих сочетаний клавиш vi, эти четыре клавиши выбраны не потому, что с них начинаются определенные английские слова, а потому, что они находятся под пальцами правой руки в центральном ряду клавиатуры. — *Примеч. науч. ред.*

Перемещаться по документу также можно с помощью клавиш со стрелками, но лучше придерживаться традиционных сочетаний клавиш vi, чем пытаться использовать vi или Vim как обычный редактор. Мы убедились, что, если приспособиться к сочетаниям клавиш vi[1], то в долгосрочной перспективе удается работать эффективнее:

w	К следующему слову[2]
b	К началу текущего слова
^	К первому значащему символу в строке
0	(цифра 0) В начало строки
$	В конец строки
g+g	В начало документа
G	В конец документа

Редактирование[3]

i	Переключиться в режим вставки перед текущей позицией курсора (чтобы непосредственно набирать текст). I — вставка в начало строки
a	Переключиться в режим вставки после текущей позиции курсора. A — вставка в конец строки
o	Вставить новую строку после текущей. O — вставить новую строку перед текущей
/	Поиск совпадений с шаблоном. Поддерживаются регулярные варажения. С помощью Enter можно запустить поиск, а с помощью n и Shift+n — перемещаться вперед или назад по результатам поиска

[1] Клавиатурный интерфейс vi, непривычный современным пользователям, объясняется историческими причинами. В 1970-е годы разработчики подобных редакторов стремились, чтобы они были совместимы с тогдашними терминалами, где не всегда были даже клавиши для символов ^ или ;, не говоря уже о стрелках. — *Примеч. науч. ред.*

[2] В этой таблице: w — от *word* (*слово*), b — от *begin* или *beginning* (*начало*), g — от *go* (*перейти*). Символы ^ и $ по историческим причинам обозначают соответственно начало и конец строки во многих продуктах и языках, в том числе в регулярных выражениях. — *Примеч. науч. ред.*

[3] В этой таблице: i — от *insert* (*вставить*), a — от *append* (*добавить*, *дополнить*), o — от *open* (*открыть*), d — от *delete* (*удалить*), y — от *yank* (*выдернуть*), p — от *put* (*положить*, *поместить*) или *paste* (*вставить*), u — от *undo* (*отменить действие*), r — от *redo* (*снова выполнить действие*). — *Примеч. науч. ред.*

d+d	Вырезать текущую строку (и поместить ее в буфер обмена)
y	Скопировать выбранный текст
y+y	Скопировать текущую строку
p	Вставить текст после курсора
u	Отменить последнее изменение
Ctrl+r	Вернуть последнее отмененное изменение
n+X	Выполнить *n* раз команду *X*. Например, 3+d+d удаляет три строки

Как эффективно освоить vi или Vim

Скорее всего, вы уверенно освоите Vim за полтора-два месяца работы над реальными задачами. Это одна из иллюстраций к тому, что обучаться новому продукту не всегда легко. Вот несколько советов о том, как облегчить себе знакомство с Vim.

Пользуйтесь встроенным обучающим пособием

В Vim есть встроенное обучающее пособие. Возможно, с него стоит начать, если вы впервые запускаете этот редактор. Введите `vimtutor` в командной строке, чтобы открыть одновременно сам Vim и пособие.

Применяйте мнемонические правила

Многие пользователи vi или Vim предпочитают составлять фразы из команд, которые мы перечислили: например, `d2w` означает «удалить два слова» (*delete 2 words*). Разным людям могут быть удобнее разные ментальные модели, так что не стесняйтесь придумывать собственные мнемонические правила.

Не используйте клавиши со стрелками

Постарайтесь не использовать клавиши со стрелками или вовсе отключите их поддержку. Благодаря этому вы не будете относиться к Vim как к еще одному обычному редактору и потратите меньше времени на то, чтобы привыкнуть к его стандартным сочетаниям клавиш. Не волнуйтесь: хотя поначалу они кажутся экзотичными, после некоторой практики вы начнете использовать их на автомате.

Не используйте мышь

Хотя в Vim можно пользоваться мышью (например, для визуального выделения), лучше преодолеть это искушение и закрепить в своей рабочей памяти сочетания клавиш. Иначе вы застрянете «между двумя мирами» и не сможете эффективно

работать, когда это понадобится, например, если в три часа ночи придется устранять неполадки на удаленном сервере, который не принимает ввод от мыши.

Не используйте gVim

Хотя gVim (графический интерфейс для Vim) иногда может пригодиться, вряд ли его специфические возможности принесут пользу там, где нет подходящего терминала. Преимущество vi и Vim в том, что они позволяют эффективно работать с текстом с помощью клавиатуры, когда недоступна графическая среда, — как на многих серверах Linux, где вы будете устранять неполадки после того, как прочтете эту книгу!

Первое время не злоупотребляйте собственной конфигурацией или дополнениями

Типичная ошибка неопытных пользователей Vim состоит в том, что они начинают работать с конфигурацией, которую настроил кто-то другой. Поначалу может показаться, что это хорошая идея, однако чересчур персонализированная настройка Vim нередко сбивает с толку, если вы еще только знакомитесь с основными понятиями. Чужая пользовательская конфигурация сама по себе не сделает вас продуктивнее, особенно в начале работы. Впрочем, когда вы будете увереннее ориентироваться в Vim, вы наверняка разработаете собственную конфигурацию.

Также не стоит активно подключать дополнения (плагины), особенно на начальном этапе. Бывает, что они нарушают работу тех или иных стандартных функций, а это доставляет лишние хлопоты и создает проблемы на пустом месте. Вряд ли вам доставит удовольствие устранять неполадки в дополнениях Vim, когда вы только начинаете его изучать. Сторонние конфигурационные файлы и дополнения порой приносят огромную пользу, но могут оказаться и в роли костыля: если вам внезапно придется работать в среде, которая существенно отличается от вашего компьютера, вы не сможете полагаться на все эти удобные надстройки. Если вы приучились к индивидуально отрегулированным рабочим процессам, то в стороннем окружении даже элементарные операции редактирования могут оказаться сложными и нервными, особенно в стрессовой ситуации.

Более конструктивный подход заключается в том, чтобы начать с минимальной конфигурации и добавлять только те настройки, которые вы полностью понимаете (и уверены, что они вам нужны). Работая в Vim, лучше всего уделять основное время реальным проектам, потому что это поможет вам закрепить наиболее важные сочетания клавиш в рабочей памяти. Это потребует некоторых усилий, но со временем вы начнете применять их так, чтобы не задумываться о конкретных командах, а ориентироваться только на мнемонические правила, которые вы придумали сами для себя.

Вот пример минимальной конфигурации Vim, которая может пригодиться на начальном этапе. Вам ничто не мешает изменять ее, как вы считаете нужным, или выбрать отсюда только те настройки, которые кажутся вам необходимыми.

Поместите этот код в файл `$HOME/.vimrc`:

```
" Эта настройка нарушает совместимость с vi,
" зато позволяет пользоваться преимуществами Vim
set nocompatible

" Включить синтаксическую подсветку
syntax on

" Увеличить историю команд до очень большого значения
set history=10000

" Настроить автоматический отступ по предыдущей строке
set autoindent

" Сделать так, чтобы операции поиска переходили к началу файла
" после его окончания
set wrapscan

" Отображать текущий режим в командной строке
set showmode

" Отображать частичные команды на последней строке
set showcmd

" Подсвечивать найденные совпадения
set hlsearch

" Выполнять поиск без учета регистра
set ignorecase
" Учитывать регистр, когда шаблон для поиска содержит прописные буквы
set smartcase

" Отображать позицию курсора в строке состояния
set ruler
```

Настройки Vim в других программах

Если вам понравится работать в Vim, то имейте в виду, что у многих текстовых редакторов и IDE есть настройки и дополнения для того, чтобы переключиться в режим ввода Vim. Бывают даже веб-браузеры, где поля ввода работают в стиле Vim!

Если вы хотите закрепить знания о редактировании файлов из командной строки, предлагаем вашему вниманию видеопособие «Vim Basics in 8 Minutes», где рассматриваются наиболее важные темы этой главы, а также некоторые дополнительные возможности Vim: youtu.be/ggSyF1SVFr4.

Как редактировать файл, если у вас недостаточно прав

Независимо от того, каким редактором вы пользуетесь, время от времени бывает нужно отредактировать файл, на запись в который у вашей учетной записи нет прав. Например, если вы — обычный пользователь, но хотите внести изменения в `/etc/hosts` — файл, который принадлежит `root` и доступен для записи только для `root`, — вам понадобится либо стать `root`'ом, либо вызвать команду `sudo`. Подробнее эта тема рассматривается в главе 7 «Пользователи и группы».

Чтобы редактировать файлы как `root`, можно запустить команду вида `sudo $EDITOR /etc/hosts`, но лучше использовать `sudoedit`[1]:

- `sudoedit /etc/hosts`
- `EDITOR=nano sudoedit /etc/hosts`
- `EDITOR=vi sudoedit /etc/hosts`

В первом из этих примеров запускается тот редактор, который настроен в переменной окружения `EDITOR`, а в двух других примерах значение этой переменной задается (или переопределяется) в составе команды.

Как настроить редактор по умолчанию

Не только в Linux, но и во всех системах семейства Unix можно настроить редактор по умолчанию с помощью переменной окружения `EDITOR`[2]. Большинство программ с интерфейсом командной строки, которым нужен редактор для тех или иных задач, — например, `git`, с помощью которого вы вносите коммит, или `visudo`, в котором вы редактируете файл `sudoers`, — используют значение этой переменной, чтобы узнать, какой редактор запускать. Переменной `EDITOR` можно присвоить путь к любому редактору на ваш вкус, даже с графическим интерфейсом (если ваша система его поддерживает):

```
bash-3.2$ echo $EDITOR
nano
bash-3.2$ export EDITOR=vim
```

Обратите внимание, что настройки из этого примера перестанут действовать после того, как сеанс интерактивной оболочки закроется. Чтобы сохранить значение

---

[1] Команда `sudoedit` эквивалентна `sudo` с флагом `-e` (`--edit`, *редактировать*). — *Примеч. науч. ред.*

[2] *РЕДАКТОР. — Примеч. пер.*

переменной в оболочке Bash, его следует добавить в файл `~/.bashrc`. (Подробнее см. главу 4 «История команд в оболочке Linux».)

Итоги

Из этой главы вы узнали, как редактировать текстовые файлы в командной строке. Сначала мы рассмотрели самый простой инструмент для этой задачи (nano), а затем заложили фундамент для профессиональных навыков, которые существенно помогут вам развить успешную карьеру. Вы познакомились с vi и Vim и их сочетаниями клавиш, которые поддерживаются в невероятно широком спектре приложений.

Чтобы начать редактировать файлы в командной строке, вам будет достаточно шпаргалок из этой главы, но имейте в виду, что после пары дней практики вы почувствуете, что готовы осваивать и другие сочетания клавиш и команды Vim. Для этого лучше всего органично совмещать встроенное пособие `vimtutor`, шпаргалки в интернете и видеоуроки на YouTube. Мы также рекомендуем книгу Дрю Нила (Drew Neil) «Practical Vim»[1].

Если вы уверенно редактируете текст в командной строке, это существенно подкрепляет ваш образ и репутацию настоящего профессионала. Не пренебрегайте этим навыком!

[1] Д. Нейл. «Практическое использование Vim».

7

Пользователи и группы

Эта глава посвящена двум основным механизмам, с помощью которых в Linux можно управлять ресурсами и обеспечивать безопасность: пользователям и группам. Сначала мы рассмотрим основы этих механизмов и узнаем, какую роль выполняет особый пользователь `root`, а затем продемонстрируем, как понятие групп в Linux добавляет удобный уровень абстракции поверх механизма пользователей.

После общего введения вы познакомитесь с командами, которые позволяют создавать и редактировать пользователей и группы. И в качестве «вишенки на торте» (которая может здóрово пригодиться на собеседовании!) вы узнаете, *что на самом деле представляет собой пользователь в Linux* (подсказка: это всего три строки обычного текста).

К концу этой главы вы будете знать:

- Что такое пользователи в Linux и зачем они нужны.
- Чем отличается `root` от обычных пользователей и как переключаться между обычным пользователем и суперпользователем, если это необходимо.
- Как создавать и редактировать пользователей и группы.
- Как устроены метаданные и что на самом деле представляет собой пользователь в Linux.

Что представляет собой пользователь

В системах семейства Unix пользователь — это просто именованная сущность, которая может выполнять те или иные операции в системе. Пользователи могут запускать процессы и владеть ими, владеть файлами и каталогами и иметь различные права доступа к ним. Пользователям может быть разрешено или запрещено выполнять определенные операции или обращаться к определенным ресур-

сам. С практической точки зрения пользователь — это объект, под чьим именем вы входите в систему, от чьего имени вы запускаете процессы и который владеет вашими файлами.

Очевидно, что слово «пользователь» (*user*) — это метафора живого человека, у которого есть учетная запись, пароль и т. д. Но в реальных системах большинство «пользователей» не ассоциированы с реальными людьми: это машинные учетные записи, которые помогают группировать ресурсы (например, процессы или файлы), чтобы обеспечить безопасность системы.

Однако главная разница между пользователями состоит не в том, предназначена ли учетная запись для того, чтобы под ней работал живой человек в интерактивном режиме. На самом деле в Unix существует ровно два типа пользователей. Давайте познакомимся с ними, прежде чем переходить к практическим навыкам, которые позволят вам управлять пользователями.

Root и остальные пользователи

Как известно, мир полон подвохов, и запускать команды не всегда безопасно. Например, команда `fdisk` может удалить раздел диска или еще как-то воздействовать на оборудование. Команда `iptables` может открыть сетевой порт, через который злоумышленник воспользуется уязвимостью в системе. Даже безобидная на первый взгляд команда `echo` способна вызвать неочевидные, но разрушительные повреждения, если направит значение в неположенное место файловой системы.

Для предотвращения этих неприятностей в системах семейства Unix имеются средства контроля за тем, чтобы ваши команды выполнялись в особом защищенном окружении. В каждой системе Unix есть суперпользователь, который называется `root`[1] (корневой пользователь). Итак, базовая модель безопасности такова:

- Прежде всего существует пользователь `root`, который аналогичен администратору в других системах и обладает наивысшим уровнем доступа в системе, то есть может сделать в ней практически что угодно.
- На следующей ступени существуют все остальные пользователи, чей уровень доступа ограничен: они не могут запускать процессы или редактировать файлы, от которых зависит работа всей системы, но могут запускать свои собственные непривилегированные приложения и редактировать свои собственные файлы.

---

[1] *Корень.* Это имя, как и многое другое, пришло в Unix из предшествующей системы Multics. Доподлинно неизвестно, почему суперпользователя назвали именно так, но среди специалистов популярна версия о том, что это связано с корневым (root) каталогом, который в Multics был домашним каталогом суперпользователя. Технически имя суперпользователя в Unix можно сменить, но делать это почти никогда не рекомендуется. — *Примеч. науч. ред.*

Из соображений безопасности только `root` может запускать команды, которые влияют на «несущие конструкции» системы. Иногда даже команды, которые выглядят вполне невинно, могут посеять хаос, если вызвать их с определенными аргументами. Поэтому время от времени вы будете сталкиваться с ситуацией, когда права `root` нужны даже для того, чтобы просто отредактировать текстовый файл.

sudo

Было бы неудобно входить в систему под учетной записью суперпользователя всякий раз, когда нужно сделать что-то потенциально опасное. К счастью, на этот случай есть команда `sudo`[1]. Если приписать `sudo` перед любой командой (через пробел), то она будет выполнена *от имени корневого пользователя*. После того как она завершится и вернет управление, последующие команды будут снова запускаться от имени обычного пользователя, под которым вы вошли в систему (не `root`).

Чтобы убедиться в этом, можно запустить две команды. Сначала запустите `whoami`[2] — команду, которая выводит имя текущего пользователя:

```
whoami
```

Мы вошли в систему как пользователь `dave`, поэтому в нашем случае команда выведет такой результат:

```
dave
```

А теперь запустите ту же команду, добавив перед ней `sudo`[3]:

```
sudo whoami
```

Хотя вы по-прежнему аутентифицированы в системе как обычный пользователь, благодаря `sudo` ваш *эффективный* идентификатор пользователя изменился на `root` на время действия одной этой команды:

```
root
```

---

[1] Сокр. от *substitute user (and) do* (*подменить пользователя и выполнить*). — *Примеч. пер.*

[2] От *who am I* (*кто я?*). — *Примеч. пер.*

[3] Скорее всего, перед тем как продолжать, командная оболочка запросит пароль вашего пользователя. Грамотно настроенная система не позволит произвольному пользователю переключиться в режим `root` без соответствующей аутентификации. — *Примеч. науч. ред.*

Давайте посмотрим на более практический пример, где одну особую операцию нужно выполнить от имени `root`, а затем продолжать работать как обычный пользователь:

```
sudo systemctl start nginx
# … и затем вернуться к задачам обычного пользователя …
```

Эта команда запускает веб-сервер `nginx` (если соответствующий пакет установлен в системе), — эту операцию может выполнить только `root`. А все последующие команды снова выполняются от имени рядового пользователя.

Такова обычная практика, которая помогает обеспечить безопасность: бóльшую часть времени вы работаете как обычный пользователь, который не способен своротить всю систему одной неосторожной командой. Но когда вам нужны возможности суперпользователя, то вы получаете их, указывая `sudo` *только перед теми командами*, которые требуют расширенных прав. Это своеобразный психологический барьер, который мешает случайно сломать что-то в системе.

Этот прием используется для различных потенциально опасных операций: например, когда нужно редактировать конфигурационные файлы на уровне системы, создавать каталоги за пределами вашего домашнего каталога (см. далее в этой главе) и т. д.:

- `sudo mkdir /var/log/foobar`
- `sudo vim /etc/hosts`
- `sudo mount /dev/sdb1`

Если вы собираетесь выполнять целую последовательность команд от имени `root`, то с помощью `sudo` с флагом `-i`[1] можно инициировать долговременный сеанс командной оболочки с правами суперпользователя. Например, это может пригодиться, если вы устраняете неполадки службы, которая должна запускаться от имени `root`, или имитируете окружение, в котором должен выполняться сценарий `cloud-init`:

```
sudo -i
```

В этом случае у вас запустится интерактивный сеанс командной оболочки от имени пользователя `root`. Работайте в нем осмотрительно, потому что одна-единственная опечатка или неправильно составленная команда может погубить всю систему!

---

[1] От *initial login* (*начальный вход в систему*). `sudo -i` приводит к тому, что командная оболочка ведет себя так, как будто изначально была запущена от имени `root`. Интересно, что длинная форма флага `-i` не `--initial`, а `--login`. — *Примеч. науч. ред.*

По умолчанию `sudo` подменяет текущего пользователя пользователем `root`, но с помощью флага `-u`[1] можно переключиться в режим какого-нибудь третьего пользователя, например:

```
sudo -u имя_пользователя vim /home/myuser/.bashrc
```

В файле `/etc/sudo.conf` точно определяется, какие операции разрешены каждому пользователю (и каждой группе). Категорически не рекомендуется редактировать этот файл непосредственно; чтобы вносить в него изменения, используйте команду `visudo`.

Что такое группа

Группа — это еще один системный примитив, который позволяет наделять совокупность пользователей одинаковыми правами. Этот механизм часто применяют, чтобы получить функциональность набора разрешений или профиля. Например, в Linux часто бывает группа, которая называется `sudoers`[2], а в macOS — `wheel`[3]. По традиции пользователям, которые входят в эти группы, разрешено вызывать `sudo`, чтобы запускать команды от имени `root`. Функционально эти группы ведут себя так же, как «Администраторы» (`Administrators`) в Windows.

Вы наверняка уже догадались, что поскольку с помощью групп можно управлять тем, кому разрешено запускать команду `sudo`, то с их же помощью можно объединять пользователей и с другими разрешениями.

Мини-проект: как управлять пользователями и группами

Допустим, что мы хотим разрешить каждому разработчику ПО в компании открывать в режиме чтения некоторый файл — назовем его `document.txt`. Для этого можно начать с того, чтобы просто создать группу `developers`[4] и включить в нее всех разработчиков.

Затем, когда мы будем настраивать владельцев и разрешения для `document.txt`, мы сможем ссылаться на группу `developers`, а не пытаться отслеживать каждого отдельного пользователя, который может в нее входить.

---

[1] Длинная форма — `--user` (*пользователь*). — *Примеч. пер.*

[2] Дословно — «судошники», то есть *те, кто пользуется sudo*. — *Примеч. пер.*

[3] Букв. *колесо*. Это название для суперпользователей впервые появилось в системах, предшествующих Unix, и происходит от жаргонного выражения *big wheel* — букв. *большое колесо*, перен. *большая шишка, влиятельная персона*. — *Примеч. пер.*

[4] *Разработчики*. — *Примеч. пер.*

Как создать пользователя

Если в вашей системе Linux установлена команда `adduser`[1], она позволит интерактивно создать пользователя с именем `dave`. Если она не установлена, это можно исправить с помощью пакета, который обычно называется `useradd` (подробнее о том, как устанавливать пакеты, вы узнаете в главе 9 «Как устанавливать программное обеспечение»).

Если передать этой команде имя пользователя как единственный аргумент, то она предложит настроить свойства пользователя в режиме пошагового мастера. Обратите внимание, что здесь используется `sudo`, потому что только `root` может добавлять или удалять пользователей:

```
$ sudo adduser dave
Adding user `dave' ...                      # Добавляем пользователя dave[2]
Adding new group `dave' (1000) ...          # Добавляем группу dave[3]
Adding new user `dave' (1000) with group `dave' ...
    # Включаем пользователя dave в группу dave
Creating home directory `/home/dave' ...
    # Создаем домашний каталог /home/dave
Copying files from `/etc/skel' ...          # Копируем файлы из /etc/skel
New password:                               # Введите новый пароль
Retype new password:                        # Повторите новый пароль
passwd: password updated successfully       # Пароль успешно обновлен
Changing the user information for dave
    # Настраиваем сведения о пользователе dave
Enter the new value, or press ENTER for the default
    # Введите новые значения или нажмите Enter,
    # чтобы применить значения по умолчанию
Full Name []: Dave                          # Полное имя
Room Number []:                             # Рабочий кабинет
Work Phone []:                              # Рабочий телефон
Home Phone []:                              # Домашний телефон
Other []:                                   # Прочее
Is the information correct? [Y/n] y         # Эти сведения верны? [Да/Нет] Да
```

---

[1] От *add user* (*добавить пользователя*). — *Примеч. пер.*

[2] Фиктивные комментарии на русском языке добавлены для удобства читателей. В настоящем терминале Linux этих «комментариев» нет. — *Примеч. пер.*

[3] Модель безопасности файловой системы Unix требует, чтобы у каждого файла был пользователь-владелец и группа-владелец. При создании пользователя команда `adduser` и другие инструменты автоматически создают одноименную группу и включают в нее нового пользователя, а затем по умолчанию назначают эту группу владельцем всех файлов пользователя. Если бы этого не происходило, то для новых файлов пользователя нужно было специально указывать группу-владельца или создавать новую. Обратите внимание, что пользователь `dave` и группа `dave` — это два разных объекта в системе; технически «собственную» группу пользователя не обязательно именовать так же, как его самого. — *Примеч. науч. ред.*

В этом листинге выделены строки, в которых `adduser` запрашивает данные у пользователя: пароль, полное имя, а также подтверждение того, что мы хотим завести нового пользователя в системе.

Таким образом удобно добавить одного или двух пользователей, но что, если вы работаете на тестовом сервере Linux, где нужно завести учетные записи для 300 крупнейших клиентов? В этом случае пригодится команда `useradd`, которая работает не в интерактивном режиме и принимает свойства пользователя в качестве аргументов. С помощью этой команды легко писать сценарии, которые управляют пользователями (см. «О сценариях командной оболочки» далее в этой главе):

```
useradd --home-dir /home/dave --create-home --shell /bin/zsh -g dave
-G sudoers dave
```

Эта команда создает пользователя `dave`, а также:

- создает и настраивает домашний каталог пользователя (`--home-dir` и `--create-home`);
- связывает с пользователем определенную оболочку (`--shell`);
- с помощью флага `-g` задает для пользователя основную группу `dave` (хотя это может быть и другая группа, например `employees`);
- с помощью флага `-G` добавляет пользователя в дополнительную группу `sudoers` (с этим флагом можно передать несколько имен групп через запятую).

Вот и все: если эта команда завершится успешно, то в системе появится новый пользователь!

Но мы еще не закончили. Допустим, что наш пользователь должен работать над новым сверхсекретным приложением `tutorialinux`, так что давайте создадим группу для этого проекта и добавим в нее пользователя.

Как создать группу

Чтобы создать новую группу `tutorialinux`, используйте команду `groupadd`[1]:

```
groupadd tutorialinux
```

Эта команда создает в системе новую группу, а более конкретно — добавляет соответствующую строку в файл `/etc/group`, где регистрируются все группы, которые существуют в системе Unix. Чтобы убедиться, что группа создана, можно найти ее имя в этом файле с помощью grep:

```
# grep tutorialinux /etc/group
tutorialinux:x:1001:
```

[1] От *group* (*группа*) и *add* (*добавить*). — *Примеч. пер.*

Здесь видно, что в системе существует группа `tutorialinux` с идентификатором (**GID**, или **Group ID**[1]) `1001`.

Мы не будем вдаваться в подробности того, что здесь означает буква `x`[2]. Сейчас достаточно знать, что каждая строка в файле `/etc/group` соответствует одной группе и состоит из значений, разделенных двоеточием. В дальнейшем вам понадобится только имя группы (первое значение), ее идентификатор (третье значение) и перечень пользователей (последнее значение, которое в этом примере пусто).

Как управлять пользователями Linux

Вы уже знаете, что с помощью команды `useradd` можно добавить метаданные пользователя при его создании. А команды `usermod`[3] и `gpasswd`[4] позволяют редактировать свойства уже существующего пользователя. Давайте включим пользователя `dave`, созданного ранее, в новую группу `tutorialinux`, чтобы он мог работать с файлами проекта, которые разрешено просматривать и изменять только членам этой группы.

Как добавить пользователя в группу

Чтобы сменить основную группу пользователя, можно запустить команду `usermod -g` *`имя_группы имя_пользователя`*.

Правда, это не совсем то, чего мы здесь добиваемся: на самом деле пользователь `dave` должен оставаться в одноименной группе `dave`, но мы хотим *дополнительно* включить его в группу `tutorialinux`. Чтобы добавить пользователя в группу, но *не менять* его основную группу, используйте параметры `-aG`[5]:

```
sudo usermod –aG tutorialinux dave
```

Если теперь снова изучить файл `/etc/group`, вы увидите, что пользователь `dave` входит в три группы: `dave`, `sudoers` и `tutorialinux`:

```
grep dave /etc/group

sudoers:x:27:dave
dave:x:1000:
tutorialinux:x:1001:dave
```

---

[1] *Идентификатор группы. — Примеч. пер.*

[2] Второе поле в спецификации группы описывает пароль группы. В современных системах это свойство используется редко. — *Примеч. науч. ред.*

[3] От *user* (*пользователь*) и *modify* (*модифицировать*). — *Примеч. пер.*

[4] От *group* (*группа*) и *password* (*пароль*), хотя команда позволяет управлять не только паролями групп, но и другими атрибутами членства в группах. — *Примеч. пер.*

[5] Длинные формы флагов `-a` и `-G` соответственно `--append` (*добавить*, *дополнить*) и `--groups` (*группы*). — *Примеч. пер.*

С помощью команд из следующей главы, которые позволяют управлять владельцами файлов и правами доступа к ним, вы сможете настроить доступ всех членов группы `tutorialinux` к определенным файлам и каталогам.

После того как тот или иной пользователь перестанет участвовать в проекте `tutorialinux`, вы сможете отозвать его доступ, и для этого не понадобится перенастраивать права доступа к отдельным файлам и каталогам.

Как исключить пользователя из группы

Чтобы исключить пользователя из группы, запустите команду `gpasswd` с флагом `-d`[1]:

```
gpasswd -d  имя_пользователя имя_группы
```

Как удалить пользователя Linux

Команда `userdel` полностью удаляет пользователя Linux:

```
userdel -r имя_пользователя
```

Если нужно сохранить домашний каталог удаляемого пользователя, не указывайте флаг `-r` (`--remove`[2]).

Как удалить группу в Linux

С помощью команды `groupdel` можно удалить группу, которая больше не нужна:

```
groupdel имя_группы
```

Дополнительный материал: что представляет собой пользователь на самом деле

На примере пользователей и групп можно понаблюдать одно удивительное качество Unix и Linux: здесь нет почти никакой «черной магии».

Пользователь Linux — это на самом деле всего лишь идентификатор пользователя (**UID**, или **User ID**), то есть просто-напросто число, которое обозначает пользователя (беззнаковое 32-разрядное целое значение). Идентификатор пользователя `root` — число `0`, а UID всех остальных пользователей больше нуля. То же самое верно и для групп.

Эта информация хранится не в каком-то секретном месте, не в двоичном формате и не в проприетарной структуре данных, с которой может работать только операционная система. Пользователи и группы определены в обычных текстовых

---

[1] Длинная форма — `--delete` (*удалить*). — *Примеч. пер.*

[2] *Удалить*. — *Примеч. пер.*

файлах, и их можно редактировать с помощью простых команд, которые мы рассмотрели в этой книге.

Благодаря такому элегантному решению простые смертные (например, разработчик, который в суматохе подзабыл материал этой главы) могут быстро отследить состояние пользователей и групп в действующей системе, например когда в приложении нужно устранить неполадки, которые возникли из-за неправильно подготовленного системного окружения, где не хватает нужного пользователя приложения. Соответствующие знания также могут пригодиться на собеседовании с работодателем, если зайдет речь о «системной инженерии».

А чтобы закрепить ваше представление о том, как все это устроено, раскроем еще несколько секретов о том, что происходит в Linux на внутреннем уровне, когда мы создаем пользователей и группы и управляем ими.

Метаданные (атрибуты) пользователей

От того, что пользователи идентифицируются числами, не много проку, если эта идентификация не сопровождается содержательными метаданными. Например, вот какие метаданные могут быть связаны с учетной записью с `UID 502`, под которой один из авторов этой книги обычно работет на своем компьютере под управлением Linux или macOS:

- «Человеческое» имя пользователя (`dave`)
- Собственная группа учетной записи (группа `dave`)
- Сведения о членстве в группах (`staff`, `developer` и `wheel`)
- Командная оболочка для входа в систему (`bash`, `zsh` и т. д.)
- Домашний каталог (`/Users/dave/` в macOS или `/home/dave/` в Linux)

Чтобы получить информацию о текущем пользователе, можно запустить команду `id`:

```
# id
uid=0(root) gid=0(root) groups=0(root)
```

По умолчанию все эти сведения о пользователях находятся в нескольких файлах, где каждая запись состоит из полей, разделенных двоеточиями:

- `/etc/passwd` содержит имя пользователя, UID, GID, домашний каталог и оболочку входа для каждого пользователя:

```
root@localhost:~# cat /etc/passwd

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
```

- `/etc/shadow` содержит хешированные пароли пользователей с «солью». Этот файл доступен для чтения только пользователю `root`:

```
root@localhost:~# cat /etc/shadow

root:$6$SPevRPxD94AYwtmF$IOp9k15dnaN8FW8RUpDDQlifLPp9pJ3btgJcMfI
QEs1kT.ZNjDfX66XBOcPOBZzkRcGOb3Rwq6qTsDQ0jiZNh/:19251:0:99999:7:::
daemon:*:19251:0:99999:7:::
bin:*:19251:0:99999:7:::
sys:*:19251:0:99999:7:::
sync:*:19251:0:99999:7:::
```

- `/etc/group` аналогичен `/etc/passwd`, но содержит сведения о группах, а не о пользователях. Вы видели и анализировали этот файл ранее в этой главе.

ПРИМЕЧАНИЕ

Внимание! Хотя полезно знать, как устроены эти файлы, ни один из них *ни в коем случае* не следует редактировать вручную. Создавайте, удаляйте и модифицируйте пользователей и группы с помощью инструментов, которые мы рассмотрели в предыдущих разделах.

Надеемся, что этот краткий теоретический обзор внутренней механики, а точнее статических текстовых файлов, которые определяют пользователей и группы, был для вас поучительным. Мы стремились не только к тому, чтобы вы хорошо представляли себе, как работают эти механизмы, но и к тому, чтобы вы в полной мере осознали, насколько *просто* они устроены. Здесь нет никакой черной магии! Теперь вы можете быть уверены, что не упустите никаких важных деталей в следующий раз, когда будете разбираться, почему приложение не запускается или почему у пользователя нет прав на то, чтобы просматривать тот или иной файл, которым владеет та или иная группа.

О сценариях командной оболочки

Ранее в этой главе мы подчеркивали, что лучше использовать автоматизируемые инструменты (как `useradd`), а не интерактивные мастеры (как `adduser`), несмотря на то что неинтерактивные команды немного сложнее освоить. Возможно, вы задаетесь вопросом: почему бы не пользоваться графическими инструментами вместо того, чтобы зубрить эти многоэтажные команды оболочки?

Дело в том, что в этой книге мы хотели бы привить вам один важный навык — в общем случае выбирать неинтерактивные команды.

Их главное преимущество в том, что они не зависят от пользовательского ввода в режиме реального времени, а значит, их можно использовать в сценариях, ко-

торые позволяют создать сотню пользователей почти так же просто, как одного. Это весьма полезно на практике, когда вы решаете реальные задачи: например, собираете образы Docker, настраиваете множество типовых продакшен-сред или пишете установочные сценарии `cloud-init` для облачных экземпляров.

Вам как разработчикам это наверняка покажется разумным: если автоматизировать те или иные операции, то их можно повторно использовать, они становятся более безопасными и работают быстрее. Если вы уверенно осваиваете команды, которые выполняются не в интерактивном режиме, вы сможете использовать их в сценариях автоматизации, вместо того чтобы прибегать к ручной работе, которая отнимает много времени, плодит лишние ошибки и создает ненужные риски.

Итоги

В этой главе вы изучили основы того, как Linux применяет абстракции пользователей и групп, чтобы управлять процессами, файлами и другими системными ресурсами. Что не менее важно, вы освоили ключевые команды для того, чтобы создавать и модифицировать пользователей и группы в реальной системе. Также вы узнали о важной разнице между пользователем `root` (корневым пользователем) и всеми остальными обычными пользователями системы.

После этого вы выполнили практическое упражнение: завели в системе пользователя, добавили новую группу, отредактировали свойства пользователя, а затем очистили все ресурсы, которые создали.

Наконец, мы заглянули «под капот» привычных команд, чтобы убедиться, что в их внутренней механике нет ничего волшебного: в системах Unix пользователи и группы определяются в обычных текстовых файлах. Это хорошая новость, потому что такой подход облегчит вам многие задачи разработки, например:

- создать образ Docker, в котором ваше приложение запускается от имени определенного пользователя с обычными правами;
- развернуть долгосрочный экземпляр облачной службы с учетными записями пользователей и общей группой для вашей команды специалистов по data science;
- минимизировать потенциальный ущерб от ошибки в вашем локальном тестовом окружении;
- устранить неполадки в приложении, связанные с доступом пользователей и групп: например, решить проблемы с веб-службой, которой нужны права `root`, чтобы открыть секретный файл или выполнить особые операции в системе.

В следующей главе мы задействуем весь этот материал, чтобы более подробно узнать, как устроена модель безопасности Unix, которая опирается на два основных понятия — владение ресурсами и права доступа.

8

Владельцы ресурсов и права доступа

В этой главе мы поговорим о том, как в модели безопасности Linux понятия «пользователи» и «группы» объединяются с понятиями «владельцы ресурсов» и «права доступа» (разрешения). Это объединение примитивов позволяет управлять доступом практически ко всем ресурсам системы Linux — процессам, файлам, сетевым сокетам, устройствам и другим объектам.

Прежде всего вы узнаете, какие важные сведения о файлах можно получить из вывода команды `ls` в расширенном формате (конечно, с особым упором на права доступа). Затем мы рассмотрим распространенные разрешения, которые вы встретите в системах Linux в среде реальной эксплуатации, и, наконец, представим все команды Linux, которые понадобятся, чтобы устанавливать и редактировать права доступа к файлам. До конца главы нам предстоит:

- Научиться ориентироваться в выводе команды `ls` в расширенном формате.
- Рассмотреть атрибуты файлов.
- Понять, как устроено владение файлами и права доступа к ним.
- Разобраться с традиционным камнем преткновения — записью разрешений в восьмеричном формате.
- Освоить практические команды для того, чтобы управлять владельцами ресурсов и правами доступа к ним.

Перечень файлов в расширенном формате

Давайте погрузимся в тему с помощью перечня файлов и каталогов в расширенном формате.

Когда вы осматриваетесь в системе, одних имен файлов и каталогов часто бывает недостаточно. Если вам нужно больше сведений о файлах, используйте команду `ls` с флагом `-l` (вывод в расширенном формате).

Вот пример того, что выводит эта команда, если применить ее к каталогу `/lib` в Linux. Откройте терминал и запустите `ls -l /lib/`:

```
# ls -l /lib/
total 56[1]
drwxr-xr-x 10 root root 4096 Mar  8 02:12 aarch64-linux-gnu
drwxr-xr-x  5 root root 4096 Mar  8 02:12 apt
drwxr-xr-x  3 root root 4096 Mar  8 02:08 dpkg
drwxr-xr-x  2 root root 4096 Mar  8 02:12 init
lrwxrwxrwx  1 root root   39 Jul  6  2022 ld-linux-aarch64.so.1 ->
aarch64-linux-gnu/ld-linux-aarch64.so.1
drwxr-xr-x  3 root root 4096 Mar  4  2022 locale
drwxr-xr-x  3 root root 4096 Mar  8 02:12 lsb
drwxr-xr-x  3 root root 4096 Aug 29  2021 mime
-rw-r--r--  1 root root  386 Feb 16  2023 os-release
drwxr-xr-x  2 root root 4096 Mar  8 02:12 sysctl.d
drwxr-xr-x  3 root root 4096 Apr 18  2022 systemd
drwxr-xr-x 16 root root 4096 Jan 17  2022 terminfo
drwxr-xr-x  2 root root 4096 Mar  8 02:12 tmpfiles.d
drwxr-xr-x  3 root root 4096 Mar  8 02:12 udev
drwxr-xr-x  3 root root 4096 Mar  8 02:12 usrmerge
```

Мы видим здесь много интересной информации; давайте рассмотрим все поля этого вывода по порядку.

Атрибуты файлов

Первое поле содержит атрибуты файла, то есть его тип и права доступа. Другими словами, оно показывает, какого рода файл перед нами и какие разрешения для него заданы. Команда `ls` выводит эту информацию в виде символов, а не в числовом формате, о котором мы узнаем далее в этой главе.

Тип файла

```
-rw-r--r--  1 root root  386 Feb 16  2023 os-release
```

Первый символ в этой строке показывает тип файла. В данном случае дефис (`-`) обозначает, что это обычный файл. Если строка начинается с буквы `l`, то это *сим-*

[1] Значение *total* (*всего, итого*) — количество так называемых *блоков*, которые занимают все перечисленные файлы и каталоги. В современных системах один блок (в контексте команды `ls`) обычно составляет 1 Кбайт (1024 байта). — *Примеч. науч. ред.*

волическая ссылка, то есть специальный файл, у которого нет собственного содержания и который просто ссылается на другой объект в файловой системе. Это то же самое, что ярлык в Windows или псевдоним файла в macOS.

Другие распространенные типы файлов — `d` (каталог) и `c` (символьное устройство). Символьные устройства обычно находятся в каталоге `/dev` и представляют аппаратные устройства ввода, например клавиатуру. Подробнее типы файлов рассматриваются в разделе «Типы файлов» в главе 5 «Файлы в Linux».

Права доступа (разрешения)

```
- rw-r--r--  1 root root  386 Feb 16  2023 os-release
```

Символы `rw-r--r--` в этом примере — так называемые *биты разрешений*, которые показывают, каким пользователям и группам разрешено читать, записывать и выполнять этот файл. Мы поговорим об этом подробнее в разделе «Права доступа» далее в этой главе.

Количество жестких ссылок

```
-rw-r--r--  1 root root  386 Feb 16  2023 os-release
```

Следующее поле (в данном случае `1`) обозначает количество жестких ссылок, которые ведут на этот файл. Жесткие ссылки — это особые указатели, которые связывают имена файлов с их содержимым. В большинстве ситуаций у обычных файлов бывает по одной жесткой ссылке. В отличие от символической ссылки, которая ведет на путь к файлу, жесткая ссылка ведет непосредственно на файл. Если переместить файл, на который указывает символическая ссылка, то она станет недействительной. А жесткая ссылка продолжает указывать на файл, даже если его переместить, переименовать или еще как-то изменить.

Вы наверняка заметили, что с большинством файлов связано только по одной ссылке, а у каталогов их гораздо больше. Это потому, что каждый файл или каталог создает ссылку на свой родительский каталог. Даже у пустого каталога есть две ссылки: одна из них связана с каталогом, который обозначается точкой (`.`) и представляет сам текущий каталог, а другая — с каталогом, который обозначается двумя точками (`..`) и представляет родительский каталог текущего каталога.

Пользователь-владелец

```
-rw-r--r--  1 root root  386 Feb 16  2023 os-release
```

Третье поле показывает, какой пользователь назначен владельцем файла. В нашем примере всеми файлами владеет пользователь `root`. Здесь отображается «челове-

ческое» имя пользователя, но если запустить команду `ls` с флагом `-n`[1], то она выведет числовой идентификатор (UID) пользователя.

Группа-владелец

```
-rw-r--r--  1 root root  386 Feb 16  2023 os-release
```

Следующее поле показывает, какая группа владеет файлом. В нашем примере эта группа — тоже `root`.

Размер файла

```
-rw-r--r--  1 root root  386 Feb 16  2023 os-release
```

Как вы наверняка догадались, следующее поле отвечает за размер файла. Если не задано дополнительных флагов, то размер отображается в байтах. Чтобы это значение было удобнее воспринимать, можно использовать флаг `-h` («человекочитаемый» формат).

Если вы внимательно просматривали список файлов, то заметили, что размер всех каталогов одинаков и равен `4096`, например:

```
drwxr-xr-x  3 root root 4096 Apr 18  2022 systemd
```

Это связано с тем, что место для файлов выделяется в зависимости от их содержимого, а дисковое пространство под каталоги отводится дискретными блоками файловой системы. В большинстве файловых систем минимальный размер блока составляет 4096 байт, поэтому каталогам соответствует значение `4096`.

Эту тему можно развивать глубже, но дальнейшие подробности уже не относятся к повседневной работе программиста, поэтому здесь мы не будем в них углубляться. Если вам все-таки интересно узнать больше, поищите информацию об индексных дескрипторах (inodes) в Linux.

Время последнего изменения

```
-rw-r--r--  1 root root  386 Feb 16  2023 os-release
```

Следующее поле — метка времени последнего изменения файла, то есть дата и время того момента, когда файл в последний раз модифицировался.

---

[1] Длинная форма — `--numeric-uid-gid` (*идентификаторы пользователя и группы в числовом формате*). — *Примеч. пер.*

Имя файла

```
-rw-r--r--  1 root root  386 Feb 16  2023 os-release
```

Наконец мы добрались до имени файла — это единственный атрибут, который вы увидите, если запустите `ls` в обычном режиме (то есть без ключа `-l`). Как правило, это просто текстовое имя файла или каталога, за исключением символических ссылок, когда отображается имя ссылки и путь к файлу или каталогу, на который она указывает.

Владельцы файлов

Чтобы сменить владельца файла или каталога, запустите команду `chown`. Ее синтаксис таков: `chown` *`пользователь:группа путь`*, где *`пользователь`* — имя нужного пользователя, *`группа`* — имя группы, а *`путь`* — абсолютный или относительный путь к файлу или каталогу.

Если вы хотите сменить только пользователя-владельца и не трогать группу, просто опустите двоеточие и имя группы. Естественно, чтобы сменить владельца, вы должны иметь права на эту операцию, поэтому в большинстве случаев команду `chown` запускают от имени `root`.

В следующем листинге из командной оболочки показан файл, которым изначально владеет `root`, а затем его владелец меняется с помощью команды `chown`:

```
bash-3.2$ ls -l mysecret.txt
-rw-r--r-- 1 root staff 0 Apr 12 15:39 mysecret.txt
bash-3.2$ sudo chown dave mysecret.txt
bash-3.2$ ls -l mysecret.txt
-rw-r--r-- 1 dave staff 0 Apr 12 15:39 mysecret.txt
```

Права доступа (разрешения)

Вот один из файлов, которые вы уже видели в предыдущем выводе команды `ls -l`. Мы перенастроили права доступа к нему, чтобы пример был более показательным:

```
- rwxr-xr-x  1 root root  386 Aug  2  13:14 os-release
```

Обратите внимание на биты разрешений:

```
rwxr-xr-x
```

На самом деле это три блока по три бита в каждом. Для наглядности изобразим эти блоки отдельно друг от друга:

```
rwx  r-x  r-x
```

Каждый блок состоит из трех значений, которые отображают права на чтение (`r`, *read*), запись (`w`, *write*) и выполнение (`x`, *execute*) для определенной совокупности пользователей в зависимости от того, какой пользователь и какая группа владеют файлом. Если вместо буквы отображается дефис (`-`), это означает, что у тех или иных пользователей нет прав на соответствующую операцию. Рассмотрим эти разрешения подробнее.

1. Первые три бита — разрешения для пользователя, который владеет файлом. В данном случае владелец — `root`, и он может читать, записывать и выполнять файл (`rwx`).
2. Вторые три бита — разрешения для группы, которая владеет файлом (в данном случае эта группа — тоже `root`). Они выглядят как `r-x`, то есть членам этой группы можно читать файл и выполнять его (но нельзя записывать!). Но поскольку пользователь `root` — тоже владелец файла, его разрешения более приоритетны, и `root` может записывать в этот файл. Права доступа такого рода часто встречаются потому, что у каждого файла в Unix *должна быть* группа-владелец, и если вы не хотите использовать файл совместно с кем-нибудь, то в качестве такой группы можно указать «собственную» группу пользователя-владельца. Разрешения файлов для групп обычно строже, чем для пользователей.
3. Последние три бита — права доступа к файлу для всех остальных пользователей системы («остальной мир» («the world») на жаргоне Unix). Почти всегда это самый ограниченный набор разрешений, потому что большинство файлов не предназначены для того, чтобы делиться ими с кем-то, кроме владельца (а иногда также отдельной группы-владельца). В данном случае мы имеем дело с файлом совместно используемой библиотеки, доступ к которой нужен всем пользователям, поэтому заданы права `r-x` (можно читать и выполнять, но нельзя записывать).

Права доступа в восьмеричном формате

В разрешениях легко ориентироваться, когда они выражены в терминах «чтения», «записи» и «выполнения», однако в Linux и других системах семейства Unix права доступа также представляются в другом важном формате — восьмеричном.

Возможно, вы как разработчик знакомы с разными системами счисления помимо десятичной и знаете, что восьмеричная система — это система счисления с основанием 8. (При этом обычные люди чаще всего пользуются десятичной системой, а компьютеры — двоичной, то есть системой с основанием 2.)

Для каждой комбинации разрешений возможны всего 8 состояний, и поэтому восьмеричная система прекрасно подходит для того, чтобы их отражать.

Вспомните наши 9-битные разрешения, которые были разделены на блоки по три бита. Каждому из этих блоков может соответствовать восьмеричное число, потому что все однозначные числа в восьмеричной системе укладываются ровно

в три бита. Таким образом, 9 бит хватит для того, чтобы представить три восьмеричных числа: одно — для разрешений пользователя, другое — для разрешений группы и третье — для разрешений остального мира.

Восьмеричное число	Двоичное число	Разрешения			Описание	Обозначение
		Чтение	Запись	Выполнение		
0	000	–	–	–	Нет доступа	`---`
1	001	–	–	+	Только выполнение	`--x`
2	010	–	+	–	Только запись	`-w-`
3	011	–	+	+	Запись и выполнение	`-wx`
4	100	+	–	–	Только чтение	`r--`
5	101	+	–	+	Чтение и выполнение	`r-x`
6	110	+	+	–	Чтение и запись	`rw-`
7	111	+	+	+	Чтение, запись и выполнение (полный доступ)	`rwx`

Обратите внимание, что эта таблица построена так, что в ней работает сложение в восьмеричной системе: например, если сложить «только чтение» (4) и «только выполнение» (1), то получится «чтение и выполнение» (5).

Логика, по которой правам присваиваются те или иные числовые значения, может показаться странной и произвольной (так и есть на самом деле!), но вы легко к ней приспособитесь. В профессиональной деятельности вам чаще всего будут встречаться восьмеричные значения 7 (полный доступ), 6 (чтение и запись), 5 (чтение и выполнение), 4 (чтение) и 0 (нет доступа).

Распространенные права доступа

На практике особенно популярны такие права доступа к файлам:

Разрешения	Восьмеричный код	Описание
`-rw-r--r--`	644	Владелец может читать и записывать, все остальные могут только читать
`-rwxr-xr-x`	755	Владелец может делать с файлом что угодно, а все остальные — только читать и выполнять его. Это распространенные разрешения для исполняемых файлов (таких, как сценарии командной оболочки или программы в формате двоичного кода); по умолчанию эти же разрешения задаются для каталогов

Разрешения	Восьмеричный код	Описание
`-rw------`	600	Только владелец файла может читать и записывать его, а больше никому файл не доступен. Типичные примеры файлов с такими разрешениями — секретные ключи, файлы с паролями и другие конфиденциальные данные. Например, SSH не будет работать с ключами, которые доступны для чтения группе или всем на свете; чтобы использовать эти ключи, вам придется изменить разрешения и сделать их секретными

Как сменить владельцев файла (chown) и редактировать права доступа к нему (chmod)

Существуют две команды для того, чтобы управлять владельцами файлов и правами доступа к ним, — `chown` и `chmod`.

Chown — назначить владельцев файла

С помощью команды `chown`[1] можно назначить определенного пользователя и определенную группу владельцами файла. Синтаксис команды таков:

```
chown [параметры]... [пользователь][:[группа]] файл...
```

Для последующих примеров допустим, что мы работаем с таким файлом:

```
$ ls -lh testfile
-rw-r--r-- 1 dave dave 10 Aug 14 16:18 testfile
```

Как сменить владельца файла

Посмотрим, как сделать, чтобы владельцем файла стал пользователь `chris` (при условии, что такой пользователь есть в системе):

```
$ chown chris testfile
$ ls -lh testfile
-rw-r--r-- 1 chris dave 10 Aug 14 16:18 testfile
```

Как сменить пользователя-владельца и группу-владельца файла

В предыдущем примере мы сменили пользователя, который владеет файлом, а если бы мы хотели одновременно с этим сменить группу-владельца на `staff`, то команду `chown` можно было бы запустить так:

---

[1] От *change owner* (*сменить владельца*). — *Примеч. пер.*

```
$ chown chris:staff testfile
$ ls -lh testfile
-rw-r--r-- 1 chris staff 10 Aug 14 16:18 testfile
```

Как рекурсивно сменить владельца файлов

Часто требуется назначить определенного пользователя и/или определенную группу владельцем всех файлов в том или ином каталоге. Это можно сделать с помощью параметра `-R` (или `--recursive`):

```
chown -R dave:staff /home/dave
```

Эта команда рекурсивно назначит пользователя `dave` и группу `staff` владельцами каталога `/home/dave`, а также всех файлов и каталогов внутри него.

Chmod — задать права доступа к файлу

Команда `chmod`[1] позволяет настроить права доступа к файлу или каталогу. В качестве параметров можно указывать разрешения в текстовом или восьмеричном формате:

```
chmod [параметры]... разрешения[,разрешения]... файл...
chmod [параметры]... разрешения_в_восьмеричном_формате файл...
```

Здесь *разрешения* задаются в формате `ugo{+,-}rwx`, где:

1. `ugo` — одна, две или три буквы из набора `u` (user, *пользователь*), `g` (group, *группа*) и `o` (other, *остальные*); если этот компонент не указан, то подразумеваются все три буквы.
2. Знак `+`, чтобы добавить разрешения, или знак `-`, чтобы отменить их.
3. `rwx` — одна, две или три буквы из набора `r` (read, *чтение*), `w` (write, *запись*) и `x` (extcute, *выполнение*).

Например, так можно добавить права на выполнение для пользователя, который владеет файлом:

```
$ chmod u+x testfile
$ /tmp ls -lh testfile
-rwxr--r-- 1 dave dave 10 Aug 14 16:18 testfile
```

А теперь добавим права на запись и выполнение для группы и остальных пользователей:

```
$ chmod go+wx testfile
$ /tmp ls -lh testfile
-rwxrwxrwx 1 dave dave 10 Aug 14 16:18 testfile
```

---

[1] От *change mode* (*изменить режим*). — *Примеч. пер.*

Как быть, если мы на самом деле не хотели добавлять разрешения для «остального мира»? Удалим эти разрешения:

```
$ chmod o-rwx testfile
$ /tmp ls -lh testfile
-rwxrwx--- 1 dave dave 10 Aug 14 16:18 testfile
```

Права можно задавать и в восьмеричном формате:

```
$ chmod 744 testfile
$ /tmp ls -lh testfile
-rwxr--r-- 1 dave dave 10 Aug 14 16:18 testfile
```

А что, если мы хотим, чтобы этот файл был доступен только для чтения, причем только владельцу? Этого тоже легко добиться:

```
$ chmod 400 testfile
$ /tmp ls -lh testfile
-r-------- 1 dave dave 10 Aug 14 16:18 testfile
```

Настройка владения и прав доступа по образцу

Команды `chown` и `chmod` можно запускать с аргументом `--reference`, в котором передается путь к файлу, чьих владельцев и разрешения нужно скопировать в целевой файл[1].

Итоги

В этой главе мы рассмотрели все, что нужно знать, чтобы решать самые распространенные проблемы с правами доступа (разрешениями) в Linux: узнали, как просматривать и редактировать права доступа к файлам. Что не менее важно, мы продемонстрировали, как интерпретировать разрешения и как они связаны с пользователями и группами Linux; это тема, которая нередко сбивает новичков с толку.

Убедитесь, что вы твердо усвоили материал этой главы, ведь огромная доля неполадок, которые вам придется устранять в профессиональной деятельности, связана с владением файлами и с правами доступа к ним. К счастью, бóльшая часть подобных проблем возникает оттого, что люди плохо понимают эту тему, а вам этот недостаток уже не грозит. Смело приступайте к работе и устраняйте неполадки!

---

[1] Например: `chmod --reference=`*`файл_образец целевой_файл`*. У параметра `--reference` нет короткой формы. — *Примеч. науч. ред.*

9

Как устанавливать программное обеспечение

Если вы работаете в Linux или других системах семейства Unix, то время от времени вам требуется устанавливать или удалять программное обеспечение. Обычно это делается с помощью систем управления пакетами, хотя в особых случаях приходится применять другие методы.

Скорее всего, вы уже знакомы с инструментами, которые позволяют управлять библиотеками в вашей среде разработки, — `npm`, `gem`, `pip`, `go get`, `maven`, `gradle` и т. д. Эти системы управления пакетами работают по тому же принципу, что и аналогичные компоненты Linux и Unix.

Системы управления пакетами (так называемые пакетные менеджеры) инкапсулируют многочисленные двоичные и конфигурационные файлы, из которых состоит программное обеспечение, в монолитные установочные пакеты. Это понятие должно быть вам знакомо, если раньше вы работали с Windows, где установочные комплекты встречаются в виде файлов `.exe` или `.msi`, либо с macOS, где применяются образы дисков в формате `.dmg`.

Большинство систем управления пакетами в Linux также добавляют отдельный уровень безопасности:

- загружают пакеты по защищенному каналу (TLS);
- используют криптографические подписи пакетов, чтобы подтвердить, что их авторы — по крайней мере те, за кого себя выдают (а доверяете ли вы им — уже другой вопрос).

Именно для различных дистрибутивов Linux впервые появились пакетные репозитории с возможностью поиска, благодаря которым становится легче найти и загрузить нужное программное обеспечение. Принципы этих репозиториев сейчас воплощены в магазинах приложений, например App Store или Microsoft Store.

В этой главе мы рассмотрим такие темы:

- Что такое системы управления пакетами.
- Какие системы управления пакетами наиболее популярны.
- Какие важнейшие операции управления пакетами вам понадобятся и какие команды позволяют их выполнять в разных системах; здесь вы получите примерно 90 % всех навыков, которые потребуются на практике.
- Как загружать и выполнять специализированные установочные сценарии.
- Как локально собирать и устанавливать программное обеспечение из исходного кода (краткое практическое введение).

Если вас интересует, как устанавливать программы в Docker, обратитесь к главе 15 «Контейнерная виртуализация приложений с помощью Docker».

Начнем с самого главного: команды для управления пакетами пригодятся вам как разработчику, чтобы решать распространенные задачи:

- устанавливать новые программные пакеты, например зависимости, которые нужны вашему приложению в среде выполнения;
- проверять, какие пакеты установлены (например: «Установлен ли уже в этой системе nginx?»);
- обновлять текущий набор установленных приложений, чтобы гарантировать, что вы пользуетесь самыми свежими версиями всего программного обеспечения. Это часто полезно, чтобы ликвидировать обнаруженные уязвимости или добиваться, чтобы в вашем распоряжении были все новейшие функции той или иной программы;
- удалять пакеты из системы.

Давайте посмотрим, как выполнять эти задачи и какие команды для этого понадобятся.

Как работать с системами управления пакетами

Прежде чем переходить к практическим командам, следует знать, что они могут различаться в зависимости от дистрибутива Linux. В разных дистрибутивах доступны разные системы управления пакетами: хотя все они работают почти идентично, синтаксис их команд различается *ровно настолько*, чтобы досаждать пользователям. К популярным системам управления пакетами относятся:

- `homebrew`[1] (macOS);

[1] *Ручной работы, самодельный, крафтовый. — Примеч. пер.*

- `apt`[1] (практически всегда есть в системах семейств Ubuntu и Debian, даже в минималистичных, где не установлен `aptitude`[2]);
- `pacman`[3] (Arch Linux);
- apk[4] (Alpine Linux);
- dnf (CentOS, RHEL).

В последующих разделах, которые посвящены практической работе с пакетами, мы будем сначала обозначать верхнеуровневую задачу, которую нужно выполнить (например, «как установить тот или иной пакет»), а затем демонстрировать конкретные команды, с помощью которых этого можно добиться в популярных системах управления пакетами.

Как обновить локальный кэш пакетов

Перед тем как устанавливать или удалять пакеты, стоит убедиться, что локальный кэш пакетов (хранящийся в вашей системе каталог пакетов, которые доступны в интернете) находится в актуальном состоянии.

Например, если вы устанавливаете веб-сервер nginx, но локальный кэш обновлялся месяц назад, то вы рискуете непреднамеренно установить устаревшую версию от прошлого месяца, а не обновление за эту неделю.

Чтобы обновить кэш, найдите свою систему управления пакетами в этом списке и запустите соответствующую команду:

Система управления пакетами	Команда
`homebrew`	`brew update`
`apt`	`apt update`
`pacman`	`pacman -Sy`
`apk`	`apk update`

[1] Аббр. *Advanced Package Tool* (*продвинутая система управления пакетами*). Также *apt* — *подходящий, приходящийся кстати*. — *Примеч. пер.*

[2] Букв. *способность, талант, предрасположенность* (к чему-л.); также обыгрывается связь с системой управления пакетами apt, поверх которой работает `aptitude`. — *Примеч. пер.*

[3] Сокр. *package manager* (*система управления пакетами*); также *Pac-Man* — название культовой видеоигры начала 1980-х годов и имя ее главного персонажа. — *Примеч. пер.*

[4] Аббр. от *Alpine Package Keeper* (*хранитель пакетов Alpine*). — *Примеч. пер.*

После этого ваш локальный кэш доступных пакетов будет обновлен и можно будет переходить к следующей увлекательной процедуре — искать и устанавливать пакеты.

Как найти пакет

Не все пакеты называются так же, как программное обеспечение, которое в них находится. Например, браузер Firefox обычно доступен в одноименном пакете `firefox`, но с `ag` совсем другая история: если вы попытаетесь установить пакет с таким именем, то будете разочарованы, потому что на самом деле нужный пакет называется `silversearcher-ag`. С помощью следующих команд можно найти названия и/или описания пакетов по ключевому слову:

Система управления пакетами	Команда
`homebrew`	`brew search` *ключевое_слово*
`apt`	`apt-cache search` *ключевое_слово*
`pacman`	`pacman -Ss` *ключевое_слово*
`apk`	`apk search` *ключевое_слово*
`dnf`	`dnf search` *ключевое_слово*

Это хороший способ убедиться в том, что вы нашли именно тот пакет, который вам нужен, но при этом можно также распространить поиск на более широкий спектр программ, относящихся к вашей задаче. Например, чтобы в Ubuntu найти все инструменты, похожие на `grep`, можно запустить команду `apt-cache search grep`. Она перечислит все пакеты, в названии или описании которых есть строка `grep`.

Как установить пакет

И наконец, наступает главное событие! После того как кэш актуализирован *и* мы точно знаем, какой пакет хотим установить, можно запустить команду установки:

Система управления пакетами	Команда
`homebrew`	`brew install` *имя_пакета*
`apt`	`apt install` *имя_пакета*
`pacman`	`pacman -Sy` *имя_пакета*
`apk`	`apk add` *имя_пакета*
`dnf`	`dnf install` *имя_пакета*

Когда система управления пакетами запросит подтверждение, подтвердите операцию, и пакет будет установлен вместе со всеми остальными пакетами, от которых он зависит.

Поскольку некоторые команды запрашивают подтверждение, прежде чем установить пакет, это может заблокировать сценарий. Когда вы автоматизируете задачи с помощью сценариев и вам нужно, чтобы сценарий устанавливал пакет, обратитесь к справочной странице (`man`) соответствующей системы управления пакетами и узнайте, как устанавливать пакеты в неинтерактивном режиме. Чаще всего это делается с помощью переменной окружения или дополнительного аргумента команды.

Как обновить все пакеты, для которых доступны обновления

Если система работает достаточно долго, то имеет смысл время от времени обновлять установленные пакеты до последних версий. Обычно при этом исправляются обнаруженные уязвимости, становятся доступными новейшие функции, а системы, которые были установлены с разницей в несколько месяцев, лучше согласуются друг с другом.

Система управления пакетами	Команда
`homebrew`	`brew upgrade`
`apt`	`apt dist-upgrade`
`pacman`	`pacman -Syu`
`apk`	`apk upgrade`
`dnf`	`dnf update`

Эти команды тоже запрашивают подтверждение, так что если вы используете их в сценариях, то найдите аналогичный способ запустить их неинтерактивно. Например, команда `apt -y dist-upgrade` просто обновит пакеты, не ожидая подтверждения в ручном режиме.

Как удалить пакет (и все его зависимости, если они не нужны другим пакетам)

Иногда нужно удалить пакет: например, если вы установили его, чтобы просто опробовать, или если требования вашего приложения изменились, или если в пакете обнаружена уязвимость, к которой не предвидится исправления в обозримом будущем. Во всех системах управления пакетами есть команды для того, чтобы удалить установленный пакет:

Система управления пакетами	Команда
homebrew	`brew remove` *имя_пакета*
apt	`apt remove` *имя_пакета*
pacman	`pacman -Rs` *имя_пакета*
apk	`apk del` *имя_пакета*
dnf	`dnf remove` *имя_пакета*

Однако перед тем как применить команду удаления пакета (а иногда и после этого), нередко имеет смысл проверить, а установлен ли вообще данный пакет. Посмотрим, как это сделать.

Как анализировать список установленных пакетов

Чтобы просмотреть список всех пакетов, которые установлены в системе, можно запустить такую команду:

Система управления пакетами	Команда
homebrew	`brew list`
apt	`dpkg -l`
pacman	`pacman -Qi`
apk	`apk info`
dnf	`rpm -qa`

Правда, работать с этим списком не слишком удобно, потому что в нем часто бывают сотни или тысячи пакетов. Чтобы отфильтровать вывод, можно передать его по конвейеру команде поиска вроде `grep`:

```
dpkg -l | grep silversearcher
ii  silversearcher-ag  2.2.0+git20200805-1  arm64  very fast grep-like
program, alternative to ack-grep[1]
```

Если вам не вполне понятно, как здесь работает конвейер, который перенаправляет вывод из `dpkg` на вход команды `grep`, обратитесь к главе 1 «Как устроена командная строка», где вкратце рассказывается, как объединять команды в це-

[1] Описание пакета: *очень быстрая программа, подобная `grep`; замена `ack-grep`. — Примеч. пер.*

почки с помощью символа конвейера (|). А гораздо подробнее мы рассмотрим эту тему в главе 11 «Конвейеры и перенаправление ввода-вывода».

Итак, мы перечислили основные команды, которые пригодятся вам в 90 % случаев, когда понадобится управлять пакетами. Теперь давайте узнаем о том, что делать, когда для программного обеспечения, которое вы хотите установить, нет готовых установочных пакетов.

Будьте осторожны с конвейером curl | bash

Иногда нужная программа не поставляется в виде готового установочного пакета. В этом нет ничего плохого: даже авторитетное и популярное программное обеспечение, например `homebrew` в macOS, рекомендует процедуру установки через команду такого вида:

```
curl URL | bash
```

Здесь команда `curl`[1] загружает данные из Сети, а затем передает их на вход команды `bash` с помощью символа конвейера |, о котором мы говорили в главе 1 «Как устроена командная строка». При этом вы фактически запускаете сценарий из интернета, а не из локального файла. Иногда таким способом очень удобно устанавливать программное обеспечение, но вы должны быть *абсолютно* уверены в том, что сценарий поступает из надежного источника.

Мы советуем всегда как минимум *просматривать* исходный код сценария; для этого посетите его URL в обычном браузере. Также можно разбить единую команду `curl` *URL* `| bash` на несколько команд, благодаря которым вы сможете выполнить необходимые действия:

- загрузить сценарий;
- просмотреть его исходный код в текстовом редакторе на своем компьютере, чтобы убедиться, что он не делает ничего вредоносного, а также, если нужно, отредактировать код в соответствии со своими требованиями;
- запустить сценарий после того, как вы убедились, что он делает только то, что нужно.

Вот как можно разбить команду типа `curl` *URL* `| bash` на отдельные шаги:

```
# Загрузить установочный сценарий и сохранить его в файле installer.sh
curl адрес_URL -o installer.sh
```

[1] Сокр. *client for URL* (*клиент для URL*). В свою очередь, *URL* — аббр. *Unform Resourse Locator* (досл. *единообразный указатель местонахождения ресурса*). — *Примеч. пер.*

```
# Просмотреть и по необходимости отредактировать сценарий
# в текстовом редакторе — например, vim
vim installer.sh

# Сделать сценарий исполняемым и запустить его
chmod +x installer.sh
./installer.sh
```

В этом примере мы не стали запускать непроверенный сценарий сразу после того, как загрузили его, а разбили процедуру на несколько шагов, чтобы просмотреть код сценария и убедиться, что он безопасен. Конечный результат получится тем же самым (вы запустите установочный сценарий), но при этом вы будете лучше контролировать происходящее, а не слепо доверять сценарию.

Однако есть и еще один способ установить программное обеспечение в системе Linux, даже если у него нет готового установочного сценария.

Как компилировать стороннее ПО из исходного кода

Компилировать и настраивать программное обеспечение в ручном режиме — это самый трудоемкий и долгий способ устанавливать программы в системе. При этом вам недоступны такие преимущества систем управления пакетами, как скорость, воспроизводимость и удобство управления, а также нет возможности криптографически проверить двоичный код, который вы устанавливаете.

Однако, несмотря на все сложности, это традиционно самый надежный способ установить ПО, не прибегая ни к каким сторонним зависимостям помимо стандартных инструментов программирования (компилятор, компоновщик и make-файл), с которыми вы уже знакомы как разработчик.

Вот некоторые случаи, когда придется вручную компилировать и устанавливать программное обеспечение:

- В доступной системе управления пакетами нет готового установочного пакета для нужной программы. Например, в минималистичных контейнерных дистрибутивах (таких, как Alpine) «пакетные менеджеры» обеспечивают далеко не все, что вам нужно. В этом случае вы можете сами скомпилировать исходный код в двоичные файлы и вручную добавить их в контейнер.
- Вам нужно добавить в контейнер Docker свое собственное или другое специально разработанное программное обеспечение.
- Вам необходима самая свежая, «с пылу с жару» версия программы, для которой еще не существует установочного пакета. Это может понадобиться, если вы имеете дело с медленно развивающимся проектом, чьи обновления не всегда

сопровождаются новыми пакетами, а также если вам *немедленно* нужно развернуть исправление для критической уязвимости, не дожидаясь, пока оно пройдет цикл подготовки пакета.

Вся процедура состоит из нескольких этапов и может незначительно различаться в зависимости от того, что и куда вы устанавливаете. Обычные этапы таковы:

1. С помощью `curl` или `wget` загрузить сжатый архив с исходным кодом ПО.
2. Запустить `tar zxf` *`файл_архива`* или `unzip` *`файл_архива`*, чтобы разархивировать и распаковать каталог с исходным кодом, который вы только что загрузили.
3. Перейти в каталог с исходным кодом и прочесть в нем все файлы `README`. Обычно там указано, как конкретно установить это программное обеспечение, а также чем установка отличается от стандартной процедуры, которую мы здесь описываем.
4. Чтобы собрать и установить двоичный код программы, запустить `./configure`, затем `make`, а затем `sudo make install`.

Когда вы устанавливаете программы — неважно, вручную или с помощью системы управления пакетами, — не забывайте, что команды `configure` и `make` *по своей природе* устроены так, что выполняют произвольный код, который им передали. То есть если вы запустите `make install` от имени пользователя `root`, то весь код будет выполнен от его имени. Относитесь к этому ответственно! Убедитесь, что исходный код безопасен и что вы загружаете его из надежного источника.

Пример: компилируем и устанавливаем htop

Чтобы отточить навыки на реальном примере, давайте загрузим, откомпилируем и установим `htop`[1] — маленький и очень полезный инструмент для мониторинга системы (он похож на встроенную команду `top`, но во многом ее превосходит). Заметим, что практически в любом дистрибутиве Linux эту программу легко можно установить с помощью системы управления пакетами, но для данного упражнения мы представим себе, будто это редкая специализированная программа, которая не распространяется через пакеты.

Мы будем устанавливать `htop` на серверной версии Ubuntu 22.04, так что если вы собираетесь воспроизводить процесс на своем компьютере, проще всего будет использовать тот же дистрибутив.

---

[1] Автор программы `htop` Хишам Мухаммад (Hisham Muhammad) составил ее название, соединив первую букву своего имени (*h*) с названием существующей программы `top`. — *Примеч. пер.*

Первым делом мы узнаем, каков последний выпуск программы в официальном репозитории GitHub (github.com/htop-dev/htop/releases); на момент подготовки книги к печати самой свежей была версия 3.3.0.

После этого имеет смысл завести каталог для этой сборки, чтобы не создавать беспорядка в системе. Мы рекомендуем создать этот каталог в каталоге `/tmp`[1], где хранятся временные файлы, которые удаляются при каждом запуске системы:

```
mkdir /tmp/htopbuild && cd /tmp/htopbuild
```

В результате мы сможем удалить все лишнее после того, как закончим сборку, чтобы не засорять систему ненужными файлами от прежних сборок. Теперь мы готовы приступить к основной процедуре.

Предварительные условия для установки

Прежде всего нужно установить набор основных инструментов для разработки на C: компилятор, компоновщик, `make` и другие средства, которые нужны, чтобы компилировать код на C в системе Linux. В Ubuntu для этого можно установить метапакет `build-essential`[2] (*метапакет* — это пакет, который функционирует как псевдоним для комплекта других пакетов):

```
sudo apt install build-essential
```

Установим также другие инструменты, которые нам понадобятся: `wget`[3] — чтобы загружать файлы из интернета, и библиотеку разработки `ncurses`[4], которая нужна программе `htop` для интерактивного интерфейса командной строки:

```
sudo apt install wget libncurses-dev[5]
```

---

[1] От *temporary* (*временный*). — *Примеч. пер.*

[2] *Build* — здесь *сборка ПО*; *essential* — *существенный*, *первостепенный*. — *Примеч. пер.*

[3] От *World Wide Web get* (прибл. *получить из Всемирной паутины*). На момент издания этой книги набирает популярность `wget2` — улучшенная версия `wget`. Чтобы установить ее, укажите имя пакета `wget2` вместо `wget`. — *Примеч. науч. ред.*

[4] `ncurses` — библиотека для интерфейса текстовых терминалов, обновленная реализация классической библиотеки `curses`; буква *n* в названии — от *new* (*новый*). В свою очередь, название `curses` — игра слов: буквально это слово означает *проклятия*, *ругательства*, но официально расшифровывается как *cursor optimization* (*оптимизация курсора*). — *Примеч. пер.*

[5] В названии пакета `libncurses-dev` часть `lib` — сокр. *library* (*библиотека*), `dev` — сокр. *developer* (*разработчик*). В именах пакетов, которые представляют собой программные библиотеки, часто встречается префикс `lib`. «Суффиксом» `-dev` также часто помечают программное обеспечение, которое предназначено для разработчиков. — *Примеч. науч. ред.*

Загружаем, проверяем и распаковываем исходный код

На этом этапе мы первым делом загрузим исходный код и проверим его криптографическую подпись, чтобы убедиться, что это подлинный выпуск, который удостоверен ключом разработчика:

```
wget https://github.com/htop-dev/htop/releases/download/3.3.0/htop-3.3.0.tar.xz
```

Таким образом мы получим сжатый и заархивированный каталог[1] с исходным кодом, который будем компилировать, чтобы получить двоичные файлы.

Давайте теперь убедимся, что это авторизованный выпуск: для этого мы проверим подпись, которая представляет собой просто хеш-сумму исходного кода в формате SHA-256.

Загрузим файл, который содержит хеш-сумму для этого выпуска, и выведем его в терминал:

```
wget https://github.com/htop-dev/htop/releases/download/3.3.0/htop-
3.3.0.tar.xz.sha256
cat htop-3.3.0.tar.xz.sha256
```

Если вы работаете с той же версией 3.3.0, что и мы, то увидите такое значение:

```
a69acf9b42ff592c4861010fce7d8006805f0d6ef0e8ee647a6ee6e59b743d5c  htop-
3.3.0.tar.xz
```

Теперь с помощью `sha256sum` захешируем исходный код, который мы загрузили до этого, чтобы убедиться, что хеш-суммы совпадают[2]:

```
sha256sum htop-3.3.0.tar.xz
a69acf9b42ff592c4861010fce7d8006805f0d6ef0e8ee647a6ee6e59b743d5c  htop-
3.3.0.tar.xz
```

---

[1] Новичков в Linux часто сбивает с толку то, что сжатие и архивация — две разные операции. В системах семейства Unix *архивация* традиционно означает, что фрагмент файловой системы (или даже целая система) объединяется в один файл, из которого затем можно восстановить этот фрагмент со всеми атрибутами файлов и каталогов; данные при этом не сжимаются. В свою очередь, *сжатие* в Unix может применяться к произвольным данным (неважно, архив это или нет). Суффиксы `.tar.xz` в названии файла означают, что это архив в формате `tar`, дополнительно сжатый в контейнер XZ. — *Примеч. науч. ред.*

[2] В мире Unix не принято сравнивать хеш-суммы на глаз. Чтобы автоматизировать проверку того, совпадает ли хеш-сумма файла `htop-3.3.0.tar.xz` с контрольным значением из файла `htop-3.3.0.tar.xz.sha256`, в данном случае можно было запустить команду: `sha256sum -c htop-3.3.0.tar.xz.sha256`. Вывод `htop-3.3.0.tar.xz: OK` будет означать, что проверка прошла успешно и хеш-суммы совпадают. Подробнее см. `man sha256sum`. — *Примеч. науч. ред.*

Великолепно! Вот мы и убедились, что архив, который у нас есть, совпадает с официальным выпуском, который мы хотели загрузить. Давайте разархивируем каталог с исходным кодом и перейдем в него:

```
tar xf htop-3.3.0.tar.xz
cd htop-3.3.0
```

Теперь самое время прочесть файл `README` с общими сведениями о программе, а также файл `INSTALL` с инструкциями о том, как собрать и установить программу.

Конфигурируем и компилируем htop

Теперь пора запустить сценарий `./configure`[1] из каталога с исходным кодом. Этот сценарий проверяет, доступны ли все зависимости, которые нужны для компиляции (общие библиотеки, инструментарий и т. д.), и настраивает конфигурацию для того, чтобы скомпилировать программу:

```
./configure
```

Вывод этой команды состоит из большого количества строк, которые сообщают о том, обеспечены ли все зависимости и готова ли ваша среда для компиляции.

Если этот сценарий выводит сообщения об ошибках, прочтите их внимательно: обычно они в доступной форме дают представление о том, в чем проблема: например, если не хватает какой-то библиотеки или если в операционной системе обнаружены несовместимые настройки. Исправьте выявленные проблемы и запустите сценарий снова. После того как он успешно завершит работу, можно компилировать двоичный файл `htop` командой `make`[2]:

```
make
```

Эта команда снова производит целую «простыню» сообщений о том, что делает сценарий компиляции. Если вы до сих пор не встречались с make-файлами, то имейте в виду, что это чрезвычайно полезный инструмент автоматизации, который весьма популярен среди разработчиков. Вот прекрасное пособие на эту тему: makefiletutorial.com.

После того как компиляция закончится, в текущем каталоге появится двоичный файл, который называется `htop`. Его можно установить в систему; как правило, это делается автоматически:

```
sudo make install
```

---

[1] *Конфигурировать, настраивать. — Примеч. пер.*

[2] Сделать. — *Примеч. пер.*

Здесь требуется `sudo`, потому что в процессе установки вы перемещаете скомпилированные двоичные файлы в системное местоположение, которым владеет `root`. После этого можно проверить, что `htop` установлен и работает; для этого просто запустите эту программу:

```
htop
```

Если все в порядке, то вы увидите живописный текстовый интерфейс, который использует библиотеку `ncurses` и отображает состояние вашей системы: текущую нагрузку на центральный процессор, потребление памяти и список процессов.

Если полноценная команда `install` не поставляется с программным обеспечением, вы всегда можете полагаться на то, что в Linux нет никакой магии, поэтому двоичный файл можно установить вручную, переместив его в каталог `/usr/local/bin`, где хранятся откомпилированные двоичные файлы:

```
mv htop /usr/local/bin/
```

Теперь вы убедились, что компилировать ПО из исходного кода совсем не сложно? Попрактикуйтесь в этом деле и переходите к следующей теме!

Итоги

В этой главе вы узнали основы того, как управлять программным обеспечением, которое установлено в вашей среде Linux. Сначала мы рассмотрели, как осуществлять это простым способом — с помощью систем управления пакетами, с которыми вы практически наверняка будете иметь дело. Хотя в 90 % случаев вам хватит этого способа, мы также продемонстрировали, что делать в оставшихся ситуациях: тщательно убедиться, что файлы и их поставщик заслуживают доверия, а затем запустить специальный установочный сценарий или вручную откомпилировать и установить программу.

Надеемся, что по ходу изучения этой главы вы практиковались в командной строке и благополучно установили системный монитор `htop`. К счастью, эта программа повсеместно доступна через системы управления пакетами. Это чрезвычайно полезный инструмент, который многие системные администраторы считают незаменимым помощником в продакшен-средах, рассчитанных на долговременную работу.

Теперь вы должны уверенно ориентироваться как в верхнеуровневых понятиях, так и в практических командах, которые нужны для того, чтобы устанавливать приложения в большинстве систем семейства Unix и Linux — и в среде разработки, и в среде реальной эксплуатации.

10

Как настраивать конфигурацию приложений

Рано или поздно каждому разработчику приходится настраивать конфигурацию приложений в системе Linux. Это можно делать по-разному, однако, к счастью, существует стандартная процедура, которая позволяет добиться нужных результатов. Она особенно распространена в Linux, где большинство популярных инструментов следуют принципу «будь узкоспециализированным, но острозаточенным». В результате прослеживается тенденция к широкому ассортименту небольших, но мощных программ, гибкость которых обеспечивается благодаря богатым возможностям конфигурации.

В этой главе вы познакомитесь с иерархией конфигурации, которой следуют хорошо продуманные программы. Вы узнаете, как настраивать конфигурацию, — независимо от того, требуется ли вам всего лишь найти нужный аргумент командной строки для утилиты, которую вы запускаете один раз, или задать переменную окружения для всех команд в вашей оболочке.

Здесь мы продемонстрируем стандартную иерархию конфигурации, которая применяется практически ко всему программному обеспечению Linux, так что вы всегда сможете проверить, работает ли программа так, как нужно, с учетом конфигурации, которую вы для нее настроили.

Наконец, вы увидите, как эта конфигурация применяется к программам под управлением `systemd` — самой популярной системы управления службами в Linux.

Эта глава охватывает такие темы:

- Иерархия конфигурации.
- Аргументы командной строки.
- Переменные окружения.

- Конфигурационные файлы.
- Конфигурация для Docker.

Иерархия конфигурации

Если вы запускаете программы в Linux, то одна из главных задач, которой вам придется заниматься, — настройка этих программ под ваши индивидуальные нужды. Фактически вы уже это проделывали: когда вы передавали аргументы командам вроде `ls`, `grep` и др., вы конфигурировали их поведение.

Скорее всего, у вас уже есть интуитивное представление о том, как работает конфигурация, потому что на протяжении всей своей карьеры разработчика вы имеете дело с программным обеспечением. Например, вы наверняка привыкли к тому, что аргументы командной строки переопределяют поведение программы по умолчанию: скажем, `ls -l` выводит не то же самое, что просто `ls` без флагов.

Давайте систематизируем это интуитивное представление и сформулируем эвристические правила о том, как *обычно* устроена конфигурация в средах Unix. В частности, большинство стандартных программ Unix с интерфейсом командной строки следуют определенной иерархии конфигурации, где значения, которые заданы раньше, перекрываются значениями, заданными позже. Если вы писали программное обеспечение, которое принимает пользовательские настройки, то наверняка уже настраивали приоритеты конфигурации наподобие таких:

1. Присвоить встроенные значения по умолчанию параметрам, которые можно конфигурировать.
2. Извлечь из конфигурационных файлов значения, которые переопределяют эти стандартные настройки.
3. Проверить, есть ли переменные окружения, которые переопределяют настройки конфигурационных файлов и другие ранее заданные значения.
4. Если нужно, также обновить значения в соответствии с аргументами, которые переданы в командной строке.

Чем выше уровень этой иерархии, тем ближе он находится к пользователю, который запускает программу в тот или иной момент, и поэтому каждый следующий уровень имеет приоритет над предыдущими.

Например, если ваша программа обнаружила, что аргументы командной строки, с которыми она была вызвана, конфликтуют со значениями в конфигурационном файле, то программа должна предпочесть значения из командной строки. Иначе говоря, **значения, которые находятся ближе к вызову программы, «заслоняют собой» значения, которые находятся дальше**. Аргумент командной строки при-

оритетнее, чем настройка из конфигурационного файла, потому что этот файл находится дальше от точки вызова, чем аргументы, которые переданы в командной строке в момент запуска программы. Это интуитивно понятно: нельзя доверять программе, которая игнорирует флаги командной строки в пользу настроек по умолчанию. Команда `ls -l` не должна выводить то же самое, что просто `ls`.

Большинство программ для Linux, которые можно конфигурировать несколькими способами, следуют этой иерархии. Однако имейте в виду, что бывают программы, получающие свою конфигурацию не из всех перечисленных источников, а некоторые программы не полностью соблюдают указанные приоритеты.

Давайте еще раз проследим эту иерархию, но теперь на практическом примере конкретного программного обеспечения — веб-сервера **nginx**. Если вы еще не встречались с ним на практике, то наверняка встретитесь, потому что это один из самых популярных веб-серверов в мире, и с его помощью работают самые разные динамические веб-приложения. Посмотрим, как в конфигурации nginx отражается каждый уровень иерархии приоритетов, которую мы только что рассмотрели:

1. **Встроенные значения.** Пользователю, который запускает nginx, по умолчанию назначается имя `nobody`.
2. **Глобальные конфигурационные файлы** (на уровне системы) могут переопределить это имя для всех процессов nginx. В глобальном конфигурационном файле nginx `/etc/nginx/nginx.conf` часто встречается строка `user www;`, которая заставляет веб-сервер запускаться от имени пользователя `www`.
3. **Пользовательские конфигурационные файлы**, как правило, находятся в домашнем каталоге пользователя (`/home/`*`имя_пользователя`*), а их имена начинаются с точки (`.`), благодаря чему они не включаются в стандартный вывод команды `ls`. Например, пользовательская конфигурация командной оболочки Bash для пользователя `dave` обычно находится в файле `/home/dave/.bashrc`. Долговременный процесс `nginx`, как правило, запускается не от имени обычного пользователя Linux; чаще конфигурация каждого сайта содержится в отдельном файле `/etc/nginx/conf.d/`*`имя_сайта`*`.conf` и наследует значения от глобального конфигурационного файла `nginx.conf`, который находится на предыдущем уровне иерархии.
4. **Переменные окружения.** nginx получает сведения о часовом поясе из переменной окружения `TZ`[1].
5. **Аргументы командной строки** задаются, когда программа запускается — либо вручную, либо автоматически (например, с помощью планировщика cron или файлов модулей). Если вы устраняете неполадки, не забудьте проверить, какие

[1] Аббр. от *timezone* (*часовой пояс*). — *Примеч. пер.*

аргументы командной строки могли поступить из подобных сторонних источников. Они часто виноваты в том, что поведение программы не соответствует конфигурационным файлам. nginx может работать по-разному в зависимости от аргументов командной строки: начиная с того, чтобы запускаться с настройками не из конфигурационных файлов, и заканчивая тем, чтобы вообще не запускать веб-сервер, а завершить или перезапустить уже запущенный процесс `nginx`.

После того как вы узнали в теории и на практике, как иерархия конфигурации сказывается на программах, которые вы запускаете в Linux, и на программах, которые вы сами *пишете* для Linux, давайте проанализируем эту иерархию в деталях и разберем каждый ее уровень подробнее. Начнем с самого непосредственного и мощного способа конфигурации, который «заслоняет» все остальные и заключается в том, чтобы передать аргументы командной строки в тот момент, когда вы запускаете программу.

Аргументы командной строки

Вы уже знакомы с самым очевидным способом конфигурировать ПО — с аргументами командной строки. Они позволяют настраивать программу в тот момент, когда она запускается как команда оболочки.

Чтобы узнать, какие аргументы доступны для программы, обратитесь к справочной странице (`man`) соответствующей команды. За исключением совсем минималистичных сред, ПО для систем семейства Unix поставляется со справочными страницами, которые документируют большинство программ, разъясняют доступные флаги, а также — обычно в конце — перечисляют другие методы настройки, такие как конфигурационные файлы.

Рассмотрим начальный фрагмент справочной страницы для команды `find`:

```
FIND(1)                    Команды общего назначения                    FIND(1)

ИМЯ
     find – обходит иерархию файлов

ОБЗОР
     find [-H | -L | -P] [-EXdsx] [-f путь] путь ... [выражение]
     find [-H | -L | -P] [-EXdsx] -f путь [путь ...] [выражение]

ОПИСАНИЕ
     Утилита find рекурсивно спускается по дереву каталогов для каждого
     указанного пути и вычисляет выражение (составленное из "примитивов"
     и "операндов", которые перечислены ниже) относительно каждого файла
     в составе дерева.
```

```
Доступны следующие параметры:

-E      Интерпретировать регулярные выражения после примитивов -regex
        и -iregex как расширенные (современные) регулярные выражения,
        а не базовые выражения (BRE).  Оба формата исчерпывающе описаны
        на справочной странице re_format(7).

-H      Для каждой символической ссылки, указанной в командной строке,
        возвращать сведения о файле и его типе (см. stat(2)) для файла,
        на который ведет ссылка, а не для самой ссылки.  Если целевого
        файла не существует, возвращать сведения для самой ссылки.
        Информация о файлах для всех символических ссылок, не указанных
        в командной строке, соответствует самим ссылкам.
```

Как видите, бóльшая часть этой справочной страницы документирует различные аргументы командной строки, которые доступны при запуске `find`. Вы познакомились с аргументами командной строки еще в главе 1 этой книги и с тех пор использовали их много раз, так что все это должно выглядеть привычно.

Давайте перейдем к следующему, немного более опосредованному способу конфигурации — переменным окружения.

Переменные окружения

Аргументы командной строки — мощное средство, но они применяются только к тому единственному вызову команды, для которого заданы. Если запустить `ls -l`, то именно в этот конкретный раз команда `ls` выведет результаты в расширенном формате. Но что, если нужно, чтобы одна и та же настройка сохранялась для множественных вызовов команды? Например, это может пригодиться, если вы пишете сценарий, который будет устанавливать пакеты в разные моменты времени, и хотели бы настроить соответствующую конфигурацию *однократно*, вместо того чтобы повторно указывать ее в виде аргументов командной строки каждый раз, когда нужно установить пакет.

Поскольку вы разрабатываете программное обеспечение, вы, вероятнее всего, уже знакомы с переменными окружения: это значения, которые действуют в командной оболочке и аналогичны переменным в любом другом языке программирования. Они отличаются от аргументов командной строки тем, что применяются на более высоком уровне (и имеют меньший приоритет). Переменные окружения обеспечивают больше контроля, потому что если один раз присвоить значение конфигурационной переменной в оболочке, то оно будет применяться ко всем вызовам программы в текущем сеансе оболочки. Если задана переменная окружения, то программы, которые ее используют, будут применять ее значение при каждом запуске — до тех пор, пока оно не изменится или пока не завершится сеанс командной оболочки.

ПРИМЕЧАНИЕ

Мы подробнее поговорим о переменных окружения в главе 12 «Как автоматизировать задачи с помощью сценариев командной оболочки». Однако здесь мы рассмотрим базовые понятия этой темы.

В большинстве сред Unix переменные окружения позволяют задавать общие настройки, которые применяются ко многим различным программам, а не к какой-то одной. Например, в переменных окружения хранятся сведения о том, где находится домашний каталог текущего пользователя (`HOME`), каков текущий рабочий каталог (`PWD`), какую командную оболочку нужно использовать по умолчанию (`SHELL`), где искать исполняемые файлы, которые соответствуют командам, запущенным в оболочке (`PATH`), и т. д.

Чтобы просмотреть текущее значение той или иной переменной окружения, передайте ее имя в качестве аргумента команде `echo`[1], которая выводит значение в терминал:

```
$ echo $SHELL
/bin/zsh
```

Можно также просмотреть список всех действующих переменных окружения с помощью команды `env`[2]:

```
$ env
…
# много строк вывода, по одной для каждой переменной окружения
…
```

Чтобы настроить переменную окружения в своей оболочке, просто присвойте ей значение с помощью оператора `=` (убедитесь, что до и после этого знака нет пробелов)[3]:

---

[1] Слово *echo* буквально означает *эхо* и в ранних телекоммуникационных устройствах использовалось как метафора в ситуациях, когда устройство дублирует принятый или переданный сигнал в тот или иной поток вывода (обычно на терминал). В современных командных оболочках и языках программирования метафора размылась, и команды или функции с именем `echo` просто выводят те или иные данные; во многих языках аналогичные функции называются `print`, `printf`, `puts`, `log` и т. д. — *Примеч. науч. ред.*

[2] Сокр. *environment* (*окружение, среда*). — *Примеч. пер.*

[3] В последующих примерах `MYVAR` — сокр. *my variable* (*моя переменная*); это одно из стандартных имен переменных, которые используются для примеров в документации и технической литературе. — *Примеч. пер.*

```
MYVAR=beatles[1]
```

Эта переменная действует в текущей командной оболочке:

```
$ echo $MYVAR
beatles
```

Чтобы закрепить эту переменную для всех дочерних оболочек, которые вы порождаете (например, когда запускаете сценарий), используйте встроенную команду `export`:

```
export MYVAR=beatles
```

Подробнее эта тема рассматривается в главе 12 «Как автоматизировать задачи с помощью сценариев командной оболочки», однако вышеприведенной команды хватит для того, чтобы передавать конфигурацию переменных окружения в большинство программ, с которыми вы имеете дело.

Вернемся к примеру с `find`. Обратите внимание, что если прокрутить вниз справочную страницу, которую мы демонстрировали в предыдущем разделе, можно добраться до раздела `ENVIRONMENT (ОКРУЖЕНИЕ)`:

```
ОКРУЖЕНИЕ
     Переменные окружения LANG, LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES
и LC_TIME влияют на выполнение утилиты find, как описано в environ(7).
```

Это другой уровень конфигурации: здесь участвуют не аргументы командной строки, которые передаются при запуске программы, а конфигурационные директивы; их можно получить из переменных окружения командной оболочки.

Почему программа должна отдавать аргументам приоритет перед переменными окружения? Логика проста: аргумент командной строки (например, `-H`) чрезвычайно специфичен, потому что он определяется на уровне вызова команды, а значит, распространяется только на этот вызов.

А переменные окружения менее специфичны: они определяются на уровне командной оболочки и поэтому доступны всем командам, которые в ней запускаются.

Давайте продолжим путешествие по иерархии конфигурации: если значение не передается при вызове в качестве аргумента командной строки и не определяется переменной окружения в сеансе оболочки, в которой запущена программа, то откуда еще может поступить конфигурация?

---

[1] В современных дистрибутивах Linux переменным окружения технически можно присваивать практически любые текстовые значения, включая строки с кириллицей или даже эмодзи. Однако самые безопасные символы, как и на заре вычислительной техники, — латинские буквы, цифры и знак подчеркивания. — *Примеч. науч. ред.*

Конфигурационные файлы

Следующий источник, из которого программа получает свои настройки, — ее конфигурационные файлы. Они могут находиться в самых разных местах файловой системы, но здесь мы перечислим самые традиционные расположения.

Конфигурация системного уровня в /etc

Стоит начать с каталога `/etc`, с которым вы уже встречались в главе 5 «Файлы в Linux». Очень часто программное обеспечение хранит свою конфигурацию системного уровня в каталоге `/etc/`*`имя_программы`*. Для большинства программ этого хватает. Например, веб-сервер nginx — программа уровня системы: как правило, разные пользователи не запускают свои собственные экземпляры веб-сервера на одном и том же компьютере, поэтому конфигурации системного уровня более чем достаточно.

Впрочем, конфигурация крупного или сложного ПО иногда разбивается по отдельным файлам и каталогам внутри `/etc/`*`имя_программы`*. nginx — показательный пример такого рода: его основной конфигурационный файл — `/etc/nginx.conf`, но есть и дополнительные конфигурационные файлы, которые размещаются в каталоге `/etc/nginx/conf.d`.

Конфигурация пользовательского уровня в ~/.config

Если программа подразумевает расширенные настройки для отдельных пользователей, как это часто бывает, например, с текстовыми редакторами, инструментами разработки или играми, то для конфигурации обычно используется каталог `~/.config` внутри домашнего каталога пользователя. Вспомните материал главы 1 «Как устроена командная строка»: символ `~` обозначает домашний каталог текущего пользователя, а файлы и каталоги, имена которых начинаются с точки (`.`), не включаются в вывод команды `ls`, если не передать ей флаг `-a`. Каталог `~/.config` предписан стандартом XDG Base Directory[1], обзор которого находится по адресу wiki.archlinux.org/title/XDG_Base_Directory.

Например, конфигурация текстового редактора neovim существенно различается от разработчика к разработчику, однако единственный двоичный файл neovim в системе позволяет сотням программистов работать на одной машине одновременно, потому что для каждого из них neovim вызывается со своими собственными пользовательскими конфигурационными файлами, которые хранятся в `~/.config/nvim`. Это очень удобно!

---

[1] Досл. *Стандарт базовых каталогов XDG*. XGD (X Desktop Group) — прежнее название рабочей группы по стандартизации технических решений для сред рабочего стола; позже она трансформировалась в проект freedesktop.org. — *Примеч. пер.*

Только представьте, какой кавардак возник бы, если бы конфигурацию neovim можно было задавать только в каталоге `/etc` на уровне системы. Перед тем как запустить редактор, каждому разработчику приходилось бы настраивать кучу переменных окружения или запускать `nvim` с бесчисленными флагами командной строки.

После того как вы получили представление о классических источниках конфигурации ПО в системах семейства Unix, давайте посмотрим на специфическую возможность Linux: как работать с конфигурацией, которая поступает из файлов и аргументов командной строки для программ под управлением `systemd`.

Модули systemd

В большинстве дистрибутивов Linux, за исключением контейнеров Docker, всем заведует подсистема инициализации `systemd`. Мы уже изучали ее основы в главе 3 «Как управлять службами с помощью systemd», а здесь вкратце рассмотрим, как эта подсистема управляет конфигурацией ПО.

Если вы подзабыли главу 3, напомним главное: в среде Linux, где работает `systemd`, службы упаковываются в файлы модулей `systemd`, которые служат оберткой и механизмом управления для исполняемого двоичного файла, его аргументов и команд, с помощью которых модуль запускается, перезапускается и устанавливается, а также для многих других компонентов.

Как уже упоминалось, существует много типов модулей `systemd`, однако здесь мы рассмотрим тип `service`.

Мы также отмечали, что файлы модулей в зависимости от их назначения могут находиться в разных каталогах, хотя ваши собственные модули `systemd` обычно будут располагаться в `/etc/systemd/system`.

Чтобы разобраться, как модуль `systemd` позволяет влиять на иерархию конфигурации, о которой шла речь в этой главе, давайте создадим службу под управлением `systemd`, написав свой собственный модуль `systemd` для воображаемой программы. Назовем ее *`ваша_программа`*.

Как создать собственную службу

Вам как разработчику может понадобиться упаковать свое ПО в службу; ею удобнее управлять, чем программой, которая запускается в ручном (интерактивном) режиме. Это чрезвычайно эффективно само по себе, однако в этой главе мы углубимся в то, как модули `systemd` помогают лучше контролировать, как и откуда программа получает свою конфигурацию. Чтобы узнать, как создать службу, давайте упакуем двоичный файл в модуль `systemd`.

Прежде всего убедитесь, что нужный исполняемый файл скопирован в каталог `/usr/local/bin/`*`ваша_программа`*, который входит в стандартное значение переменной окружения `PATH`. Если вы хотите извлечь максимальную пользу из этого раздела, возьмите программу, которую вы откомпилировали вручную, например `htop` из предыдущей главы 9 «Как устанавливать программное обеспечение», и подставьте `htop` вместо *`ваша_программа`*.

Создайте модуль `systemd` — файл `/etc/systemd/system/`*`ваша_программа`*`.service` с таким содержанием:

```
[Unit]
Description=Описание вашей программы.
After=network-online.target

[Service]
Type=exec
ExecStart=/usr/local/bin/ваша_программа -clioption=1 -clioption2
EnvironmentFile=-/etc/ваша_программа/prod_defaults
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Вы заметили в этом файле модуля две строки, которые относятся к конфигурации?

Параметр `ExecStart` отвечает за то, как вызывается программа, когда кто-то запускает соответствующую службу `systemd`. В этом модуле указаны аргументы командной строки, которые будут передаваться программе, чтобы гарантировать, что при каждом запуске она будет выполняться с нужными параметрами. Всякий раз, когда кто-то запускает команду `systemctl start` *`ваша_программа`*, она будет вызвана с параметрами `-clioption=1` и `-clioption2`[1].

Кроме того, параметр `EnvironmentFile` задает путь к файлу, к которому будет обращаться `systemd`, чтобы назначить переменные окружения, относящиеся к вашей программе. Этот файл предназначен для того, чтобы его обрабатывала командная оболочка, с помощью которой `systemd` запускает двоичный код программы. Файл должен содержать значения, которые присваиваются переменным окружения, например:

```
# переменные окружения для ваша_программа
ENV=production
DB_HOST=localhost
DB_PORT=5432
```

[1] Чтобы программа запустилась, вместо `-clioption=1` и `-clioption2` нужно подставить конкретные параметры, которые она поддерживает. Например, для `htop` строка `ExecStart` может выглядеть так: `ExecStart=/usr/local/bin/htop --readonly -u=root`. — *Примеч. науч. ред.*

Запустите следующую команду, чтобы `systemd` заново прочитала свои конфигурационные файлы и обнаружила новую службу, которую вы создали:

```
$ sudo systemd daemon-reload
```

Теперь вашей программой можно управлять так же, как любой другой службой `systemd`:

- `systemctl start` *`ваша_программа`*
- `systemctl status` *`ваша_программа`*
- `systemctl stop` *`ваша_программа`*
- `systemctl enable` *`ваша_программа`*
- `systemctl disable` *`ваша_программа`*

Можете быть уверены, что каждый раз, когда вы будете запускать эту службу, она получит переменные окружения из вашего файла `/etc/`*`ваша_программа`*`/prod_defaults`, а параметры командной строки — из настройки `ExecStart`.

Мы продемонстрировали здесь предельно простую службу только для того, чтобы вы поняли, как `systemd` позволяет управлять конфигурацией программ. Однако существует гораздо больше директив конфигурации, которые можно передать в модуле. Если вам придется налаживать более серьезную службу, уделите немного времени тому, чтобы ознакомиться с документацией по модулям `systemd`: freedesktop.org/software/systemd/man/latest/systemd.unit.html#[Unit] Section Options.

Дополнительно: конфигурация в Docker

Ранее в этой главе мы обмолвились, что с Docker связаны исключения в том, что касается конфигурации. Контейнеры Docker — это крайне минималистичные среды, и поэтому в них нет огромного количества дополнительных исполняемых файлов, служб и конфигурационных файлов, которые бывают в традиционных системах Unix. Но поскольку в наше время значительная часть программного обеспечения выполняется в контейнерах, а не в классических средах с полноценным окружением операционной среды, мы осветим здесь основные моменты, чтобы вы представляли себе, чем отличается конфигурация в контейнерах Docker. Про контейнеры в целом мы подробнее поговорим в главе 15 «Контейнерная виртуализация приложений с помощью Docker».

В контейнерной среде, будь то Docker или другая система контейнерной виртуализации, мы имеем дело с крайне ограниченным окружением. В нем установлено очень мало программ и утилит, вместо `systemd` функционирует урезанная версия `init`, а в файловой системе нет многих каталогов, о которых мы упоминали.

Однако иерархия конфигурации опирается на те же принципы. Большинство контейнеризованных приложений получают свою конфигурацию из следующих источников:

- конфигурационный файл в файловой системе контейнера; этот файл часто создается планировщиком контейнера прямо перед тем, как контейнер запускается;
- переменные окружения, которые передает планировщик контейнера или оператор, который его запускает;
- аргументы командной строки.

Хотя это весьма упрощенная иерархия конфигурации, нельзя не заметить, что концептуально она устроена практически так же, как в полноценных системах Linux без контейнерной виртуализации.

Мы познакомимся с контейнерами подробнее в главе 15 «Контейнерная виртуализация приложений с помощью Docker».

Итоги

Из этой главы вы узнали об иерархии конфигурации Linux и о том, как она применяется к программам, которые вы регулярно используете и сами пишете. Мы рассмотрели аргументы командной строки, переменные окружения и все остальные источники, из которых программное обеспечение получает свою конфигурацию.

Если вы практиковались по ходу чтения этой главы, то вы даже создали службу `systemd`, которая выступает в качестве обертки для программы и позволяет управлять ее конфигурацией более единообразно.

11

Конвейеры и перенаправление ввода-вывода

Из этой главы вы узнаете, как приручить одну из самых мощных возможностей программирования — конвейеры! С их помощью можно соединять команды в цепочки и выстраивать сложные специализированные потоки данных, чтобы решить ту или иную задачу. К концу главы вы сможете не только понимать, но и самостоятельно составлять конвейеры вроде этого:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10
```

Если вам интересно, что делает этот код, расскажу: он выводит список из 10 самых часто используемых команд оболочки. На компьютере одного из авторов книги этот список выглядит так:

```
1000 git
 115 ls
 102 go
  83 gpo
  68 make
  65 cd
  59 docker
  42 vagrant
  35 GOOS=linux
  30 echo
```

Чтобы по-настоящему понять конвейеры, нужно освоить файловые дескрипторы и перенаправление ввода-вывода, так что мы начнем с этих тем. В этой главе много новой информации; не читайте ее в спешке, а постарайтесь воспроизвести на практике все примеры, чтобы убедиться, что вы во всем разобрались. Потратив немного времени на эту главу, вы сэкономите не один час на протяжении своей дальнейшей карьеры.

В этой главе рассматриваются такие темы:

- Что такое файловые дескрипторы.
- Как связывать команды в конвейеры с помощью символа |.
- Необходимые инструменты командной строки.
- Практические примеры конвейеров.
- Как анализировать файловые дескрипторы.

Файловые дескрипторы

Скорее всего, вы уже встречались с файловыми дескрипторами, когда разрабатывали программное обеспечение. Если нет, просмотрите главу 5 «Файлы в Linux». В двух словах, если ваша программа открывает файл в операционной системе для чтения или записи, вы получаете его дескриптор, то есть указатель, или ссылку, на соответствующий файловый объект.

Поскольку доступ к таким системным ресурсам, как файлы, осуществляется через механизмы операционной системы, она отслеживает, на какие файловые дескрипторы имеются активные ссылки в вашей программе.

Но даже если процесс не затрагивает ни одного файла в операционной системе, с ним все равно связано несколько открытых дескрипторов. В операционных системах семейства Unix у каждого процесса есть как минимум три файловых дескриптора:

Имя[1]	Описание	Файл устройства
`stdin`	Стандартный поток ввода	`/dev/fd/0`
`stdout`	Стандартный поток вывода	`/dev/fd/1`
`stderr`	Стандартный поток ошибок	`/dev/fd/2`

Эти три дескриптора служат стандартными каналами коммуникации, по которым процесс получает и передает данные, и существуют для каждого процесса, запущенного в системе. Дескриптор с индексом 0 всегда указывает на файл, из которого процесс получает входные данные, с индексом 1 — на файл, в который он записывает выходные данные, с индексом 2 — на файл, в который выводятся данные об ошибках.

[1] В этой таблице: `std` — сокр. *standard* (*стандартный*), `in` — здесь *ввод*, *входные данные*, `out` — здесь *вывод*, *выходные данные*, `err` — сокр. *error* (*ошибка*), `fd` — аббр. *file descriptor* (*файловый дескриптор*). — *Примеч. пер.*

Помимо этих трех стандартных файловых дескрипторов, программа может использовать любое количество других дескрипторов в зависимости от того, что она делает. Например, с процессом могут быть связаны:

- файлы, с которыми он работает;
- сокеты, из которых он получает или в которые передает данные (допустим, сокеты Unix или TCP, через которые данные передаются по сети);
- устройства, к которым обращается процесс, например клавиатуры или диски.

На что ссылаются файловые дескрипторы

Итак, с точки зрения процесса стандартные файловые дескрипторы предназначены для таких потоков данных:

Индекс	Имя	Назначение
0	`stdin`	Принимать входные данные из этого потока
1	`stdout`	Передавать обычные выходные данные в этот поток
2	`stderr`	Передавать данные об ошибках в этот поток

Но если абстрагироваться от конкретного процесса, то как узнать, на что именно указывают эти файловые дескрипторы? Откуда поступают входные данные и куда записываются выходные данные и ошибки?

Для примера рассмотрим процесс командной оболочки Bash. По умолчанию он принимает входные данные (`stdin`) от вашего терминала (который представляется в виде файла в файловой системе). В тот же терминал Bash выводит выходные данные и ошибки. По сути, весь сеанс оболочки состоит из операций чтения и записи в один и тот же файл. В следующей главе (глава 12 «Как автоматизировать задачи с помощью сценариев командной оболочки») вы узнаете о Bash гораздо больше.

Давайте подробнее рассмотрим такое перенаправление ввода и вывода.

Перенаправление ввода-вывода

Материал этого раздела пригодится во многих реальных задачах разработки: например, когда вы захотите, чтобы программа приняла входные данные из файла, чтобы не набирать их вручную, или когда понадобится заносить вывод программы в журнал, а также во многих других ситуациях. Когда создается процесс, можно управлять тем, куда указывают три его стандартных файловых дескриптора, и это приводит к впечатляющим результатам.

Перенаправление ввода: <

С помощью символа < («меньше») можно управлять тем, откуда процесс получает входные данные. Например, вы привыкли вводить данные в Bash с помощью клавиатуры, по одной команде за раз; но давайте сделаем так, чтобы Bash вместо этого приняла данные из файла.

Допустим, у нас есть файл `commands.txt` с таким содержимым (мы выводим его на экран с помощью команды `cat`):

```
#  cat commands.txt
pwd
echo "Всем привет! 👋"
echo $SHELL
cd /tmp
pwd
```

Все это допустимые команды оболочки (по крайней мере в Bash), так что запустим новый процесс Bash и укажем этот файл в качестве стандартного потока ввода:

```
# bash < commands.txt
/home/dave
Всем привет! 👋
/bin/bash
/tmp
```

Вместо того чтобы запрашивать у пользователя входные данные и ожидать, пока они поступят, Bash принимает по одной строке файла за раз и выполняет их последовательно: оболочка считывает данные из файла, пока не встретит перевод строки (символ `\n`). После этого она выполняет команду так, как если бы вы на этом месте нажали Enter.

В этом примере стандартный вывод программы по-прежнему направляется на терминал, где пользователь может его просмотреть. Давайте посмотрим, как это изменить.

Перенаправление вывода: >

Вот как перенаправить стандартный поток вывода (`stdout`, файловый дескриптор 1) не в терминал, а в файл, в результате чего вывод каждой команды не будет отображаться на терминале в режиме реального времени, а запишется в файл:

```
# bash < commands.txt > output.log
```

Обратите внимание, что теперь в терминал не выводятся видимые результаты, потому что символ > («больше») перенаправил вывод в файл `output.log`. С помощью `cat` можно вывести этот файл на экран и убедиться, что в нем оказались нужные выходные данные:

```
# cat output.log
/home/dave
Всем привет! 👌
/bin/bash
/tmp
```

Заметим, что, поскольку файловый дескриптор 1 — это стандартный вывод, символ > означает то же самое, что `1>`. Вы редко встретите вторую запись, потому что по умолчанию перенаправляется именно стандартный вывод. Однако две такие команды эквивалентны:

```
данные >  имя_файла
данные 1> имя_файла
```

Дополнение вывода без перезаписи: >>

В предыдущем примере мы создали файл отчета, перенаправив в него вывод команд с помощью >. Если запустить этот пример несколько раз, вы обнаружите, что файл не увеличивается: каждый раз, когда вы перенаправляете вывод в файл (`> имя_файла`), все прежнее содержимое файла заменяется на новое.

Чтобы избежать этого — например, если нужно, чтобы файл журнала накапливал вывод более чем от одного процесса или команды, — применяйте оператор дополнения >>. Он просто запишет данные в конец выходного файла, вместо того чтобы каждый раз переписывать его содержимое.

В следующей главе мы более подробно разберем сценарии Bash, но сейчас продемонстрируем элементарный сценарий, который ежесекундно записывает в журнал текущую метку даты и времени:

```
while true; do
    date >> /tmp/date.log
    sleep 1
done
```

Этот сценарий запускает бесконечный цикл `while true; do [ … ] done`, в котором выполняется команда `date`[1] (она выводит текущую метку времени). Вывод команды перенаправляется в файл `/tmp/date.log` с помощью оператора >>, который добавляет новые данные в конец файла (в отличие от >, который каждый раз перезаписывал бы файл). Затем сценарий приостанавливается на 1 секунду с помощью команды `sleep`[2], и цикл начинается заново.

Если просто однократно запустить команду `date`, она выведет текущую дату и время:

---

[1] *Дата. — Примеч. пер.*

[2] *Спать. — Примеч. пер.*

```
→ ~ date
Sat Jan 6 16:39:37 EST 2024
```

Однако если запустить наш сценарий, мы получим другой результат. Сначала в терминале ничего не будет отображаться, потому что вывод перенаправляется в файл. Вот что происходит, если вставить сценарий в командную строку, ненадолго запустить его, принудительно завершить сочетанием клавиш Ctrl+C, а затем вывести содержимое созданного файла:

```
→ ~ while true; do \[1]
date >> /tmp/date.log \
sleep 1 \
done
^C

→ ~ cat /tmp/date.log
Sat Jan 6 16:44:01 EST 2024
Sat Jan 6 16:44:02 EST 2024
Sat Jan 6 16:44:03 EST 2024
# … и т. д. …
Sat Jan 6 16:44:08 EST 2024
```

Такое перенаправление вывода будет весьма полезно в повседневной работе, например, когда нужно создать незапланированный файл журнала для отладочного сценария, который вы изготовили на скорую руку.

Перенаправление ошибок: 2>

Многие программы с интерфейсом командной строки, которые интенсивно выводят данные, также время от времени порождают ошибки: например, команда `find` может сталкиваться с эпизодическими ошибками доступа для каталогов, если у нее не хватает прав, чтобы их просматривать.

Хотя подобные ошибки незначительны и ожидаемы, не хочется засорять ими обычный поток вывода. Это особенно важно, если вы не запускаете инструменты командной строки в интерактивном режиме, а пишете небольшие сценарии или крупные программы, которые обрабатывают вывод команд.

Вы уже знаете, как перенаправлять стандартные потоки ввода и вывода (файловые дескрипторы с индексами 0 и 1 соответственно). Конструкция `2>` позволяет перенаправлять стандартный поток ошибок (файловый дескриптор с индексом 2):

```
find /etc/ -name php.ini > /tmp/phpinis.log 2>/dev/null
```

[1] Чтобы в командной оболочке ввести команду из нескольких строк, можно разделять строки обратным слешем (\). — *Примеч. науч. ред.*

Эта команда ищет любые файлы с именем `php.ini` в поддереве каталога `/etc`. Имена обнаруженных файлов (стандартный поток вывода команды `find`) записываются в файл `/tmp/phpinis.log`, а любые ошибки игнорируются благодаря тому, что команда перенаправляет их в специальный файл `/dev/null`.

СОВЕТ

`/dev/null` — это специальный объект, подобный файлу, который возвращает метку конца файла, если читать из него, и отбрасывает все, что в него записывается. Его используют как «мусорный бак» для вывода, который нужно подавить или игнорировать. Этот объект часто встречается в сценариях.

После того как вы узнали, как перенаправлять ввод и вывод, давайте посмотрим на конвейеры, которые объединяют эти операции: они перенаправляют вывод одной команды на вход другой.

Как объединять команды с помощью конвейеров (|)

Вы уже умеете перенаправлять каждый из трех стандартных файловых дескрипторов в разные местоположения и знаете, когда это бывает полезно. Но что, если нужно не перенаправлять ввод и вывод в те или иные файлы, а связать между собой *несколько программ*?

Чтобы связать вывод одной программы с вводом другой, в командной строке можно использовать вертикальную черту (|)[1]. Это чрезвычайно мощный принцип, с помощью которого в Unix и Linux создаются пользовательские команды для того, чтобы сортировать, фильтровать и перерабатывать данные:

```
echo -e "какой-то текст \n что-то интересное \n еще текст" | grep интересное
```

Если запустить эту команду в оболочке, она выведет `что-то интересное`. Вот что здесь происходит:

1. Запускается первая команда `echo`, которая должна вывести текст, заключенный в кавычки (из-за символов `\n` он представляет собой три строки текста).

---

[1] Канал межпроцессной коммуникации (как и системный вызов для его создания) исторически называется *pipe* (досл. *труба*, *трубка*), а структура из процессов, которые связаны по своим стандартным потокам данных, — *pipeline* (досл. *трубопровод*). Сам символ | англоязычные специалисты тоже часто называют *pipe*. В русскоязычной компьютерной литературе *pipeline* в этом значении принято переводить как *конвейер*, хотя в неформальной речи часто используется калька с английского термина. — *Примеч. пер.*

2. Символ | перенаправляет этот вывод (файловый дескриптор 1) на вход (файловый дескриптор 0) следующей команды `grep`. Теперь ее вход связан с выходом предыдущей команды.
3. Команда `grep` просматривает каждую строку данных по очереди, находит совпадение со словом `интересное` во второй строке и выводит всю эту строку в стандартный вывод.

Многоступенчатые конвейеры

Вот гораздо более интересный пример, похожий на тот, который вы уже видели в начале этой главы:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10 > \
/tmp/top10commands
```

Каждый оператор | в этой цепочке просто берет вывод (`stdout`) предыдущей команды и передает его на вход (`stdin`) следующей команды.

Благодаря тому, что вывод одной команды перенаправляется на вход другой, можно составлять сложные потоки данных, где данные фильтруются и сортируются, передаваясь от одних команд к другим, — и для этого не приходится писать специальное программное обеспечение. В Linux нет готовой команды, которая выводит 10 самых частых команд, но ее всегда можно быстро сконструировать из существующих стандартных команд, как в примере выше.

Как читать (и составлять) сложные многоступенчатые конвейерные команды

Какими бы сложными или эзотерическими ни казались те или иные команды с конвейерами, которые вам встретятся, все они строятся по одному и тому же принципу: команды выполняются по цепочке, по одной команде за раз. Независимо от того, читаете ли вы сложную конвейерную команду или сами ее составляете, это делается единообразно:

1. Проанализируйте первую команду и убедитесь, что хотя бы в общих чертах понимаете, что она делает. Если команда вам незнакома, обратитесь к справочным страницам или другой документации.
2. Запустите команду и изучите, что она выводит.
3. Добавьте символ | и следующую команду в конвейере.
4. Повторяйте процедуру с шага 1 для каждой новой команды, пока не доберетесь до конца конвейера.

Благодаря этой процедуре вы увидите, как даже самые монструозные конвейерные конструкции становятся вполне управляемыми. Не забывайте, что вы имеете дело просто с потоком данных, который перетекает, как по трубам, с одной ступени

конвейера на другую и проходит через разные команды, которые его модифицируют, перестраивают, фильтруют и перенаправляют.

Сделаем небольшое отступление на тему, о которой мы подробнее поговорим в главе 12 «Как автоматизировать задачи с помощью сценариев командной оболочки»: проявляйте уважение к другим программистам, которые будут читать ваш код. Постарайтесь не использовать в выражении больше двух-трех символов | и присваивайте осмысленные имена переменным, в которых хранятся промежуточные результаты (если это позволяют ваши ограничения по памяти).

Итак, вы узнали, как примитивы файловых дескрипторов можно задействовать для того, чтобы удобно перенаправлять ввод-вывод. Давайте теперь рассмотрим несколько примеров полезных комбинаций программ, которые опираются на эти встроенные возможности Unix.

Полезные инструменты командной строки

Прежде чем мы перейдем к мудреным цепочкам команд наподобие той, с которой началась эта глава, давайте перечислим несколько популярных вспомогательных средств Unix для того, чтобы фильтровать, сортировать и объединять потоки данных, которые вы создаете в командной строке.

cut — разбить строку на части

Команда `cut`[1] принимает разделитель (после флага `-d`) и разбивает по нему входные данные подобно командам `String.Split()` или `String.Fields()`, существующим во многих языках программирования. Затем с помощью флага `-f` можно выбрать, какое поле (элемент списка) вывести: например, если вам нужно четвертое поле, то запустите `cut` с флагом `-f4`:

```
# echo "Это слова, разделенные пробелами" | cut -d " " -f4
пробелами
```

Если передать команде `cut` более одной входной строки, она разобьет каждую строку по заданным разделителям.

Разделитель не обязательно должен быть пробелом. В следующем примере мы разбиваем строку по запятым:

```
# echo "Это слова, разделенные пробелами" | cut -d "," -f1
Это слова

# echo "Это слова, разделенные пробелами" | cut -d "," -f2
 разделенные пробелами
```

[1] Здесь *резать*, *разреза́ть*. — *Примеч. пер.*

Количество полей зависит от того, по каким символам разбивается строка: например, в этом случае получилось два поля, потому что в тексте всего одна запятая. Если попробовать вывести четвертое поле с флагом `-f4`, как в предыдущем примере, вы получите пустую строку.

Вот как можно вывести «человеческие» имена всех пользователей macOS, в записи которых в файле `/etc/passwd` есть подстрока `root`:

```
# grep root /etc/passwd | cut -d ":" -f5
System Administrator
System Services
CVMS Root
```

sort — отсортировать строки

Команда `sort`[1] сортирует строки в алфавитном или числовом порядке.

Сортировка в обратном порядке (флаг `-r`[2]) часто полезна, когда вы работаете с числовыми данными (флаг `-n`[3]). Во многих случаях имеет смысл комбинация флагов `-rn` (см. «Как вывести «топ-X» (рейтинг с подсчетом)» в разделе «Практические идиомы конвейеров» далее в этой главе).

Флаг `-h`[4] может пригодиться, чтобы сортировать «человекочитаемый» вывод многих других команд, например:

```
# du[5] -h | sort -rh
1.6M .
1.3M ./.git
1.2M ./.git/objects
60K  ./.git/hooks
28K  ./.git/objects/d8
```

uniq — обработать повторяющиеся строки

Команда `uniq`[6] удаляет дубликаты строк. Чтобы она работала так, как вы обычно ожидаете, ей нужно передавать отсортированные данные; иначе команда удалит

---

[1] *Сортировать. — Примеч. пер.*

[2] Длинная форма — `--reverse` (*в обратном порядке*). — *Примеч. пер.*

[3] Длинная форма — `--numeric-sort` (*числовая сортировка*). — *Примеч. пер.*

[4] Длинная форма — `--human-numeric-sort` (*числовая сортировка в человекочитаемом формате*). — *Примеч. пер.*

[5] Команда `du` (аббр. *disk usage — использование диска*) оценивает, сколько дискового пространства занимают файлы и каталоги. — *Примеч. науч. ред.*

[6] Сокр. от *unique* (*уникальный*). — *Примеч. пер.*

только те дубликаты, которые следуют непосредственно друг за другом. Рассмотрим такой файл:

```
# cat /tmp/uniq
один
два
один
один
один
семь
один
```

Вот как команда `uniq` работает по умолчанию; скорее всего, это не то, что вам нужно:

```
# uniq /tmp/uniq
один
два
один
семь
один
```

Команда отбросила повторяющиеся строки, которые шли непосредственно друг за другом, но оставила дубликаты, которые разделены другими текстовыми данными. А теперь запустим `uniq` для отсортированных данных:

```
# sort /tmp/uniq | uniq
два
один
семь
```

Подсчет элементов

Команда `uniq` умеет подсчитывать элементы — эта возможность доступна с флагом `-c`[1]. При этом остается в силе предостережение об отсортированном вводе; рассмотрим, например, файл с таким содержимым:

```
arch
alpine
arch
arch
```

Если запустить для этих данных `uniq -c`, получится такой результат:

```
$ uniq -c /tmp/sort1.txt
      1 arch
      1 alpine
      2 arch
```

[1] Длинная форма — `--count` (*подсчитывать*). — *Примеч. пер.*

Это не то, чего ожидает большинство пользователей. В файле три вхождения `arch`, но `uniq` показывает для этого слова два отдельных счетчика. Чтобы строки не дублировались, входные данные нужно отсортировать.

Это раздражает начинающих пользователей, но хорошо согласуется с принципом Unix о том, чтобы инструменты были узкоспециализированными, но острозаточенными и не дублировали функции друг друга. Если вы разрабатываете инструмент для сортировки, он должен только сортировать данные, а если вы разрабатываете инструмент, который устраняет дубликаты, пусть он не занимается сортировкой, а полагается в этом вопросе на предыдущий инструмент, чтобы максимально экономить память.

Если отсортировать строки до того, как передавать их команде `uniq`, получится нужный результат:

```
$ sort /tmp/sort1.txt | uniq -c
      1 alpine
      3 arch
```

Вы наверняка обратили внимание, что строки отсортированы по возрастанию. Это не то, что нужно, если мы хотим получить список самых частых команд вроде того, что приведен в начале этой главы. Для такого списка понадобится отсортировать строки по убыванию (`-rn`); это легко сделать, потому что благодаря `uniq -c` каждая строка начинается с числа. Вот пример соответствующего конвейера для файла, где еще больше дубликатов:

```
$ sort /tmp/sortme.txt | uniq -c | sort -rn
      6 ubuntu
      4 alpine
      3 gentoo
      2 yellow dog
      2 arch
      1 suse
      1 mandrake
```

wc — подсчитать текстовые компоненты

Команда `wc`[1] подсчитывает количество слов, строк, символов и байтов во входных данных. Например, с ключом `-w`[2] она выводит количество слов, разделенных пробелами:

```
# echo "foo bar baz" | wc -w
3
```

---

[1] Сокр. *word count* (*количество слов*). Обратите внимание, что, несмотря на название, команда `wc` подсчитывает не только слова, но и другие текстовые компоненты. — *Примеч. пер.*

[2] Длинная форма — `--words` (*слова*). — *Примеч. пер.*

А такая крайне популярная идиома позволяет узнать количество строк в файле:

```
# wc -l[1] < /etc/passwd
123
```

head — вывести первые строки потока данных

Команда `head`[2] возвращает несколько первых строк потока данных или файла. По умолчанию выводится 10 строк, но их количество можно регулировать с помощью флага `-n`[3]:

```
# head -n 2 /etc/passwd
##
# User Database
```

tail — вывести последние строки потока данных

Команда `tail` противоположна `head`: она выводит последние строки потока данных или файла, и количество строк также регулируется флагом `-n`.

`tail` с флагом `-f` можно использовать в интерактивном режиме, чтобы отслеживать обновления в файле журнала в реальном времени. Такой код вам не раз пригодится при устранении неполадок:

```
tail -f /var/log/ngnix/access.log
```

tee — копировать стандартный ввод в стандартный вывод и файл

Иногда одной копии данных из стандартного ввода недостаточно. Команда `tee`[4] копирует стандартный ввод (`stdin`) в стандартный вывод (`stdout`), а также в файл. Нам как разработчикам `tee` особенно полезен в двух случаях.

Первый случай — отладка и протоколирование: когда мы запускаем сценарии или программы, которые порождают вывод, `tee` позволяет выводить его на экран и параллельно сохранять в файле, чтобы анализировать впоследствии. В следующем примере мы вызываем команду `echo`, хотя вы, скорее всего, укажете перед вертикальной чертой свою собственную программу:

---

[1] Длинная форма — `--lines` (*строки*). — *Примеч. пер.*

[2] Досл. *голова*, перен. *начало*, *начальная часть*. — *Примеч. пер.*

[3] Сокр. *number* (здесь *количество*); длинная форма — `--lines`. — *Примеч. пер.*

[4] Название команды `tee` (здесь *тройник*) происходит от названия одноименной соединительной детали трубопровода. — *Примеч. пер.*

```
# echo "Привет! 👋" | tee /tmp/greetings.txt
Привет! 👋
# cat /tmp/greetings.txt
Привет! 👋
```

Еще команда `tee` полезна для того, чтобы копировать данные из конвейеров вроде тех, которые мы научились конструировать в этой главе. `tee` можно вставить в любую точку потока данных, чтобы сохранить и/или проанализировать промежуточные результаты, не нарушая работу конвейера.

Раньше мы уже рассматривали конвейер, который выводит 10 самых частых команд, но теперь давайте добавим в него `tee`, перед тем как ограничивать количество результатов до 10. В результате конвейер сохранит полный список результатов во временный файл до того, как отсечет верхнюю часть:

```
history[1] | awk '{print $2}' | sort | uniq -c | sort -rn \
| tee /tmp/all_commands.txt | head -n 10
```

Если теперь понадобится просмотреть все команды, а не только первые 10, можно запустить `cat` или `less`, чтобы открыть файл `/tmp/all_commands.txt`.

awk — расширенная обработка входного потока

С помощью команды `awk` удобно работать со столбцами данных, хотя на самом деле AWK[2] — это целый язык для построчного обзора и обработки входного потока. Чаще всего его используют, чтобы заменять символы в потоках файлов.

Например, вот как можно извлечь второе поле из каждой строки входных данных:

```
echo "два слова" | awk '{print $2}'
слова
```

sed

`sed` — это потоковый редактор с богатым набором возможностей. Чаще всего он служит для того, чтобы заменять последовательности символов в потоках файлов.

---

[1] Команда `history` позволяет просматривать историю команд и выполнять различные операции с ней. — *Примеч. науч. ред.*

[2] Язык AWK разрабатывался в Лабораториях Белла (AT&T Bell Laboratories) в 1970-х годах; его название составлено по первым буквам фамилий разработчиков: Альфреда Ахо (*Alfred Aho*), Питера Уайнбергера (*Peter J. Weinberger*) и Брайана Кернигана (*Brian Kernighan*). Все эти люди — прославленные деятели компьютерной отрасли, которые стояли у истоков Unix и других классических технологий и на момент подготовки этого издания продолжают вносить заметный вклад в развитие цифровой индустрии. — *Примеч. науч. ред.*

Рассмотрим такой файл:

```
# cat /tmp/sensitive.txt
У меня секретов нет
это не секретная информация
птица-секретарь
секрет
кажется, это работает 😳
```

Если нужно отредактировать только ту строку, которая состоит из одного слова `секрет`, и больше ничего не менять, можно запустить такую команду:

```
sed 's/^секрет$/— КОНФИДЕНЦИАЛЬНО —/' /tmp/sensitive.txt
У меня секретов нет
это не секретная информация
птица-секретарь
— КОНФИДЕНЦИАЛЬНО —
кажется, это работает 😳
```

В этом примере мы использовали файл, а не входной поток из другой команды. По умолчанию `sed` не изменяет исходный файл; если вам нужно его изменить, добавьте флаг `-i`[1].

К этому моменту вы получили представление о конвейерах и познакомились с наиболее распространенными инструментами командной строки, так что давайте объединим всё, что вы узнали, и изучим несколько популярных шаблонов конвейерной обработки, которые наверняка пригодятся вам в профессиональной деятельности.

Практические идиомы конвейеров

Как уже упоминалось, длинные команды с многоступенчатыми конвейерами строятся итеративно, по одной команде за раз. Тем не менее существует несколько полезных идиоматических конструкций, которые будут вам часто встречаться на практике.

Как вывести «топ-X» (рейтинг с подсчетом)

В этой схеме входные данные сортируются по количеству вхождений в порядке убывания. Вы уже видели такой пример ранее в этой главе, где выводился список самых часто используемых команд оболочки из истории Bash.

Идиома выглядит так:

```
входные_данные | sort | uniq -c | sort -rn | head -n X
```

[1] Длинная форма — `--in-place` (*на месте*). — *Примеч. пер.*

Рассмотрим более подробно, как работает этот конвейер.

- Команда `sort` сортирует входные данные по алфавиту, чтобы их можно было эффективно передать команде `uniq -c` (мы уже говорили об этом ранее).
- Команда `uniq -c` отбрасывает дубликаты, но агрегирует в каждой строке количество ее дубликатов.
- Снова запускается `sort`, но на сей раз она сортирует строки по убыванию количества дубликатов (с флагами `-r` и `-n`), в результате чего в начале списка оказываются строки, у которых больше всего дубликатов.
- Команда `head` берет этот рейтинг и сокращает его до *X* первых строк. Например, с флагом `-n 3` мы получим три самые часто встречающиеся строки из первоначальных входных данных вместе с количеством вхождений каждой из них.

Эта идиома полезна, когда нужны сведения о лидерах того или иного рейтинга, то есть отсортированный и отранжированный список, — например, если вас интересует, какими веб-браузерами чаще всего пользуются посетители вашего сайта или каковы IP-адреса самых активных злоумышленников, которые пытаются найти в нем уязвимости.

Как устанавливать ПО с помощью curl | bash

Идиома `curl | bash` — это традиционный механизм для того, чтобы в Linux загрузить и тут же выполнить сценарий из интернета. Здесь совмещаются два мощных инструмента командной строки — команда `curl`, которая загружает содержимое по заданному URL, и интерпретатор командной оболочки `bash`, который выполняет загруженный код. Этот подход существенно экономит время, позволяя разработчикам быстро разворачивать приложения или запускать сценарии без необходимости сначала вручную загружать их, а потом отдельно передавать оболочке Bash для выполнения.

Вот как в качестве примера можно установить DNS-сервер Pi-hole[1], который блокирует рекламу:

```
curl -sSL https://install.pi-hole.net | bash
```

Разберем отдельные компоненты этого конвейера:

---

[1] Официальный файл `README` проекта Pi-hole (github.com/pi-hole/pi-hole#alternative-install-methods) предостерегает от того, чтобы просто запускать его по схеме `curl | bash`, и предлагает сначала ознакомиться с установочным сценарием и убедиться, что он безопасен, а потом выполнить его одним из альтернативных способов. — *Примеч. науч. ред.*

1. Первая команда `curl -sSL https://install.pi-hole.net` (до вертикальной черты) загружает установочный сценарий Pi-hole, который доступен по указанному адресу. Команда запускается с флагами [1]:

`-s`	«Беззвучный» режим, в котором в стандартный поток вывода поступает необработанный ответ от сервера без посторонних сообщений
`-S`	Когда этот флаг используется совместно с `-c`, `curl` выводит сообщение об ошибке, если не удается получить запрошенный ресурс
`-L`	Обрабатывает перенаправления протокола HTTP

2. Оператор `|` перенаправляет вывод предыдущей команды (`curl`) на вход следующей (`bash`).
3. Команда `bash` выполняет сценарий, который загрузила команда `curl`.

Эта схема помогает автоматизировать операции, например, чтобы развертывать код или настраивать локальную среду. Однако прежде чем запускать сценарий, который вы загрузили, убедитесь, что он безопасен: бездумно запускать сценарии из интернета — очень плохая идея.

Замечания о безопасности идиомы curl | sudo bash

Всякий раз, когда вы разрешаете стороннему коду запускаться на своем компьютере, вы в той или иной степени жертвуете безопасностью ради удобства. При этом, если вы устанавливаете программное обеспечение с помощью `curl | sudo bash`, запуская установочный сценарий с надежного сервера, которому вы доверяете, это мало чем отличается от установки через систему управления пакетами. Большинство этих систем (кроме разве что `nix`) не могут похвастаться высшим уровнем защиты, но они хотя бы обеспечивают разумные меры безопасности. Если вы запускаете установочный сценарий с помощью `curl | sudo bash`, вы лишаетесь этих мер:

- Не существует пакета, для которого можно было бы проверить контрольную сумму и криптографическую подпись, чтобы убедиться, что вам досталась правильная и официальная версия ПО.
- Выбор серверов, с которых вы загружаете сценарии, ничем не регулируется, и вы не знаете, насколько можно доверять этим серверам. Нет гарантий, что вы не обращаетесь к скомпрометированному серверу, который распространяет вредоносные установочные сценарии.
- Сами по себе сценарии — это всего лишь код, который запускается на вашем компьютере от имени `root` и может делать в системе *все то же, что и вы*, — будь

[1] Длинные версии флагов: `-s`: `--silent` (*тихий, беззвучный*), `-S`: `--show-error` (*отображать ошибки*), `-L`: `--location` (*местоположение*). — *Примеч. пер.*

то полезные или вредоносные операции. Впрочем, справедливости ради заметим, что многие популярные системы управления пакетами страдают от этой же проблемы.

С учетом этих соображений, пожалуйста, прислушайтесь к нашему совету: выделяйте `curl` в самостоятельную операцию и читайте установочный сценарий, который вы загрузили, прежде чем запускать его с помощью `sudo bash`. Вот на что стоит обратить внимание в первую очередь:

- Убедитесь, что вы доверяете серверу (домену), с которого загружаете сценарий: например, это сайт разработчика с хорошей репутацией или заведомо надежное стороннее хранилище кода.
- Проверьте, что `curl` загружает код по протоколу HTTPS (то есть URL начинается с `https://`).
- Внимательно прочитайте сценарий, чтобы выяснить, какие команды он запускает и откуда берет дополнительный код или исполняемые файлы. Если сценарий загружает такие ресурсы из интернета, проверьте их тоже.

Надеемся, теперь вы убедились, что `curl | sudo bash` — не самый безопасный метод установки ПО. Мы постарались дать рекомендации, которые помогут уберечься от неприятностей, если вы (как и большинство разработчиков) однажды поддадитесь искушению установить таким «ленивым» способом ту или иную программу, например `homebrew` в macOS.

Как фильтровать и искать данные с помощью grep

Если вы запускаете команды, которые выводят много данных, обычно имеет смысл фильтровать этот вывод, чтобы оставить только нужные данные. Чаще всего это делается с помощью `grep`, и эту команду можно рассматривать как функцию текстового поиска с богатыми возможностями конфигурации.

Чтобы продемонстрировать пример фильтрации, допустим, что нам нужно найти рабочий каталог определенного процесса в Linux. Для этого запустим `lsof`:

```
→ ~ lsof -p 3243 | grep cwd
vagrant  3243  dcohen  cwd  DIR  1,4  192  51689680
/Users/dcohen/code/my_vagrant_testenv
```

Вот что здесь происходит:

1. Команда `lsof` выводит список открытых файловых дескрипторов для заданного процесса (PID 3243).
2. Затем мы передаем этот список по конвейеру (`|`) утилите `grep`, которая ищет в нем подстроку `cwd`. Эта подстрока находится только в одной строке списка, так что `grep` выводит именно эту строку в терминал.

Эта идиома полезна всегда, когда у вас есть крупный массив данных, но вам нужна только небольшая часть этого массива, которую можно идентифицировать по определенной строке. Команда `grep` применяется к строкам текста из стандартного ввода, поэтому она особенно эффективна, чтобы найти данные, подобные перечисленным:

- журнальные сообщения с IP-адресом, который вы отслеживаете;
- вхождения определенного имени пользователя в потоке данных;
- строки, которые соответствуют регулярному выражению (`grep` поддерживает регулярные выражения и может принимать не только строковые литералы, но и шаблоны поиска).

`grep` — сложный и мощный инструмент, которым вы наверняка будете пользоваться почти каждый день. Чтобы познакомиться с ним подробнее, обратитесь к соответствующей справочной странице (`man grep`).

В этой книге вам уже встречался поиск по файлам с помощью `grep` (например, в командах вроде `grep tutorialinux /etc/group`), однако эта команда становится по-настоящему незаменимой в составе конвейеров. Давайте рассмотрим практический пример.

Как анализировать журналы с помощью grep и tail

Когда вы изучаете журналы продакшен-среды, чтобы понять, в чем проблема, вам часто требуется просмотреть только те записи, в которых встречаются определенные ключевые слова или совпадения с определенным шаблоном. Здесь поможет такая идиома:

```
tail -f /var/log/webapp/too_many_logs.log | grep "регулярное_выражение"
```

Этот конвейер непрерывно опрашивает файл журнала на предмет новых сообщений, которые соответствуют регулярному выражению, и отображает только те из них, что могут относиться к текущей задаче.

Как обрабатывать множество файлов с помощью find и xargs

`xargs`[1] — мощная утилита, которая позволяет выполнять перебор (другими словами, цикл типа `for`) в рамках одной команды. По умолчанию `xargs` перебирает элементы входного потока (разделенные пробелами, переводами строк, символами табуляции или конца файла) и выполняет указанную программу для каждого элемента. Например, если нужно найти то или иное ключевое слово

[1] Сокр. *extended arguments* (*расширенные аргументы*). — *Примеч. пер.*

только в файлах, которые вернул определенный запрос `find`, можно запустить такую команду:

```
find . -type f -name "*\.txt" | xargs grep "ключевое_слово"
```

Эта команда находит все файлы, имена которых оканчиваются на `.txt`, а затем благодаря `xargs` применяет `grep` к каждому файлу по отдельности. Такая идиома удобна для того, чтобы «обыскать» или модифицировать сразу несколько файлов. Обратите внимание, что `xargs` — не только мощная, но и *крупная* программа, способная выполнять много задач (вплоть до интерполяции строк в подчиненной команде). В этой книге не хватит места, чтобы рассмотреть все ее возможности, поэтому если вы попали в ситуацию, когда именно такая функциональность может спасти положение, стоит прочесть соответствующую справочную страницу (`man xargs`) и поискать в интернете подходящие примеры.

Как анализировать данные с помощью sort, uniq и числовой сортировки по убыванию

С этой полезной идиомой вы познакомились на практике в начале главы, где она помогала отфильтровать длинную историю команд, чтобы получить список из нескольких самых популярных, которые запускались в системе. Общая схема выглядит так:

```
входные_данные | sort | uniq -c | sort -rn
```

Такой конвейер полезен для анализа данных: он сортирует входные данные, подсчитывает количество дубликатов и отфильтровывает их, а затем снова сортирует результат, чтобы получить рейтинг, где самые часто встречающиеся элементы находятся сверху.

К этой идиоме часто добавляют `| head -n X`, чтобы вывести только *X* первых результатов, например:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10
```

В этом примере еще добавилась команда `awk`, потому что без нее `history` возвращала бы полные команды со всеми аргументами, например:

```
12  apt install htop
```

Нас интересуют только названия команд (в данном случае `apt`), и `awk` позволяет извлечь из полного сообщения только второе поле.

Затем список команд сортируется по алфавиту, чтобы все дубликаты каждой команды следовали друг за другом. На следующем шаге `uniq` удаляет дубликаты и добавляет их счетчик перед каждым оставшимся уникальным значением. После

этого мы снова сортируем список — теперь по убыванию счетчиков (с флагами `-rn`), что дает нам рейтинг популярности команд. Наконец, `head` ограничивает его до 10 первых элементов.

В результате выводится список из 10 самых часто используемых команд оболочки. Мы уже демонстрировали, как он выглядит на одном из наших компьютеров:

```
1000 git
 115 ls
 102 go
  83 gpo
  68 make
  65 cd
  59 docker
  42 vagrant
  35 GOOS=linux
  30 echo
```

Как форматировать данные и обрабатывать поля записей с помощью awk и sort

`awk` — больше чем программа; это язык потоковой обработки. Если вы работаете с потоками данных в системе Unix, то имеет смысл потратить несколько дней на то, чтобы освоить основы этого языка, ведь это сэкономит вам недели и месяцы в дальнейшей профессиональной практике. Можно начать с того, чтобы научиться с помощью подстановок вида `$n` обращаться к столбцам, разделенным пробелами, в каждой строке потока данных.

Для примера рассмотрим такой поток данных:

```
Foo bar baz
Эта шляпа мне как раз
```

Подстановку `$1` интерпретатор `awk` понимает как первый столбец данных — в данном случае `Foo` в первой строке и `Эта` во второй. Подстановка `$2` обозначает второй столбец (`bar`, `шляпа`), и т. д. Этот механизм очень удобен для работы с данными, с которыми простой `cut` может не справиться, например:

```
cat file.txt | awk '{print $2, $1}'
```

В результате получатся такие строки:

```
bar Foo
шляпа Эта
```

Из каждой строки этот конвейер выводит сначала второй столбец, а затем первый, игнорируя остальные. Такая идиома помогает форматировать и организовывать данные в зависимости от структуры полей.

Как редактировать файлы и создавать их резервные копии с помощью sed и tee

Потоковый редактор `sed`[1] полезен, когда нужно преобразовать поток данных. Именно этим вы занимаетесь по десять раз на дню в своем текстовом редакторе, когда выполняете поиск и замену символов. Следующая команда фактически реализует эту функциональность для командной строки: она заменяет все вхождения слова `старое` в файле `file.txt` на `новое` и записывает результаты в новый файл `file.txt.changed`. При этом исходный файл `file.txt` не меняется:

```
sed 's/старое/новое/g' file.txt | tee file.txt.changed
```

Хотя эту идиому удобно продемонстрировать на примере редактирования файла, `sed` еще лучше подходит для того, чтобы преобразовывать потоковые данные, которые получены на выходе предыдущей команды, и передавать их на вход следующей:

```
входные_данные | sed 's/old/new/g' | следующая_команда
```

Как управлять процессами с помощью p s, grep, awk, xargs и kill

Хотя утилита `pgrep` позволяет удобно передавать сигналы всем процессам, имена которых соответствуют определенному шаблону, она не всегда доступна в системе. Следующая идиома позволяет добиться похожей функциональности, причем с более широким выбором полей помимо имени процесса. Общая схема выглядит так:

```
ps aux | grep "имя_процесса" | awk '{print $2}' | xargs kill
```

Здесь `ps` выводит список действующих процессов, который затем фильтруется с помощью `grep`, чтобы остались только процессы, которые соответствуют шаблону. После этого команда `awk` извлекает из каждой подходящей строки второй столбец (идентификатор процесса) и передает полученные идентификаторы команде `xargs` (аналогу цикла `for`), которая выполняет для каждого процесса команду `kill`: все процессы получают сигнал `SIGTERM` и завершают работу (по крайней мере, мы на это надеемся).

Как создавать сжатые резервные копии с помощью tar и gzip

Хотя во многих утилитах есть флаги, которые позволяют одновременно архивировать и сжимать данные, иногда стоит объединять архивацию и сжатие с помощью конвейера, особенно когда нужна гибкость, чтобы добавлять в цепочку дополни-

---

[1] Сокр. *stream editor* (*потоковый редактор*). — *Примеч. пер.*

тельные команды. Например, если файлы надо зашифровывать, это можно сделать с помощью одного дополнительного конвейера:

```
tar cvf - /путь/к/каталогу | gzip > backup.tar.gz
```

Такой конвейер создает сжатый архив каталога: это распространенный способ хранить резервные копии. По этой схеме нередко строятся более сложные конвейеры, например:

```
ssh имя_пользователя@сервер_mysql "mysqldump --add-drop-table имя_базы_данных | \
gzip -9c" | gzip –d | mysql
```

Это особенно интересный случай: мы аутентифицируемся на сервере баз данных с помощью SSH, получаем дамп базы данных, сжимаем этот поток данных, передаем его на локальный компьютер (снова по SSH), распаковываем обратно и, наконец, размещаем на своем локальном сервере MySQL.

Вам не обязательно стремиться во что бы то ни стало составлять такие сложные команды, как эта (или другие, которые вы встречали в книге), но если вы овладеете навыком конструировать подобные конвейеры экспромтом, это поможет вам как разработчику выходить из довольно сложных профессиональных ситуаций. Хотелось бы надеяться, что материал этого раздела убедил вас, что если вы разбираетесь в примитивах перенаправления ввода-вывода, которые поддерживает Unix (с помощью операторов <, >, >>, | и файловых дескрипторов в целом), то вы, по сути, обладаете суперспособностью, которую стоит применять разумно.

Продвинутый материал: как анализировать файловые дескрипторы

В Linux можно легко *увидеть*, куда ведут файловые дескрипторы того или иного процесса. Для этого мы обратимся к немного магической виртуальной файловой системе `/proc`.

Файловая система procfs (proc virtual file system) — это специфическая для Linux абстракция, которая представляет состояние ядра и процессов в виде файлов. Данные, которые в них содержатся, поступают напрямую от системного ядра и существуют только в те моменты, когда вы к ним обращаетесь. Если просто вывести содержимое каталога `/proc`, вы увидите много файлов; вот самые важные из них в описании из справочной системы Arch Linux:

`/proc/cpuinfo`	Центральный процессор
`/proc/meminfo`	Физическая память
`/proc/vmstats`	Виртуальная память

`/proc/mounts`	Смонтированные файловые системы
`/proc/filesystems`	Файловые системы, которые были откомпилированы в ядро и чьи модули ядра сейчас загружены
`/proc/uptime`	Время бесперебойной работы систем
`/proc/cmdline`	Командная строка ядра

Но вот что самое интересное (это не отражено в таблице): в каталоге `/proc` есть подкаталог для каждого процесса в системе, и название этого подкаталога совпадает с идентификатором процесса (PID).

В каталоге `/proc/`*`pid`* файловые дескрипторы соответствующего процесса представлены в виде символических ссылок внутри подкаталога `fd`. Если вывести содержимое каталога `/proc/`*`pid`*`/fd` в расширенном формате, то вы увидите, что строки начинаются с символа `l`, который обозначает символическую ссылку (вы наверняка помните об этом из главы 5 «Файлы в Linux»).

В частности, `/proc/1` — это каталог процесса подсистемы инициализации, и чтобы увидеть ее файловые дескрипторы, можно запустить команду `ls -l /proc/1/fd`.

Давайте посмотрим на дескрипторы процесса, который связан с интерактивной оболочкой Bash; в нашей системе `ps aux | grep bash` сообщает, что ей соответствует PID 9:

```
root@server:/# ls -alh /proc/9/fd
total 0
dr-x------ 2 root root 0 Sep 1 19:16 .
dr-xr-xr-x 9 root root 0 Sep 1 19:16 ..
lrwx------ 1 root root 64 Sep 1 19:16 0 -> /dev/pts/1
lrwx------ 1 root root 64 Sep 1 19:16 1 -> /dev/pts/1
lrwx------ 1 root root 64 Sep 1 19:16 2 -> /dev/pts/1
lrwx------ 1 root root 64 Sep 5 00:46 255 -> /dev/pts/1
```

Видно, что это действительно сеанс интерактивной командной оболочки: стандартный ввод поступает от виртуального терминала `/dev/pts/1`, а стандартные потоки вывода и ошибок передаются обратно на тот же те терминал. Так и должно быть.

А что, если изучить текстовый редактор, например `vim`, который во многом подобен терминалу; в частности, его ввод и вывод передаются через терминал? Но здесь есть и дополнительная сложность: с текстовым редактором обычно связаны один или несколько файлов, открытых для записи. Как при этом выглядят их дескрипторы?

Для примера запустим `vim` и откроем в нем файл из каталога `/tmp`. Давайте откроем новый терминал и найдем идентификатор процесса `vim`, чтобы узнать, в какой каталог внутри `/proc` заглядывать дальше:

```
root@server:/# ps aux | grep vim
root  453  0.0 0.1  17232  9216  pts/1  S+  15:57  0:00  vim /tmp/hello.txt
root  458  0.0 0.0   2884  1536  pts/0  S+  15:58  0:00  grep --color=auto vim
```

Нам нужен процесс с PID 453. (Не отвлекайтесь на команду `grep`, в аргументах которой тоже упоминается `vim`.) Получив идентификатор процесса `vim`, можно посмотреть на его файловые дескрипторы:

```
root@server:/# ls -l /proc/453/fd
total 0
lrwx------ 1 root root 64 Jan  7 15:58 0 -> /dev/pts/1
lrwx------ 1 root root 64 Jan  7 15:58 1 -> /dev/pts/1
lrwx------ 1 root root 64 Jan  7 15:58 2 -> /dev/pts/1
lrwx------ 1 root root 64 Jan  7 15:58 3 -> /tmp/.hello.txt.swp
```

Как видите, потоки `stdin` (0), `stdout` (1) и `stderr` (2) указывают на терминальное устройство, как и в случае с оболочкой. Также можно видеть, что дескриптор 3 указывает на файл, который сейчас редактируется в `vim`. Если этот процесс откроет еще файлы, то для них будут созданы новые дескрипторы, которые тоже отобразятся здесь.

Изучать файловые дескрипторы — увлекательное занятие само по себе, но соответствующее умение также может пригодиться на практике, например, когда вы отлаживаете программу, которая работает неправильно из-за дефектов, или когда вы пытаетесь разобраться, что делает сценарий, который подозревается в чем-то нехорошем. Файловая система `procfs` весьма интересна, и вы не пожалеете, если потратите немного времени, чтобы ее изучить: для начала просто запустите `man proc` или прочтите вводное пособие в справочнике Arch Linux по адресу wiki.archlinux.org/title/Procfs.

Итоги

В этой главе мы связали воедино все теоретические знания и практические навыки, которые вы извлекли из предыдущего материала, чтобы овладеть одной из самых мощных функций систем Unix и Linux — возможностью пропускать потоки данных по цепочке команд с помощью конвейеров и перенаправления ввода-вывода.

В начале главы мы продемонстрировали, как операционная система представляет такие примитивы, как файловые дескрипторы, а затем начали рассматривать практические примеры перенаправления ввода и вывода. Затем мы познакомились с конвейерами, которые часто считаются одной из самых полезных функций Linux и других систем семейства Unix. После теоретического введения и нескольких наглядных примеров мы перечислили самые распространенные вспомогательные инструменты, с помощью которых профессиональные программисты управляют

потоками данных, проходящих через конвейеры. Наконец, вы увидели в действии популярные идиомы конвейерной обработки с комбинациями программ, которые полезны в реальной практике.

Материал этой главы закладывает фундамент для еще более продвинутых приемов работы с командной строкой, которые рано или поздно войдут в ваш арсенал. К этому моменту вы овладели необходимой теорией, инструментами и навыками, которые помогут вам профессионально развиваться и конструировать свои собственные команды для задач разработки, устранения неполадок и автоматизации.

Чтобы наращивать свое мастерство, применяйте полученные знания в повседневной работе! Используйте эту главу как справочник типичных примеров и продолжайте изучать новые инструменты и команды, которые пригодятся для ваших авторских рецептов фильтрации или другой обработки данных в командной строке. Очень скоро вы почувствуете себя великим волшебником!

12

Как автоматизировать задачи с помощью сценариев командной оболочки

Время от времени вы будете замечать, что повторяете одни и те же команды снова и снова, разве что с небольшими вариациями. Однажды это утомит вас настолько, что вы скажете себе: «Все, я это автоматизирую». Поскольку вы — гуру командной строки, вы решите поступить так:

1. Запустить `tail -n 20 ~/.bash_history > myscript.sh`, чтобы создать файл с 20 последними командами Bash, которые вы запускали.
2. Запустить `bash myscript.sh`, чтобы выполнить эти команды.

Хотя именно такая процедура — не самая эффективная (мы поговорим об этом позже в данной главе), это вполне рабочий способ создать и запустить сценарий командной оболочки Bash.

Эта глава — интенсивный курс по созданию сценариев Bash. Как и любой интенсивный курс программирования, он принесет пользу, только если вы будете практиковаться, самостоятельно набирать код и запускать его в своей собственной среде Linux. Мы не только продемонстрируем приемы, которые считаются современными и образцовыми способами программировать для Bash, но и поделимся собственным многолетним опытом в этой области, обратив особое внимание на распространенные заблуждения и подводные камни.

Язык сценариев Bash — не самый наш любимый, но иногда именно он лучше всего подходит, чтобы решить ту или иную проблему. Эту сторону вопроса мы тоже постараемся подробно разъяснить.

Эта глава посвящена таким темам:

- Основы разработки сценариев для Bash.
- Bash в сравнениии с другими оболочками.
- Шебанг и исполняемые текстовые файлы.
- Проверка условий.
- Условные выражения.

Зачем нужно умение писать сценарии для Bash

Сценарии командной оболочки — неотъемлемый инструмент всякого разрабочика. Даже если вы сами не пишете сценарии на постоянной основе, вам время от времени придется их читать. В этой главе мы рассмотрим основы работы со сценариями Bash, благодаря чему вы не растеряетесь в ситуациях, когда:

- Вам показывают сценарий, который кто-то разработал несколько лет назад, и просят: «Проверь, получится ли у нас использовать сценарии автоматизации, которые Стив написал до того, как ушел в Google?»
- Вам подворачивается удачная возможность написать свой собственный сценарий командной оболочки, когда вы сталкиваетесь с задачей, которую уже умеют решать существующие команды (например, если нужно фильтровать или сортировать данные, выполнять поиск и перенаправлять вывод одной программы на вход другой).
- Вам нужно полностью контролировать все, что попадает на каждый уровень Docker, когда вы составляете образ.
- Вам необходимо координировать работу другого программного обеспечения в среде Linux: последовательность запуска, проверку ошибок, досрочное завершение программ и т. д.

Можно привести бесчисленное количество примеров, когда сценарий командной оболочки — именно то решение, которое лучше всего подходит для некой конкретной проблемы. Благодаря этой главе вы овладеете навыками для того, чтобы написать тот самый специализированный сценарий.

Основы языка сценариев Bash

Язык сценариев Bash можно изучать так же, как любой другой язык программирования. У него есть свое окружение (Unix или Linux), нечто вроде стандартной библиотеки (любая установленная в системе программа с интерфейсом командной строки), переменные, порядок выполнения (циклы, проверка условий и итерация),

интерполяция, несколько встроенных структур данных (массивы, строки и логические значения — в некотором роде), а также многое другое.

На протяжении всей этой книги подразумевается, что вы — разработчик ПО, а значит, уже умеете программировать. Поэтому мы не будем учить вас всем этим стандартным возможностям языков программирования, а просто расскажем, как они выглядят в Bash, а также дадим советы касательно идиом и предостережем от распространенных ошибок.

Переменные

Как и в любом языке программрования, в Bash есть переменные, которые могут иметь или не иметь значение. Переменные, которым не присвоено значение, просто остаются пустыми, и Bash спокойно оперирует ими, если только вы не настроили оболочку так, чтобы она сообщала об ошибках для пустых переменных (`set -u`).

Как присвоить значение переменной

Чтобы присвоить значение переменной, используйте обычный знак равенства (`=`). Например, инструкция `FOOBAR=nice` присвоит переменной `FOOBAR` значение `nice`.

В Bash нет типов данных — это один из самых нетипизированных языков программирования в мире.

Имя переменной может содержать латинские буквы, цифры и символ подчеркивания, но не может начинаться с цифры.

Переменные окружения обычно принято называть прописными буквами, а внутренние переменные в сценариях Bash — строчными. Если имя переменной состоит из нескольких слов, они обычно разделяются подчеркиваниями. Как и в других языках, хорошим тоном считается, чтобы имя переменной отражало то, для чего она используется и является ли константой, а для массивов приняты имена во множественном числе:

- недопустимые имена переменных: `%foo&bar`, `2foo_bar`
- допустимое, но неудачное имя: `foo_BAR123`
- хорошее имя для переменной окружения: `PORT=443`
- хорошее имя для локальной переменной: `current_price =512`
- хорошее имя для локального массива: `words=(foo bar baz)`

Как получить значение переменной

Чтобы подставить значение переменной, ссылайтесь на нее с помощью знака доллара (`$`):

```
$ echo $FOOBAR
nice
```

Bash и другие командные оболочки

Для командных оболочек в средах семейства Unix написано великое множество программ. Многие специалисты убеждены, что одна из главных причин популярности Unix — в том, что возможности автоматизации с помощью сценариев в этой среде фактически ничем не ограничены и никогда не огранчивались.

Эта глава посвящена тому, как писать свои собственные сценарии для Bash. Многое из того, чему вы здесь научитесь, пригодится и для других командных оболочек (например, для ksh и прочих минималистичных оболочек, которые обычно находятся в `/bin/sh`), однако здесь мы в первую очередь ориентируемся на Bash.

Если говорить о сценариях командной оболочки, то Bash обеспечивает хороший баланс между широкой доступностью и достаточно богатым ассортиментом возможностей языка, чтобы на нем было удобно писать небольшие программы.

Шебанг и исполняемые текстовые файлы (они же сценарии)

В системах семейства Unix сценарий — это просто исполняемый файл в формате обычного текста. Операционная система (в Linux она часто называется *ядром*) анализирует самую первую строку файла, чтобы определить, какому интерпретатору его передавать для выполнения.

Эта первая строка начинается с так называемого шебанга[1] — пары символов, которая состоит из решетки и восклицательного знака (`#!`). За ними следует путь к интерпретатору, который должен выполнять код файла. Вот пример строки с шебангом:

```
#!/usr/bin/env bash
```

Когда ядро системы семейства Unix запускает файл с установленным битом выполнения, оно рассматривает первые несколько байтов этого файла, которые могут содержать то или иное «магическое значение». Оно может быть частью двоичного файла или человекочитаемой строкой, как в случае с шебангом. По этому значению ядро определяет, можно ли корректно выполнить этот файл. Это

[1] В английском языке восклицательный знак неофициально называется *bang*, а решетка — *hash*; она же может замещать музыкальный знак диеза (*sharp*). Традиционно считается, что название *shebang* произошло от слияния *hash* + *bang* или *sharp* + *bang*. Кроме того, в XIX в. в США слово *shebang* иногда употреблялось в значениях *пристанище*, *кров*, *салун*, *лавочка* и пр.; до сих пор в ходу фразеологизм *the whole shebang*, который примерно эквивалентен выражениям *все дела*, *вот это все*, *вся эта история* и пр. — *Примеч. пер.*

позволяет предотвращать ситуации, когда ядро, например, попытается выполнить файл изображения и произойдет отказ. Ядро или оболочка (в зависимости от системы) обеспечивают, чтобы последующая команда была выполнена. Программа `env` запускает команду и обращается к переменной окружения `PATH`, чтобы найти и выполнить `bash`.

Хотя в большинстве сценарных языков решетка обозначает комментарий и поэтому интерпретатор должен ее игнорировать, этот комментарий специального вида в начале файла сообщает операционной системе, какую команду запускать, чтобы интерпретировать остальную часть файла. Вот несколько популярных примеров:

`#!/bin/sh`	Использовать эту конкретную программу командной оболочки, которая находится в указанном месте файловой системы
`#!/usr/bin/python3`	Использовать конкретный исполняемый файл Python
`#!/usr/bin/env python`	Запросить у программы `env`, какой именно исполняемый файл Python использовать в текущем окружении. В разных системах могут быть разные версии одной и той же программы, установленные в разных местоположениях

Хотя все эти варианты имеют право на существование, рекомендуется обращаться к `/usr/bin/env`, чтобы обеспечить наилучшую переносимость. Путь `/bin/sh` — исключение из этого правила, потому что в каждой системе, совместимой с POSIX, требуется, чтобы именно по этому пути находилась оболочка, совместимая с POSIX.

Распространенные настройки Bash (параметры/аргументы)

Поскольку строка с шебангом выполняется как команда, ей можно передать аргументы. Хотя код обычно лучше не усложнять, стало распространенной практикой передавать оболочке дополнительные аргументы, особенно если эта оболочка — Bash, которая часто используется для крупных сценариев, потому что она предлагает расширенные возможности по сравнению с более простыми оболочками из `/bin/sh`.

В сценариях Bash часто передаются флаги `-e`, `-u`, `-x` и `-o pipefail`, которые иногда указываются в первой строке:

```
#!/usr/bin/env bash -euxo pipefail
```

В других случаях они настраиваются в следующей инструкции с помощью команды `set`, которая задает аргументы в Bash:

```
#!/usr/bin/env bash
set -eu -o pipefail
```

Благодаря этим настройкам поведение Bash становится еще более похожим на полноценные языки программирования, к которым вы привыкли, например:

- Сценарий немедленно завершается, если какое-либо звено конвейера отказывает.
- Если для переменной не задано значение, это интерпретируется как критическая ошибка.

Вот выжимка из документации по этим параметрам:

`-o pipefail`	Если используются конвейеры, то этот параметр означает, что ошибки, которые возникают в конвейерах, будут передаваться наружу. Если возникает более одной ошибки, то передается ближайшая ошибка с конца конвейера
`-e`	Если происходит ошибка или команда завершается с отказом, этот параметр означает, что сценарий будет немедленно завершен
`-u`	Возбуждает ошибку, если используются неинициализированные переменные

Для отладки также полезен флаг `-x`:

`-x`	Включает трассировку: каждая команда записывается в стандартный поток ошибок, прежде чем будет выполнена

Все эти аргументы, кроме `-o pipefail`, поддерживаются в большинстве оболочек Unix. Более подробные сведения о параметрах Bash можно найти в справочном руководстве по адресу manpages.org/bash.

/usr/bin/env

Не забывайте важную вещь: `/bin/sh` — это стандартный путь POSIX, который ведет к какой-либо командной оболочке, совместимой с POSIX. На это можно рассчитывать в *любой* системе семейства Linux или Unix. Чаще всего эта оболочка — не Bash, а более минималистичная программа, функциональности которой достаточно ровно для того, чтобы соответствовать стандарту POSIX, что позволяет писать широко переносимые сценарии командной оболочки. Если вам нужны другие оболочки и интерпретаторы, всегда лучше указывать префикс `#!/usr/bin/env`. Таким образом вы добьетесь, чтобы сценарий использовал правильный путь из переменной `PATH` и чтобы не возникала ошибка «Команда не найдена», если исполняемый файл находится не в `/usr/bin`.

Бывают случаи, когда ни `/usr/bin/bash`, ни `/bin/bash` не указывают на нужную командную оболочку, например:

- Системы управления пакетами или корпоративные сценарии конфигурации часто устанавливают интерпретатор не в то расположение, где он находится в вашей среде разработки.
- Когда программное обеспечение устанавливают вручную — например, чтобы воспроизвести дефект и/или обойти его, — исполняемый файл часто размещают в `/usr/local`.
- В виртуальных окружениях различных языков сценариев исполняемые двоичные файлы могут находиться в подкаталоге проекта или репозитория с исходным кодом.
- Пользователь без прав `root` может установить интерпретатор в свой домашний каталог.
- Для интерпретатора может использоваться система контроля версий (`rvm`, `nvm` и т. д.).
- В различных ОС семейства Unix и некоторых дистрибутивах Linux сторонние пакеты не устанавливаются в `/usr`.

Хотя многие разработчики не могут даже представить себе, чтобы их сценарий мог оказаться в таком нестандартном месте, тем не менее есть шанс, что рано или поздно вы столкнетесь с подобной ситуацией. Чтобы не рисковать тем, что ваша программа при этом не сможет работать, стоит взять за правило включать в сценарии преамбулу `#!/usr/bin/env bash` (или указывать другой интерпретатор, для которого написан ваш код). Тогда вам не придется заставлять коллег (или самого себя в будущем, в тот момент, когда вы очень устали, а в три часа ночи поступает срочный тикет) сталкиваться с неполадками, изучать их, находить причину и редактировать файлы с исходным кодом, если вся проблема — в том, что слегка изменилось окружение.

Специальные символы и экранирование

Вы уже знакомы со специальным символом решетки (`#`): начиная с него, вся последующая строка считается комментарием, и интерпретатор его игнорирует.

Бывают и другие символы, которые имеют особый смысл в Bash, так что если вы используете их в составе значения переменной, их нужно экранировать обратным слешем (`\`). Вот некоторые из них:

Кавычки	`"` и `'`
Скобки	`{}` `[]` `()` `<>`
Тильда	`~`
Звездочка (астериск, символ подстановки)	`*`

Амперсанд	`&`
Знак вопроса	`?`
Обычные операторы	`! = |` и т. д.

Экранируйте символы так же, как в большинстве других языков программирования:

```
$ FOO="jaa\$\'"
```

Подстановка команд

У сценариев командной оболочки есть одно преимущество, которое чрезвычайно полезно на практике: из сценария можно легко вызвать любую команду. На этом основана подстановка команд, благодаря которой в сценарии доступен вывод одной или более команд, например:

```
echo "Текущая дата и время: $(date)"
```

Таким образом можно запускать команды: в данном случае это просто команда `date`, но с таким же успехом это может быть сложное конвейерное выражение. Вместо знака $ и скобок можно заключить команду в обратные кавычки; например, эта инструкция выведет то же самое, что предыдущая:

```
echo "Текущая дата и время: `date`"
```

Проверка условий

Команды для проверки условий обычно применяются вместе с инструкциями, которые управляют потоком выполнения. Как функция строковой проверки [[, так и функция арифметической проверки ((возвращает 0, если результат проверки равен true, и 1 — если false. Это связано с тем, что, когда команды оболочки успешно завершаются, они возвращают 0 в отличие от многих других языков программирования, где значение 0 ассоциировано с false [1]. В Bash нет встроенного булева типа данных; в логическом контексте используются целые числа 0 и 1. Если нужны именно переменные `true` (*истина*) и `false` (*ложь*), их можно инициализировать в коде сценария.

[1] Возвращение нулевого значения позволяет определить, что программа выполнилась корректно, в то время как отличные от нуля значения позволяют понять конкретную причину ошибки в случае ее неверного завершения или поведения. — *Примеч. науч. ред.*

Операторы проверки условий

Вот элементарные логические операторы, с помощью которых можно конструировать инструкции в Bash. По сути, они аналогичны операторам, которые вы встречали в других языках:

`!`	НЕ
`&&`	И
`||`	ИЛИ

А следующие операторы сравнения можно использовать как со строками, так и с числами:

`==`	равно
`!=`	не равно

[[Условия со строками и файлами]]

Составная команда `[[ … ]]` позволяет формулировать условия со строковыми данными. Как упоминалось ранее, в Bash нет строгой типизации, к которой вы, возможно, привыкли в других языках, и мы называем соответствующие данные и операции «строковыми», потому что они ведут себя примерно так же, как одноименные данные и операции в типизированных языках.

Вот как можно создать домашний каталог пользователя, если этого каталога не существует:

```
if [[ ! -d $HOME ]]; then
    echo "Создаем домашний каталог: ${HOME}…"
    mkdir -p $HOME
    echo "Готово"
fi
```

Восклицательный знак в первой строке обозначает логическое отрицание в Bash, поэтому всю строку можно прочитать так: «Если не верно, что `$HOME` — это каталог, то...».

А вот немного более сложный пример: как создать домашний каталог, если его не существует, ИЛИ если значение переменной `ALWAYSCREATE`[1] равно `yes`:

```
if ! [[ -d $HOME ]] || [[ $ALWAYSCREATE == yes ]]; then
    echo "Создаем домашний каталог: ${HOME}…"
```

[1] *ВСЕГДА СОЗДАВАТЬ. — Примеч. пер.*

```
    mkdir -p $HOME
    echo "Готово"
fi
```

Полезные операторы для проверки строковых условий

`-z`	Переменная не инициализирована
`-n`	Значение переменной имеет ненулевую длину
`=~`	Левый операнд (строка) соответствует правому (регулярному выражению), например: `[[ foobar =~ f*bar ]]`

Полезные операторы для проверки файловых условий

`-d`	Файл является каталогом
`-e`	Файл существует
`-f`	Обычный файл
`-S`	Файл является сокетом
`-w`	Файл доступен для записи для процесса Bash

((Условия с арифметическими выражениями))

Арифметическое условие, которое вычисляется в круглых скобках `(( … ))`, возвращает 1, если выражение в скобках имеет значение 0, и возвращает 0 в остальных случаях. Это приводит к тому, что проверка условий вполне интуитивно работает с операторами, которые вы уже встречали практически во всех остальных языках программирования:

`>`	больше
`>=`	больше или равно
`<`	меньше
`<=`	меньше или равно
`==`	равно

Рассмотрим простое арифметическое условие `(( $SOME_NUMBER[1] == 24 ))`. Вот как оно ведет себя, если значение переменной действительно равно 24:

[1] *НЕКОТОРОЕ ЧИСЛО. — Примеч. пер.*

```
→ SOME_NUMBER=24
→ (( $SOME_NUMBER == 24 ))
→ echo $?
0
```

Команда `echo $?` выводит код завершения предыдущей команды, то есть значение, которое вернула арифметическая проверка. Вот что произойдет, если присвоить переменной другое значение (не обязательно числовое):

```
→ SOME_NUMBER=foobar
→ (( $SOME_NUMBER == 24 ))
→ echo $?
1
```

Если переменная `SOME_NUMBER` не инициализирована (то есть удовлетворяет логической проверке `[[ -z $SOME_NUMBER ]]`), то аналогичное арифметическое условие приведет к ошибке:

```
→ unset SOME_NUMBER
→ (( $SOME_NUMBER == 24 ))
-bash: ((: == 20: syntax error: operand expected (error token is "== 24")[1]
```

Подытожим все, что мы узнали из этих примеров:

- Условие `(( $SOME_NUMBER == 24 ))` вернет 0, если значение переменной `SOME_NUMBER` равно 24.
- Это же условие вернет 1, если переменная `SOME_NUMBER` имеет любое значение, отличное от 24 (не обязательно числовое).
- Если переменная `SOME_NUMBER` не инициализирована, вы увидите сообщение об ошибке, потому что у арифметической проверки нет левого операнда, с которым можно было бы сравнить правый.

Условные инструкции: if/then/else

Условные инструкции в Bash обычно записываются в такой форме:

```
if [[ проверка ]]; then выражения else другие_выражения fi
```

Перед тем как перейти к практическим примерам, обратите внимание на особенности этого синтаксиса[2]:

---

[1] *Синтаксическая ошибка: ожидается операнд (ошибочная лексема — "== 24"). — Примеч. пер.*

[2] Как и в других языках программирования, здесь *if — если*, *then — то*, *тогда*, *else — иначе*. Слово *fi — if* наоборот; у него нет самостоятельного значения в языке. — *Примеч. пер.*

- `if` и `fi` соответственно открывают и закрывают условное выражение.
- Точка с запятой (`;`) разделяет выражения в Bash и нужна сразу после проверки.
- Выражение для проверки находится в двойных квадратных скобках: `[[ … ]]`.
- Часть `else` не обязательна.

Вот пример условного выражения в Bash:

```
if [[ -e "example.txt" ]]; then
    echo "Файл существует!"
```

else

К предыдущему примеру можно добавить часть `else`[1]:

```
if [[ -e "example.txt" ]]; then
    echo "Файл существует!"
else
    echo "Файл не существует!"
fi
```

Циклы

Циклы в Bash строятся по общей схеме `for` / `do` / `done`, а также поддерживают инструкции `break` и `continue`, которые позволяют соответственно выйти из цикла и перейти к следующей итерации.

Циклы в стиле C

Bash поддерживает циклы в стиле C — с инициализирующим выражением, условным выражением и приращением счетчика, например:

```
for (( i=0; i<=9; i++ ))
do
    echo "Значение переменной i равно $i"
done
```

for … in

Посмотрим, как перебрать коллекцию значений в цикле `for … in`. Попробуйте запустить этот код в своей оболочке:

---

[1] Чтобы проверить несколько условий по цепочке, дополнительные условия можно включать с помощью `elif` (аналог `else if` в других языках). — *Примеч. науч. ред.*

```
for i in 1 2 3 4 5
do
    echo $i
done
```

А вот аналогичный цикл с условной инструкцией внутри:

```
for os in FreeBSD Linux NetBSD "macOS" DragonflyBSD
do
    echo "Проверяем систему ${os}…"
    if [[ "$os" == 'NetBSD' ]]; then
        echo "(Думаю, что ее можно запустить даже на электрочайнике)"
    fi
    sleep 1
done
```

Цикл while

Цикл `while` — еще одна управляющая структура, которая должна быть знакома вам по другим языкам программирования и которая в Bash работает примерно так же. Чтобы выйти из цикла, можно использовать инструкцию `break`.

Следующий сценарий читает последовательные строки из файла `lines.txt` до тех пор, пока не встретит строку `СТОП`. Последняя строка этого кода также демонстрирует, как файл можно передать в цикл с помощью перенаправления ввода. Команда `read` отвечает за то, чтобы считывать файл строка за строкой.

```
file="lines.txt"

while read line; do
    if [[ $line == "СТОП" ]]; then
        echo "Обнаружено: СТОП. Выход из цикла."
        break
    fi

    echo "Обрабатываем: $line"
    # Здесь могут быть дополнительные команды для обработки строки $line
done < "$file"
```

Экспорт переменных

Если сценарий экспортирует переменную с помощью команды `export`, то у всех дочерних оболочек, которые породил процесс этого сценария, тоже будет доступ к значению этой переменной. Таким образом можно гарантировать, что переменная будет распространяться через все области видимости (или пространства имен), которые являются потомками по отношению к области видимости (пространству имен) текущей оболочки.

Создайте переменную в командной строке:

```
MYDIR=$HOME
```

А теперь создайте файл со следующим сценарием и запустите его (предупреждаем: он не сработает):

```
#!/usr/bin/env bash

LISTING=$(ls „${MYDIR}/Documents")
echo $LISTING
```

Вы увидите ошибку: `ls: cannot access '/Documents': No such file or directory`[1], потому что этот сценарий выполняется в дочерней оболочке, у которой нет доступа к неэкспортированным переменным из родительской оболочки (то есть из вашей интерактивной оболочки). Чтобы у дочерних оболочек был доступ к вашим переменным, его надо предоставить в явном виде с помощью ключевого слова `export`:

```
export MYDIR=$HOME
```

После того как вы экспортировали переменную, запустите предыдущий сценарий еще раз и убедитесь, что теперь он «видит» переменную `MYDIR`.

Функции

Если в какой-то момент вы почувствуете, что вам нужно написать на Bash собственную функцию, то мы обычно рекомендуем перейти на другой язык программирования, который лучше подойдет для вашей бурно развивающейся программы. Тем не менее в отдельных случаях Bash остается оптимальным языком для той или иной задачи, так что мы рассмотрим самые элементарные принципы работы с функциями и дадим советы о том, как их лучше использовать.

Функцию можно объявить с помощью ключевого слова `function`:

```
function my_great_function {
    … инструкции …
}
```

Чтобы вызвать функцию, просто укажите ее имя:

```
my_great_function
```

---

[1] *Нет доступа к /Documents: нет такого файла или каталога. — Примеч. пер.*

Предпочитайте локальные переменные

Bash работает в более или менее глобальной области видимости — точнее, в отдельной области видимости для каждой оболочки (подоболочки). Многие современные языки программирования предоставляют отдельную область видимости для функций, чтобы состояние функции не влияло на глобальное состояние после того, как она завершит работу.

В функциях Bash для этого стоит объявлять переменные с ключевым словом `local`, как в этом примере:

```
#!/usr/bin/env bash

important_var=somevalue

function local_var_example() {  # Положительный пример с локальной переменной
    local important_var="Переменная изменилась локально, все в порядке"
    echo "Пример с локальной переменной: ${important_var}"
}

function bad_example() {   # Отрицательный пример с глобальной переменной
    important_var="Функция изменяет глобальную переменную!"
    echo "Пример с глобальной переменной: ${important_var}"
}

echo "До запуска функций: ${important_var}"
local_var_example
echo
echo "После функции с локальной переменной: ${important_var}"
echo
bad_example

echo "После функции с глобальной переменной: ${important_var}"
exit 0
```

Запустите этот сценарий и посмотрите, в чем разница между обычными и локальными переменными.

Перенаправление ввода-вывода

Когда вы запускаете сценарии, их вывод часто требуется перенаправить:

- в другую программу (с помощью конвейеров и символа | — см. главу 11 «Конвейеры и перенаправление ввода-вывода»);
- в обычный файл (например, в журнал);
- в особое место — например, в специальный файл устройства `/dev/null`, который функционирует как «черная дыра» для данных, которые больше не нужны.

Рассмотрим распространенные приемы перенаправления ввода-вывода помимо конвейеров.

Перенаправление ввода с помощью <

Такое перенаправление часто используется, чтобы извлечь входные данные из файла, а не из оболочки, которая породила процесс:

```
grep foobar < stuff.txt
```

Перенаправление вывода с помощью > и >>

Оператор > перенаправляет вывод в указанное место, и если это обычный файл, то все его содержимое перезаписывается:

```
ps aux | grep foo > /var/log/foo_overwrite.log
```

Каждый раз, когда вы запускаете эту команду, вывод конвейера `ps aux | grep foo` записывается в файл `/var/log/foo_overwrite.log` и заменяет его прежнее содержимое.

С другой стороны, оператор >> добавляет данные в конец выходного файла, оставляя прежнее содержимое нетронутым. Обычно именно это нужно для файлов журналов:

```
echo $(date && cat /proc/stat) >> /var/log/kernelstate.log
```

Как перенаправить стандартный поток ошибок и стандартный вывод с помощью 2>&1

Иногда нужно перенаправить в файл и стандартный вывод (`stdout`), и стандартный поток ошибок (`stderr`):

```
consul agent -dev >> /var/log/consul.log 2>&1 &
```

Эта команда запускает программу Consul[1] в режиме разработчика и благодаря последнему символу & протоколирует ее работу в фоновом режиме, перенаправляя стандартный вывод в файл журнала. Спецификатор `2>&1` нужен для того, чтобы Bash перенаправляла файловый дескриптор 2 (стандартный поток ошибок, или `stderr`) туда же, куда дескриптор 1 (стандартный вывод, или `stdout`), то есть в файл `/var/log/consul.log`.

[1] Программа для обнаружения сетевых служб и управления ими, разработанная в HashiCorp; см. consul.io. — *Примеч. науч. ред.*

Вы уже знакомы с файловыми дескрипторами `stdin`, `stdout` и `stderr`. Но что, если нужно перенаправить стандартный поток ошибок не в стандартный вывод, а в *отдельный* файл?

Интерполяция переменных с помощью ${ }

Интерполяция строк, которую поддерживают многие языки программирования, заключается в том, что часть строки замещается значением той или иной переменной. В Bash для этого служит конструкция ${}.

Посмотрите сами, что выводит этот код:

```
MYNAME=dave
echo "Привет, ${MYNAME}!"
```

Это не единственный способ интерполировать переменные в Bash, но мы предпочитаем именно его, потому что с ним меньше всего шансов нарушить работу программы из-за неожиданного формата ввода (с пробелами, специальными символами и т. д.).

Если вы используете переменную как строковое значение, применяйте такой синтаксис, даже если эта переменная существует сама по себе и ее не нужно на самом деле интерполировать в строку:

```
NAME="${MYNAME}"
```

Такой подход поможет предотвратить многие странности в поведении Bash, так что рекомендуем его придерживаться.

ПРИМЕЧАНИЕ

Если вы интерполируете переменные, то практически всегда имеет смысл запускать Bash с флагом `-u`: либо указывать его как аргумент командной строки, либо включать `set -euo pipefail` в начало сценариев, как мы рекомендуем. Тогда вам не понадобится специально проверять, инициализирована ли переменная, перед тем как ее использовать.

Ограничения сценариев командной оболочки

Bash отличается чрезвычайно богатыми возможностями, и многие из них мы не затрагиваем в этой книге. Если вы хотите подробнее ознакомиться с языком и средой Bash, обратите внимание на множество других книг на эту тему, а также

интернет-ресурсов в свободном доступе. Можно начать со справочной страницы Bash (`man bash`).

Скорее всего, на протяжении своей карьеры вы встретите огромное количество сценариев Bash. Однако весьма вероятно и то, что вам предстоит больше читать и разбирать чужие сценарии, чем писать собственные крупные программы для командной оболочки. Bash отлично подходит для небольших задач с уклоном в системное администрирование, которые можно решить с помощью уже существующих программ, если только правильно собрать их в единое решение. При этом *нет ничего хуже*, чем пытаться решать на Bash масштабные проблемы, для которых нужно нечто большее, чем совмещать друг с другом стандартные программы Linux и Unix.

Наш опыт работы с Bash привел к таким принципам:

- Лучше меньше, чем больше.
- Лучше понятный код, чем хитроумный.
- Лучше сначала перестраховаться, чем потом жалеть.

Нередки случаи, когда сценарии Bash разрастаются до такого масштаба, что их переписывают на другом языке программирования (например, на Python). Это вовсе не камень в огород Bash! Язык командной оболочки прекрасно проявляет себя в своей нише, и именно поэтому он уже столько лет остается широко распространенным и востребованным. Нет ничего плохого в том, чтобы время от времени задумываться: а остается ли Bash по-прежнему оптимальным решением для той или иной задачи?

Итоги

Эта глава была интенсивным и бескомпромиссным введением в разработку сценариев для Bash. Вам пришлось усвоить довольно плотный материал, но он заложил все необходимые основы для того, чтобы вы могли писать эффективные сценарии командной оболочки. Если нужно, перечитайте эту главу еще раз или два. Мы не только продемонстрировали синтаксис языка, но и представили наиболее удачные (на наш взгляд) приемы, благодаря которым ваши сценарии будет легче читать и сопровождать.

Старайтесь практиковаться, практиковаться и еще раз практиковаться, причем по возможности на реальных задачах, а не на искусственных примерах. Это лучший способ отточить профессиональные навыки.

13

Безопасный удаленный доступ по протоколу SSH

Протокол **SSH**[1], как швейцарский нож, наделен множеством функций, чтобы создавать защищенные соединения и передавать по ним данные через туннели. В профессиональной работе SSH понадобится вам для всего понемногу, например, чтобы:

- безопасно аутентифицироваться в удаленной системе;
- клонировать свой частный репозиторий Git;
- передавать файлы со своего компьютера на сервер или с одного сервера на другой;
- отображать веб-службу, которая работает из-под VPN, на локальный порт на своем компьютере, чтобы к ней могли обращаться пользователи вашей домашней сети;
- выполнять другие задачи, где требуется туннелировать трафик или передавать файлы через цепочку сетевых соединений.

В этой главе мы введем вас в курс дела, чтобы вы могли уверенно работать с SSH. Вы узнаете, как устроены криптосистемы с открытым ключом, благодаря чему сможете принимать рациональные решения о том, для каких задач их применять. Вы научитесь создавать ключи SSH и с их помощью аутентифицироваться на удаленном сервере. Чтобы закрепить материал, мы даже предложим вам небольшой проект, в ходе которого вы настроите доступ на основе ключей к удаленному узлу, с которым часто работаете.

Чтобы помочь вам преодолеть проблемы, неизбежно возникающие в работе с SSH, мы составили подборку самых частых ошибок SSH, которые встречаются в реальной практике. Вы узнаете, о чем говорят наиболее распространенные сообщения

---

[1] Сокр. *Secure Shell* (*безопасная оболочка*). — *Примеч. пер.*

об ошибках и как использовать встроенные средства отладки SSH, чтобы избавиться от этих ошибок.

В этой главе рассматриваются такие темы:

- Введение в криптографические системы с открытым ключом.
- Как шифровать сообщения.
- Как подписывать сообщения.
- Ключи SSH.
- Как преобразовывать ключи SSH2 в формат OpenSSH.
- Как передавать файлы.
- Как устроено туннелирование SSH.
- Конфигурационные файлы SSH.

Давайте начнем с самых элементарных понятий криптографических систем с открытым ключом, чтобы дальнейший материал этой главы не казался вам загадочной черной магией.

Введение в криптографические системы с открытым ключом

Вполне возможно, что в своей профессиональной практике вы уже сталкивались с темой криптографических систем с открытым ключом. Хотя криптография — это самостоятельная область знаний, и это не основная тема нашей книги, важно разбираться в ней хотя бы на элементарном уровне. К счастью, основные принципы этой области очень просты и их легко применять в работе. Мы постараемся ограничиться только самой необходимой теорией, после чего перейдем к практическим командам, с помощью которых можно настраивать и использовать защищенный доступ по SSH.

В криптосистеме с открытым ключом используются два ключа — открытый (публичный) и закрытый (частный); вместе они образуют так называемую *пару ключей*. Как вы наверняка догадались по названиям, открытый ключ предназначен для того, чтобы делиться им с кем угодно, а закрытый ключ должен храниться в строжайшем секрете.

Как шифровать сообщения

Если вы создали пару ключей, то любое сообщение, которое зашифровано с помощью открытого ключа из этой пары, можно расшифровать с помощью соответствующего закрытого ключа.

Представьте себе, что Алиса хочет передать Бобу зашифрованное сообщение. Для этого она загружает открытый ключ Боба, шифрует сообщение с его помощью и отправляет Бобу. У Боба есть соответствующий закрытый ключ, поэтому он может расшифровать сообщение и прочесть его. Если кто-то другой получит доступ к зашифрованному сообщению, он не сможет его расшифровать, потому что только Боб владеет закрытым ключом, который для этого нужен.

Вот почему *ни в коем случае* не стоит делиться своим закрытым ключом с кем бы то ни было. Это создает огромную брешь в безопасности.

Как подписывать сообщения

Пару ключей можно использовать и по-другому — чтобы подписать сообщение. Такая подпись служит криптографическим доказательством того, что сообщение составил тот, кто владеет ключом. Этот метод основан на том, что не только сообщение, которое зашифровано с помощью открытого ключа, можно расшифровать с помощью закрытого, но верно и обратное: если сообщение *зашифровано* с помощью закрытого ключа, его можно *расшифровать* с помощью открытого.

Если Алиса захочет подписать сообщение, она может зашифровать с помощью закрытого ключа само сообщение или его криптографически безопасную хеш-сумму. Любой, у кого есть открытый ключ Алисы, может этим ключом расшифровать сообщение и таким образом убедиться, что оно было зашифровано закрытым ключом Алисы.

Оба механизма — шифрование и подпись — часто используются совместно. Кроме того, подписи сами по себе нередко помогают подтвердить авторство и тем самым обеспечить безопасность, когда вы загружаете программное обеспечение, например, с помощью систем управления пакетами или через магазины приложений.

Стоит заметить, что механизмы шифрования и подписи с помощью открытого ключа применяются для чрезвычайно широкого круга задач — от шифрования электронной почты и веб-трафика, который передается по протоколу HTTPS, до великого множества других приложений. Другими словами, Алиса и Боб — это не обязательно люди; они могут быть компьютерами, службами и т. д.

Теперь, когда вы знаете, на какие принципы опирается SSH и как он способен обеспечить безопасный удаленный доступ, можно приступать к тому, чтобы применять эту потрясающую технологию на практике.

Ключи SSH

Скорее всего, одной из первых задач, которая встанет перед вами в контексте SSH, будет создать собственную пару ключей, которая понадобится для аутен-

тификации на сервере SSH. Классическая команда для создания ключей выглядит так:

```
ssh-keygen[1] -t ed25519 -C 'John Doe <john.doe@example.org> '
```

Она создает пару ключей по современному криптографическому алгоритму ed25519, который основан на эллиптических кривых, а строка `John Doe <john.doe@example.org>` служит комментарием. Этот комментарий ведет себя так же, как комментарии в обычных языках программирования: он может быть любой текстовой строкой и функционально ни на что не влияет. В случае SSH он будет добавлен к вашему открытому ключу, что поможет различать ключи, когда они загружаются на сервер, например в файлы `authorized_keys`. Позже в этой главе мы глубже познакомимся с этими файлами и посмотрим, как с их помощью настраивать прозрачный и защищенный доступ к удаленным серверам.

После того как вы запустите эту команду, OpenSSH задаст вам несколько вопросов о том, где нужно сохранить созданные файлы ключей и каким паролем вы хотели бы зашифровать закрытый ключ. Поскольку этот ключ позволит получать доступ к удаленным системам, убедитесь, что вы задали достаточно надежный пароль.

ПРИМЕЧАНИЕ

По умолчанию ваши ключи хранятся в каталоге `~/.ssh` внутри вашего домашнего каталога.

Теперь, после того как вы создали пару ключей, стоит еще раз повторить важнейшую практическую рекомендацию: ни в коем случае, никогда не давайте никому свой закрытый ключ. Если он окажется в чужих руках, посторонний человек сможет выдать себя за вас. Ни одна добропорядочная служба никогда не станет запрашивать ваш закрытый ключ. При этом открытый ключ можно безопасно передавать другим и размещать в открытом доступе; обычно его имя оканчивается на `.pub`, так что его легко отличить от закрытого.

К счастью, содержимое файлов с открытым и закрытым ключом выглядит очень по-разному, так что если у вас возникнут сомнения, можно просмотреть файлы, чтобы разобраться, какой из них соответствует какому ключу:

- Открытый ключ записывается в формате *алгоритм ключ комментарий*.
- Закрытый ключ начинается с открывающей строки вида `-----BEGIN OPENSSH PRIVATE KEY-----`, за которой следует ключ и похожая закрывающая строка.

[1] *Keygen* — сокр. *key generation* (*генерирование ключей*). — *Примеч. пер.*

Следите за тем, чтобы случайно не перезаписать эти файлы ключей, а их резервные копии хранить в надежном месте. Например, может неплохо подойти система управления паролями. У многих систем такого рода есть даже отдельная функция для того, чтобы хранить файлы закрытых ключей или произвольный текст.

Когда можно распространять закрытые ключи

Как мы уже неоднократно подчеркивали, если у вас есть закрытый ключ для персонального использования (например, на вашем ноутбуке), его следует шифровать (например, указав пароль при создании ключа) и ни в коем случае не передавать посторонним людям.

Однако бывают случаи, когда эти правила уместно нарушать, а именно если вы настраиваете ключи, с которыми будут работать автоматизированные системы. Если нужно, чтобы ваш сборочный сервер аутентифицировался на GitHub перед тем, как обращаться к вашей кодовой базе, то вам, скорее всего, понадобится пара ключей с *незашифрованной* закрытой частью (если только вы не планируете вручную вводить пароль каждый раз, когда запускается сборка).

Криптографические ключи прекрасно подходят для того, чтобы применять аутентификацию и шифрование в коммуникации между узлами. При этом важно убедиться, что для каждой подобной задачи вы используете специально выделенную пару ключей и не переносите одни и те же ключи с одной системы или службы на другую.

Аутентификация по SSH

Чтобы войти в удаленную систему с помощью аутентификации по SSH, понадобится команда такого рода:

```
ssh user@example.org
```

Здесь `user` — имя пользователя, под которым вы входите в систему, а `example.org` — адрес удаленного узла, к которому подключаетесь. Нередко это просто IP-адрес, а не полное доменное имя.

Если вы аутентифицируетесь по ключу SSH, его нужно передать с помощью флага `-i`[1]:

```
ssh -i ~/.ssh/id_ecdsa user@example.org
```

Когда вы обращаетесь к серверу SSH, к которому до этого не подключались, удаленный узел предоставляет вам свой цифровой отпечаток. Он позволяет убедиться,

[1] Сокр. *identity* (здесь *идентификация*). У флага `-i` нет длинной формы. — *Примеч. пер.*

что вы действительно подключены к нужному серверу и не подвергаетесь атаке посредника (MITM). Проверьте, что отпечаток сервера совпадает с ожидаемым.

Как только вы подтвердите, что доверяете этому отпечатку, он сохранится в файле `~/.ssh/known_hosts` на вашем компьютере. Если впоследствии отпечаток изменится — например, если злоумышленник подменит узел по тому IP-адресу, где размещался ваш доверенный сервер, — то ваш клиент SSH уведомит об этом и не позволит аутентифицироваться.

После того как вы пометите сервер как доверенный, он согласует с локальным клиентом SSH, какой формат идентификации будет использоваться. OpenSSH поддерживает много разных форматов, включая вход с паролем или с парой ключей. В зависимости от выбранного варианта система может запросить у вас пароль учетной записи или пароль, которым можно расшифровать ваш закрытый ключ. Когда аутентификация успешно завершится, вы войдете в систему.

Практическая работа: настраиваем подключение к удаленному серверу с помощью ключей

Допустим, у вас есть доступ к действующему серверу Linux и вы хотите настроить на нем аутентификацию по ключу. Для этого выполните следующие шаги.

Шаг 1. Откройте терминал на стороне клиента (не сервера) SSH

Чтобы осуществить последующие шаги этой процедуры, понадобится ваше локальное окружение командной строки.

Шаг 2. Создайте пару ключей

Если вы уже создали пару ключей, когда практиковались в командной оболочке по ходу чтения этой главы, мы рады за вас! Тогда этот шаг можно пропустить.

Но если у вас еще нет пары ключей SSH, создайте ее с помощью таких команд:

```
ssh-keygen -t ed25519

# чтобы закрытый ключ не был доступен для чтения посторонним пользователям
chmod 600 ~/.ssh/id_ed25519
```

Как уже упоминалось, мы настоятельно рекомендуем защитить ключ паролем для большей безопасности.

Шаг 3. Скопируйте открытый ключ на сервер

После того как вы создали пару ключей, открытый ключ нужно разместить на сервере. Обычно открытые ключи хранятся в каталоге `~/.ssh`, а их имена оканчиваются на `.pub`.

Можно вручную скопировать ключ в файл `authorized_keys` удаленного пользователя (в этом файле хранятся все доверенные открытые ключи пользователя — каждый ключ на своей строке), а можно добиться того же эффекта с помощью одной-единственной команды `ssh-copy-id`:

```
ssh-copy-id user@example.org
```

Замените в этой команде `user` на ваше имя пользователя на удаленном сервере, а `example.org` — на IP-адрес или доменное имя сервера.

Эта команда запросит пароль от вашей учетной записи на удаленном сервере. После того как вы его введете, открытый ключ добавится в файл `~/.ssh/authorized_keys` в домашнем каталоге пользователя на сервере. В результате больше не понадобится вводить пароль, чтобы входить в систему и запускать команды на удаленном узле.

Шаг 4. Проверьте, что все работает

Теперь попробуйте аутентифицироваться на сервере:

```
ssh username@example.org
```

Если все настроено правильно, то вы сможете войти в систему, не вводя пароль (если только вы не задавали отдельный пароль для ключа SSH). Теперь вы аутентифицируетесь не по короткому текстовому паролю, который злоумышленнику легче подобрать, а по *гораздо* более надежному криптографическому ключу. Добро пожаловать в мир защищенного доступа по SSH без паролей!

Как преобразовывать ключи SSH2 в формат OpenSSH

Если вы имели дело с операционными системами, которые не относятся к семейству Unix, то скорее всего, часто встречались с открытыми ключами в формате SSH2. Например, он применяется в популярнейшей программе PuTTY, с помощью которой многие пользователи Windows подключаются по SSH к удаленным узлам. Однако чтобы подключиться к серверу, который поддерживает протокол SFTP, репозиторию Git и многим другим системам, понадобится преобразовать открытый ключ SSH2 в формат OpenSSH. Посмотрим, как это сделать.

Чего мы хотим добиться

Допустим, у нас есть открытый ключ в формате SSH2, который выглядит примерно так:

```
---- BEGIN SSH2 PUBLIC KEY ----
Comment: "rsa-key-20160402"
AAAAB3NzaC1yc2EAAAABJQAAAgEAiL0jjDdFqK/kYThqKt7THrjABTPWvXmB3URI
pGKCP/jZlSuCUP3Oc+IxuFeXSIMvVIYeW2PZAjXQGTn60XzPHr+M0NoGcPAvzZf2
u57aX3YKaL93cZSBHR97H+XhcYdrm7ATwfjMDgfgj7+VTvW4nI46Z+qjxmYifc8u
VELolg1TDHWY789ggcdvy92oGjB0VUgMEywrOP+LS0DgG4dmkoUBWGP9dvYcPZDU
F4q0XY9ZHhvyPWEZ3o2vETTrEJr9QHYwgjmFfJn2VFNnD/4qeDDHOmSlDgEOfQcZ
Im+XUOn9eVsv//dAPSY/yMJXf8d0ZSm+VS29QShMjA4R+7yh5WhsIhouBRno2PpE
VVb37Xwe3V6U3o9UnQ3ADtL75DbrZ5beNWcmKzlJ7jVX5QzHSBAnePbBx/fyeP/f
144xPtJWB3jW/kXjtPyWjpzGndaPQ0WgXkbf8fvIuB3NJTTcZ7PeIKnLaMIzT5XN
CR+xobvdC8J9d6k84/q/laJKF3G8KbRGPNwnoVg1cwWFez+dzqo2ypcTtv/20yAm
z86EvuohZoWrtoWvkZLCoyxdqO93ymEjgHAn2bsIWyOODtXovxAJqPgk3dxM1f9P
AEQwc1bG+Z/Gc1Fd8DncgxyhKSQzLsfWroTnIn8wsnmhPJtaZWNuT5BJa8GhnzX0
9g6nhbk=
---- END SSH2 PUBLIC KEY ----
```

Наша задача — преобразовать его в открытый ключ в формате OpenSSH, который выглядит примерно так:

```
ssh-rsa  AAAAB3NzaC1yc2EAAAABJQAAAgEAiL0jjDdFqK/kYThqKt7THrjABTPW
vXmB3URIpGKCP/jZlSuCUP3Oc+IxuFeXSIMvVIYeW2PZAjXQGTn60XzPHr+M0NoG
cPAvzZf2u57aX3YKaL93cZSBHR97H+XhcYdrm7ATwfjMDgfgj7+VTvW4nI46Z+qj
xmYifc8uVELolg1TDHWY789ggcdvy92oGjB0VUgMEywrOP+LS0DgG4dmkoUBWGP9
dvYcPZDUF4q0XY9ZHhvyPWEZ3o2vETTrEJr9QHYwgjmFfJn2VFNnD/4qeDDHOmSl
DgEOfQcZIm+XUOn9eVsv//dAPSY/yMJXf8d0ZSm+VS29QShMjA4R+7yh5WhsIhou
BRno2PpEVVb37Xwe3V6U3o9UnQ3ADtL75DbrZ5beNWcmKzlJ7jVX5QzHSBAnePbB
x/fyeP/f144xPtJWB3jW/kXjtPyWjpzGndaPQ0WgXkbf8fvIuB3NJTTcZ7PeIKnL
aMIzT5XNCR+xobvdC8J9d6k84/q/laJKF3G8KbRGPNwnoVg1cwWFez+dzqo2ypcT
tv/20yAmz86EvuohZoWrtoWvkZLCoyxdqO93ymEjgHAn2bsIWyOODtXovxAJqPgk
3dxM1f9PAEQwc1bG+Z/Gc1Fd8DncgxyhKSQzLsfWroTnIn8wsnmhPJtaZWNuT5BJ
a8GhnzX09g6nhbk=
```

Как преобразовать открытый ключ из формата SSH2 в OpenSSH

Команда `ssh-keygen`, с помощью которой мы создавали новые ключи, умеет также преобразовывать их с флагами `-i` и `-f`[1]:

```
ssh-keygen -i -f ssh2.pub > openssh.pub
```

[1] Флаг `-i` — сокр. *import* (*импортировать*), флаг `-f` — сокр. *filename* (*имя файла*); у команды `ssh-keygen` нет флагов в длинном формате. — *Примеч. пер.*

Эта команда возьмет ключ из файла `ssh2.pub` и запишет его в файл `openssh.pub` в формате OpenSSH.

Если вы хотите просто просмотреть код ключа OpenSSH или вручную скопировать его и вставить куда-нибудь, то можно не перенаправлять стандартный вывод в файл, а обойтись такой командой:

```
ssh-keygen -i -f ssh2.pub
```

Эта команда просто выведет открытый ключ в формате OpenSSH на экран.

Более практический пример может заключаться в том, чтобы преобразовать ключ вашего коллеги `coworker.pub` в OpenSSH и добавить его в файл `authorized_keys` на сервере. Это можно сделать так:

```
ssh-keygen -i -f coworker.pub >> ~/.ssh/authorized_keys
```

После этого коллега, у которого есть соответствующий закрытый ключ, сможет подключаться к удаленному серверу под именем того же пользователя, который запустил эту команду.

Как преобразовать открытый ключ из формата OpenSSH в SSH2

Ключи можно преобразовывать и в обратном направлении — из формата OpenSSH в SSH2. Для этого просто укажите флаг `-e`[1] вместо `-i`:

```
ssh-keygen -e -f openssh.pub > ssh2.pub
```

Агент SSH

Если вы часто аутентифицируетесь на серверах с помощью ключей SSH, может быть утомительно вводить пароль от вашего закрытого ключа каждый раз, когда вы подключаетесь или переподключаетесь к удаленному узлу. Эту проблему решает агент SSH[2] — программа, которая сохраняет вашу идентификацию (закрытый ключ) в течение локального сеанса. Другими словами, она позволяет расшифровать ключ один раз, после чего хранит его в памяти до тех пор, пока вы не выйдете из системы или не откроете новый сеанс командной оболочки. Таким образом, вы один раз добавляете пару ключей, и она используется снова и снова без необходимости повторно расшифровывать закрытый ключ.

---

[1] Сокр. *export* (*экспортировать*). — *Примеч. пер.*

[2] В Linux в качестве агента SSH обычно выступает стандартная программа `ssh-agent`, но существуют и другие инструменты с аналогичными функциями, например Pageant в составе PuTTY. — *Примеч. науч. ред.*

ПРИМЕЧАНИЕ

Агент SSH не обязательно выполняется в сеансе вашей локальной оболочки: его также могут запускать различные IDE, системы управления окнами и рабочим столом и менеджеры паролей. О том, что в системе работает агент SSH, можно догадаться по тому, что приходится вводить пароль для идентификации только один раз.

Чтобы добавить ключ в хранилище агента, запустите команду `ssh-add`[1] и передайте ей в качестве аргумента путь к закрытому ключу:

```
ssh-add ~/.ssh/id_ecdsa
```

Мы рекомендуем приобрести полезную привычку указывать флаг `-t`[2] — предельное время, в течение которого расшифрованные ключи остаются в памяти. Следующая команда работает так же, как предыдущая, но задает лимит времени в 30 секунд, после которого агент удалит ключи из памяти:

```
ssh-add -t 30 ~/.ssh/id_ecdsa
```

А эта команда перечисляет все ключи, которые были добавлены в хранилище агента:

```
ssh-add -L[3]
```

Чтобы удалить все идентифицирующие данные из хранилища агента SSH, используйте флаг `-D`[4]:

```
ssh-add -D
```

ПРИМЕЧАНИЕ

Если вы добавили в хранилище агента более трех идентификаторов (ключей), вам может все равно понадобиться указывать параметр `-i` *`ключ`*, чтобы подключиться по SSH. Дело в том, что большинство серверов настроены так, чтобы отклонять аутентификацию после трех неудачных попыток, а агент SSH при подключении подставляет каждый из сохраненных ключей по очереди. Если первый ключ не сработал, агент пробует второй ключ, и т. д. Если сервер отклоняет аутентификацию после трех попыток, то агент SSH никогда не доберется до четвертого ключа.

---

[1] *Add — добавить. — Примеч. пер.*

[2] Сокр. *time* (*время*); у команды `ssh-add` нет флагов в длинном формате. — *Примеч. пер.*

[3] Сокр. *list* (*перечислить*). — *Примеч. пер.*

[4] Сокр. *delete* (*удалить*). — *Примеч. пер.*

С помощью флага `-A` можно настроить переадресацию агента, когда вы подключаетесь к удаленному узлу. Впрочем, этой возможностью не стоит злоупотреблять, и скоро мы объясним почему:

```
ssh -A user@example.org
```

Если вы подключились к удаленному узлу с помощью приведенной команды, то SSH переместит ваши ключи на этот узел, чтобы вы могли оттуда подключаться к другим узлам. Этой возможностью стоит пользоваться крайне избирательно, потому что в результате скомпрометированный узел сможет прочитать ваши закрытые ключи, которые мы настоятельно рекомендуем беречь от чужих глаз и хранить на своем компьютере или в крайнем случае в менеджере паролей. Иными словами, если в предыдущем примере узел `example.org` будет взломан, то ваши ключи SSH окажутся скомпрометированными.

И еще одно замечание о безопасности. Некоторые диспетчеры рабочего стола для Linux (например, GNOME/MATE) хранят ключи SSH в памяти неограниченно долго после того, как вы их один раз расшифровали. Это поведение по умолчанию создает угрозу безопасности, которую стоит иметь в виду.

Распространенные ошибки SSH и вывод диагностических сообщений с флагом -v

Флаг `-v`[1] переключает команду `ssh` в диагностический режим, в котором на экран выводится подробный пошаговый протокол процедуры соединения. Этот режим особенно полезен для анализа таких распространенных проблем, как ошибки аутентификации, исчерпанное время ожидания соединения и несовпадение ключей. Диагностический режим SSH запускается так:

```
ssh -v username@example.org
```

В результате вы увидите подробный отчет обо всех стадиях квитирования и соединения по протоколу SSH, и вам будет легче выявлять и устранять неполадки, которые могут при этом возникать.

Вот некоторые распространенные ошибки, которые поможет обнаружить диагностический режим SSH:

---

[1] Сокр. *verbose* (*подробный вывод*, *диагностический режим*). Многие другие команды оболочки тоже поддерживают соответствующий флаг `-v`, причем для еще более подробного вывода его можно дублировать: `-vv`, `-vvv` и т. д. Команда `ssh` допускает до трех `v`. — *Примеч. науч. ред.*

- **Отказано в доступе (открытый ключ/пароль) (Permission Denied — public key/password)**. Это означает, что сервер отклонил вашу попытку аутентификации. В диагностическом режиме вы увидите, какие ключи подставлял ваш клиент, и сможете узнать, использовался ли вообще нужный ключ. Это *чрезвычайно* популярная ошибка в ситуациях, когда на стороне клиента хранится больше трех пар ключей, а сервер допускает всего три попытки входа.
- **Истекло время ожидания (Connection Timed Out)**. Если вам долго не удается подключиться по SSH, проблема может быть в неполадках сети или в том, что указан неверный IP-адрес или порт. Флаг `-v` поможет разобраться, на каком этапе застрял процесс, и понять, удалось ли вообще клиенту достучаться до сервера.
- **Отказано в соединении (Connection Refused)**. Такое сообщение обычно означает, что служба SSH на целевом порте сервера вообще не запущена или не отвечает на запросы. В диагностическом режиме вы увидите, что клиент не смог соединиться с удаленным узлом, после чего, возможно, вам понадобится пересмотреть правила брандмауэра или настройки сервера SSH.
- **Не удалось проверить ключ сервера (Host Key Verification Failed)**. Ключ сервера не совпадает с тем, который хранится в файле `known_hosts` в вашей системе. Флаг `-v` позволит просмотреть несовпадающие ключи, и это поможет разобраться, в чем проблема. Может быть, по этому адресу или имени узла теперь размещен другой сервер?
- **Не удалось разрешить имя узла (Could not Resolve Hostname)**. Такая ошибка обычно свидетельствует о проблемах с DNS или сетью.
- **Не удалось построить маршрут до узла (No Route to Host)**. Это означает неполадки на уровне сети — может быть, неверно настроенный брандмауэр или некорректную маршрутизацию.
- **Слишком много неудачных попыток аутентификации (Too Many Authentication Failures)**. Исчерпано максимально допустимое количество попыток входа. В диагностическом режиме вы увидите, какие методы аутентификации пробовались, и, возможно, обнаружите, что клиент подставлял неправильные или непредусмотренные ключи.
- **Не удалось загрузить ключ (Key Load Errors)**. Обычно такое сообщение отображается, когда что-то не в порядке с форматом вашего ключа SSH или с правами доступа к нему. Флаг `-v` покажет, какой именно ключ пытается загрузить клиент SSH, и вы сможете изучить его формат и настройки прав доступа.

Благодаря флагу диагностического режима `-v` вам будет легче разобраться, в чем конкретно проблема и как ее устранить или, на худой конец, в каком направлении «копать» дальше.

Передача файлов

В этом разделе мы изучим команды `sftp` и `scp`, позволяющие передавать файлы между узлами. Вы увидите практические примеры, на основе которых сможете оперировать файлами в большинстве реальных ситуаций. Речь также пойдет о том, как передавать файлы без SFTP или SCP, если сервер не поддерживает эти протоколы.

SFTP

Хотя протокол OpenSSH часто служит для того, чтобы аутентифицироваться в удаленных системах, он также позволяет передавать файлы независимо от сеанса аутентификации. Обычно за это отвечает подсистема SFTP[1]: хотя ее название напоминает FTP[2], на самом деле это совершенно отдельный протокол. Как и FTP, SFTP позволяет зарегистрированным пользователям передавать файлы с локального узла на удаленный сервер или обратно. Однако FTP — незащищенный протокол, а в SFTP и аутентификация, и передача файлов защищены и полностью зашифрованы.

Многие клиенты FTP также поддерживают SFTP, например Filezilla со своим великолепным графическим интерфейсом. Однако поскольку эта книга о том, как работать в Linux с помощью командной строки, мы дадим общий обзор того, как использовать SFTP в командной оболочке.

Войти в систему можно практически так же, как по SSH:

```
sftp user@example.org
```

После аутентификации вы получите доступ к командной строке с интерфейсом в стиле FTP. Здесь поддерживаются упрощенные или модифицированные версии некоторых команд оболочки, которые уже знакомы вам из предыдущих глав, а также несколько специальных команд. Вот самые распространенные команды SFTP:

`help`	Вывести список всех команд с кратким обзором
`ls`	Вывести содержимое удаленного каталога[3]
`lls`	Вывести содержимое локального каталога
`cd`	Сменить удаленный каталог

[1] Первая буква *S* — сокр. *secure* (*защищенный*) или *SSH*. — *Примеч. пер.*

[2] Аббр. *File Transfer Protocol* (*протокол передачи файлов*). — *Примеч. пер.*

[3] В названиях команд `lls`, `lcd` и `lpwd` первая буква *l* — от *local* (*локальный*). — *Примеч. пер.*

`lcd`	Сменить локальный каталог
`pwd`	Отобразить текущий удаленный каталог
`lpwd`	Отобразить текущий локальный каталог
`get`	Загрузить файл с удаленного сервера
`put`	Загрузить файл на удаленный сервер
`chmod`	Задать права доступа к файлу или каталогу на удаленном сервере
`chown`	Задать владельцев для файла или каталога на удаленном сервере

`quit`
`exit`
`bye`
	Выйти из SFTP (для этого также можно нажать Ctrl+D)

SCP

На практике часто удобнее передавать файлы с помощью команды `scp`[1], а не `sftp`. Хотя `scp` исторически возникла независимо от SFTP, сейчас она использует подсистему SFTP. Она работает практически так же, как `cp`, но кроме этого позволяет передавать файлы между локальным и удаленным узлом.

Формат команды `scp` таков:

```
scp источник:путь/к/файлу приемник:путь/к/файлу
```

Источник, приемник или оба этих узла могут быть удаленными; в этом случае их адрес указывается в уже знакомом вам формате `user@example.org`.

Вот как может выглядеть вызов `scp`:

```
scp user@example.org:/home/user/my_remote_file /home/user/my_local_file
```

Эта команда выполняет такие операции:

1. Соединяется с сервером `example.org`.
2. Входит в систему под именем `user` (запрашивая пароль, если вы не пользуетесь ключами и агентом SSH).
3. Копирует файл `my_remote_file` на локальный компьютер под именем `/home/user/my_local_file`.

---

[1] Первоначально *SCP* — аббр. *Secure Copy Protocol* (*протокол защищенного копирования*). Этот протокол считается устаревшим, и современные версии программы `scp` реализуют протокол SFTP, а ее название чаще расшифровывается как *secure cp* (здесь *защищенная версия команды* `cp`). — *Примеч. науч. ред.*

Как видите, аргументы команды следуют в том же порядке, что и для обычной команды `cp`.

Если поменять местами источник и приемник, то команда сделает именно то, чего вы ожидали, — загрузит файл с локального компьютера на удаленный сервер:

```
scp /home/user/my_local_file user@example.org:/home/user/my_remote_file
```

Как и в случае с `cp`, можно указывать относительные пути. Например, такая команда скопирует удаленный файл в текущий каталог на локальном компьютере:

```
scp user@example.org:/home/user/my_remote_file .
```

Также по аналогии с `cp` можно рекурсивно скопировать целый каталог с помощью флага `-r` (в некоторых системах `-R`):

```
scp -r user@example.org:/home/user/directory local_directory
```

Продвинутые примеры

Как и все прочие команды и инструменты, `ssh` и `scp` можно использовать в сценариях. Например, вот как с помощью `ssh` быстро создать резервную копию базы данных:

```
ssh username@example.org "pg_dump имя_базы_данных | gzip -c" > \
database_backup.sql.gz
```

Если вы применяете в сценариях хитроумные приемы такого рода, не забудьте предусмотреть сообщения об ошибках, иначе, если процесс столкнется с проблемой, он просто молча зависнет. Подобные команды весьма полезны в задачах разработки.

Что делать без SFTP и SCP

Изредка вы можете столкнуться с ситуацией, когда на сервере недоступен SFTP и можно аутентифицироваться только в интерактивной оболочке. Если вы все-таки хотите передать файл в удаленную систему, его можно просто открыть в текстовом редакторе, но это не всегда практично (например, представьте себе целый каталог двоичных файлов). Рассмотрим технические хитрости, с помощью которых вы сможете добиться своей цели; во всех этих случаях фигурирует перенаправление ввода-вывода и/или конвейеры в сочетании с сеансом SSH.

Проще всего загрузить файл с удаленного сервера:

```
ssh user@example.org "cat /путь/к/файлу" > local_target_file
```

Вот что делает эта команда:

1. Аутентифицируется на удаленном сервере.
2. Запускает на сервере команду `cat` */путь/к/файлу*, которая выводит содержимое указанного файла в стандартный поток вывода (`stdout`).
3. Перенаправляет `stdout` в локальный файл `local_target_file.`

Как и в большинстве порядочных программ Unix, ошибки и другой посторонний вывод будут направлены в стандартный поток ошибок (`stderr`), так что вам не стоит беспокоиться о том, что посреди процесса выскочит запрос пароля и испортит содержимое файла, который вы передаете по туннелю SSH.

Как загрузить каталог со сжатием в .tar.gz

Давайте разовьем тему и загрузим целый каталог. Поскольку здесь фигурирует большой объем данных, мы упакуем каталог в архив с помощью программы `tar`[1], которая объединяет совокупность файлов и/или каталогов в один файл — так называемый *архив*.

Очень часто заархивированные файлы дополнительно сжимаются — так получаются, например, архивы с именами, которые оканчиваются на `.tar.gz` или `.tar.bz2`. Это означает, что сначала файлы были упакованы в единый архив с помощью `tar`, а затем сжаты с помощью `gzip` или `bzip2`.

```
tar czf - /home/user/directory | ssh user@example.org "tar -xvzf -C \
/home/user/"
```

Здесь мы сначала архивируем каталог `/home/user/directory` и преобразовываем результат в поток байтов.

Дефис (`-`) обозначает целевой файл. Если бы мы хотели сохранить его, а не направить в виде потока в другую программу, вместо дефиса значилось бы имя наподобие `/home/user/directory.tar.gz`. Как принято во многих командах Unix, дефис означает, что данные нужно не сохранять в файл, а передать в `stdout`.

Затем этот поток байтов передается по конвейеру процессу `ssh`, который превращает его в стандартный ввод (`stdin`) команды `tar`. В свою очередь, эта команда распаковывает и разархивирует поток и записывает полученный каталог в `/home/user/directory` на удаленном узле.

---

[1] Сокр. от *tape archive* (*архив на магнитной ленте*); первые версии `tar` записывали данные на устройства с последовательным вводом, такие как ленточные накопители. — *Примеч. науч. ред.*

Туннелирование

Туннелирование SSH применяется для того, чтобы передавать данные по соединению SSH. В этом разделе мы рассмотрим два метода туннелирования: переадресация на локальный порт и проксирование.

Переадресация на локальный порт

SSH умеет создавать защищенные зашифрованные туннели, по которым можно передавать данные между локальной и удаленной системой. Эта схема похожа на VPN и также позволяет обращаться к службам, которые доступны удаленной системе.

Это весьма полезный эффект, и его довольно легко достичь с помощью SSH. Все, что для этого нужно, — при инициализации сеанса SSH передать дополнительный аргумент `-L`[1], в котором указать целевую службу и локальный порт, на который ее нужно переадресовать.

Допустим, в удаленной системе запущен сервер HTTP на порте `8080`. Чтобы обращаться к этому серверу со своего ноутбука по порту `3000`, запустите такую простую команду:

```
ssh -L 3000:localhost:8080 user@example.org
```

Если вы теперь откроете браузер и зайдете в нем по адресу `http://localhost:3000`, то получите такой же доступ к веб-серверу, как если бы вы открыли браузер на удаленном узле и посетили в нем `http://localhost:8080`.

Проксирование

Похожий метод применяется, если вам нужен доступ к серверу, но у вас этого доступа нет, а у другого удаленного сервера есть. Представьте, что ваше веб-приложение работает на узле `app.example.org` и обращается к серверу баз данных (например, PostgreSQL) по адресу `db.example.org`, доступному только из внутренней сети `example.org`. Как и большинство баз данных в продакшен-среде, этот сервер защищен брандмауэром, который не допускает прямых соединений из внешнего мира.

С точки зрения сетевой архитектуры это выглядит так:

`localhost` → `app.example.org` → `db.example.org`

---

[1] Сокр. *local* (*локальный*). — *Примеч. пер.*

Допустим, вы хотите подключиться к этой базе данных через свой локальный клиент `psql`. Для этого можно создать такой туннель:

```
ssh -L 5000:db.example.org:5432 user@app.example.org
```

Эта команда открывает новый сеанс входа на `app.example.org`, обращается оттуда к серверу баз данных и отображает `db.example.org:5432` на `localhost:5000`.

Если вы теперь запустите на своем ноутбуке команду `psql --port=5000 --host=localhost`, то получите доступ к базе данных PostgreSQL, которая находится по адресу `db.example.org:5432` и проксируется через `app.example.org`.

Конфигурационные файлы SSH

Конфигурацию удаленных узлов можно специфицировать в файле `~/.ssh/config`. Это может пригодиться во многих ситуациях, потому что в этом файле можно хранить различные настройки узлов и подключений к ним, в том числе:

- пользовательские имена узлов;
- сведения о пользователе по умолчанию;
- порты;
- туннели, которые нужно открыть перед подключением;
- идентификационные данные (ключи).

ПРИМЕЧАНИЕ

Если у сервера, к которому вы подключаетесь, есть постоянный IP-адрес, то имеет смысл зафиксировать его в конфигурационном файле SSH, чтобы в случае аварийного восстановления не пришлось полагаться на DNS или сети CDN.

Конфигурационные файлы SSH устроены довольно просто, и здесь мы приведем пример такого файла, который иллюстрирует многие доступные возможности:

```
# Настройки по умолчанию для всех узлов (метасимвол *)
Host *
  ServerAliveInterval 30       # Каждые 30 с проверять, что соединение активно
  ForwardAgent yes             # Перенаправлять агент SSH на удаленный узел
  Compression yes              # Включить сжатие данных
  IdentityFile ~/.ssh/id_rsa   # Идентификация по умолчанию

# Особые настройки для узла example1.com
Host example1
  HostName example1.com        # Настоящее имя узла для подключения
```

```
  User john                    # Имя пользователя по умолчанию для этого узла
  Port 22                      # Порт SSH (22 — порт по умолчанию)
  IdentityFile ~/.ssh/id_ecdsa # Особая идентификация для этого узла

# Особые настройки для другого узла example2.com
Host example2
  HostName example2.com
  User jane
  Port 22000                   # Другой порт SSH для этого узла
  IdentityFile ~/.ssh/id_ed25519

# Доступ к узлу, защищенному брандмауэром, через промежуточный узел
Host target-behind-fw
  HostName 192.168.0.2         # Частный IP-адрес целевого узла
  User alice
  Port 22
  ProxyJump jump-host
    # Использовать jump-host как промежуточный узел

# Конфигурация промежуточного узла
Host jump-host
  HostName jump.example.com
    # Публичный IP-адрес или доменное имя промежуточного узла
  User jumpuser
  Port 22
  IdentityFile ~/.ssh/jump_key

# Подключение к узлу через прокси-сервер SOCKS
Host proxy-host
  HostName proxy-target.com    # Реальное имя целевого узла
  User proxyuser
  Port 22
  ProxyCommand nc -X 5 -x localhost:1080 %h %p
    # Использовать прокси-сервер SOCKS на узле localhost, порт 1080
```

Итоги

OpenSSH — многофункциональный инструмент, и мы надеемся, что вводный материал этой главы побудит вас экспериментировать и узнавать больше.

Вы узнали основные принципы, по которым работают криптографические системы с открытым ключом, благодаря чему сможете принимать рациональные решения о том, для каких задач их применять. Вы научились создавать ключи SSH и использовать их, чтобы подключаться к удаленной командной оболочке.

Надеемся, что вы также приобрели определенный практический опыт, когда выполняли упражнения и настраивали доступ по ключу к удаленному узлу, с которым вы часто работаете. Если этот удаленный узел находится на AWS или на другой платформе, где используются ключи в формате `.pem`, вы сможете преоб-

разовать их в другой нужный формат (этот фокус сам по себе впечатлит ваших коллег).

Даже если вы сами еще не натыкались на ошибки SSH, мы перечислили самые распространенные из них и продемонстрировали, как их анализировать с помощью флага `-v`.

Мы также уделили внимание тому, как применять SSH не только для интерактивных сеансов в удаленных командных оболочках, но и для того, чтобы передавать файлы, туннелировать сетевой трафик и настраивать пользовательскую конфигурацию для различных серверов. Это само по себе впечатляет, однако OpenSSH способен решать и другие задачи:

- шифровать и подписывать файлы с помощью `ssh-keygen`;
- обеспечивать двухфакторную аутентификацию через FIDO/U2F, когда ваш ключ хранится на внешнем устройстве;
- принудительно запускать определенные команды после входа в систему, что ограничивает вектор потенциальных атак, а также позволяет использовать SSH как интерфейс к той или иной службе.

Проект OpenSSH предоставляет превосходную документацию на соответствующих справочных страницах и на сайте `openssh.com`. Если для ваших задач требуется защищенное соединение между узлами и если вас интересует хорошо испытанное и проверенное временем решение, на OpenSSH определенно стоит обратить внимание. Как говорится, дерзайте и шифруйте!

14

Как управлять версиями с помощью Git

Git — это **распределенная система управления версиями**, которая за последние двадцать лет стала самой популярной системой такого рода в мире. Хотя вы, скорее всего, уже умеете пользоваться Git хотя бы на элементарном уровне, вы можете не знать распространенных идиом командной строки или некоторых редко используемых, но мощных возможностей. Здесь мы поговорим о них. Также в этой главе мы освежим некоторые фоновые знания, которые помогут прояснить часто используемые термины Git и принципы управления версиями.

В этой главе будут рассмотрены:

- Основы Git и распределенных систем управления версиями.
- Первоначальная установка Git.
- Основные команды Git.
- Стандартная терминология Git.
- Две мощные продвинутые возможности Git: бисекция и перебазирование.
- Передовые практики Git, особенно в отношении того, как эффективно работать с коммитами.
- Полезные псевдонимы командной оболочки для Git, которые сэкономят вам много нажатий на клавиши.
- Инструменты с графическим интерфейсом для работы с Git.

И наконец, в разделе «GitHub на коленке» мы представим небольшой, но вполне содержательный проект; вы сможете выполнить его на практике и задействовать навыки работы в Linux, которыми овладели к этому моменту. Мы надеемся, что

вы добросовестно попрактикуетесь, потому что это поможет еще более виртуозно обращаться с командной строкой.

Git: немного истории

Git — это распределенная система управления версиями, которую разработал Линус Торвальдс, создатель ядра Linux. История Git началась в 2005 году, когда разработчики ядра Linux испортили отношения с поставщиками проприетарной системы управления версиями BitKeeper.

В итоге Торвальдс решил создать бесплатную систему с открытым исходным кодом, которая обеспечивала бы необходимые функции для разработки ядра Linux. Всего за несколько дней он построил концептуальную модель такой системы и заложил основы будущего Git.

Новая система быстро обратила на себя внимание в многочисленных сообществах разработчиков ПО благодаря тому, что ставила во главу угла производительность, безопасность, гибкость и нелинейную разработку (возможность поддерживать тысячи параллельных ветвей проекта). Git впечатлял скоростью, целостностью данных и широкими возможностями для распределенных процессов разработки, отчего и стал любимым инструментом многих программистов и в наше время фактически служит стандартной системой управления версиями во всей индустрии разработки ПО.

Что такое распределенная система управления версиями

Традиционные системы управления версиями (такие, как **CVS** и др.) строились на основе центрального сервера, который поддерживал единое согласованное состояние репозитория в любой момент времени. Эти системы позволяли разработчикам размещать код в репозитории и извлекать его оттуда, а также поддерживали ветвление, теги и другие знакомые механизмы. Важной особенностью этих продуктов было то, что они ориентировались на централизованное управление.

В Git, а также в других распределенных системах (например, **Mercurial** и **Fossil**) принят другой подход. У каждого разработчика есть свой собственный полноценный репозиторий. Вместо того чтобы обращаться к центральному серверу, разработчики получают изменения из репозиториев друг друга. Например, разработчики Linux используют сотни независимых репозиториев. Если разработчик считает, что код в том или ином репозитории готов к выпуску, он может

запросить, чтобы соответствующие изменения были внесены в основное ядро. Отсюда происходит термин «**запрос на принятие изменений**», или **пул-реквест** (*pull request*).

Такие платформы, как GitHub, GitLab, sourcehut, и им подобные предоставляют централизованный хостинг для репозиториев Git, обслуживают вспомогательные задачи (например, авторизацию пользователей) и выполняют много других функций, связанных с разработкой программных проектов. Однако и без этих платформ Git способен эффективно функционировать и обеспечивать широкие возможности работы над проектами. В этой системе можно даже отправлять и получать исправления и пакеты изменений по электронной почте, не выходя из командной строки и клиента Git. Это облегчает совместную работу, даже если у коллег нет никаких средств коммуникации, кроме почты.

Основы работы с Git

Давайте вкратце освежим в памяти все самое важное о том, как работать с Git в командной строке. Мы предлагаем этот материал скорее в виде справочника, чем пошаговых инструкций, хотя постарались построить его так, чтобы вы все равно могли практиковаться по ходу этого раздела.

Как настроить Git с самого начала

Если вы впервые запускаете Git на компьютере, стоит настроить несколько глобальных параметров.

Во-первых, присвойте ветви по умолчанию имя `main`:

```
git config --global init.defaultBranch main
```

Во-вторых, укажите свое имя и электронный адрес по умолчанию. Этими метаданными будут сопровождаться все ваши коммиты:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

После этого можно инициализировать новый репозиторий Git.

Как инициализировать новый репозиторий Git

Создайте каталог и переключитесь в него:

```
mkdir my-repo
cd my-repo
```

Теперь запустите `git init`, чтобы инициализировать этот каталог как новый репозиторий Git:

```
git init
```

Как вносить изменения и просматривать их

Создайте файл с простым содержимым и просмотрите изменения, которые обнаружил Git:

```
echo "Всем привет! 🌍" >> README
git status
```

Как индексировать и фиксировать изменения

Проиндексируйте изменения, которые вы собираетесь зафиксировать, и посмотрите, что теперь выведет `git status`:

```
git add README
git status
```

Выведите проиндексированное содержимое:

```
git diff --staged
```

Зафиксируйте проиндексированные изменения:

```
git commit -m 'Добавлен файл README'
```

Это сокращенная форма команды `commit`, в которой комментарий (`-m`[1]) указывается напрямую.

Команду `commit` можно запустить и в интерактивном режиме, то есть просто `git commit`. В этом случае откроется текстовый редактор, который указан в переменной окружения `EDITOR` вашей оболочки. После того как вы сохраните файл и выйдете из редактора (то есть после того, как команда `$EDITOR` вернет управление), изменения будут зафиксированы.

Дополнительно: как добавить удаленный репозиторий Git

Следующая команда добавит удаленный репозиторий с локальным именем `origin`, с которым Git может взаимодействовать, передавая и принимая изменения. Это похоже на команду аутентификации SSH, которую мы рассматривали в главе 13 «Безопасный удаленный доступ по протоколу SSH», потому что, по сути, Git здесь

[1] Сокр. *message* (*сообщение*). — *Примеч. пер.*

выполняет именно эту операцию. Git поддерживает и другие протоколы, например HTTPS.

```
git remote add origin git@example.org:путь/к/репозиторию
```

Это просто воображаемый пример, но если вы работаете с реальным репозиторием, например, размещенным на GitHub, подставьте в эту команду настоящее имя сервера и путь к нужному репозиторию. GitHub и другие платформы, где можно размещать код, предоставляют подробную документацию о том, как выполнять такие операции с их репозиториями.

Как передавать и принимать изменения

Команда `git push` передает изменения из вашей текущей ветки в удаленный репозиторий Git:

```
git push -u origin HEAD
```

А эта команда принимает изменения из удаленной ветки:

```
git pull
```

Как клонировать репозиторий

Давайте клонируем удаленный репозиторий — пусть это будет весь код одного из наших учебных курсов по Linux:

```
git clone https://github.com/groovemonkey/hands_on_linux-self_hosted_
wordpress_for_linux_beginners
```

Эта команда примет историю Git для кодовой базы удаленного репозитория и установит указанный URL в качестве его источника. После этого можно будет работать с кодовой базой с помощью всех команд Git, которые вы изучили в этой главе.

Состояние репозитория можно проверять так же, как и раньше:

```
git status
```

Хотя обычно эту команду запускают, чтобы посмотреть, что изменилось, она также предоставляет информацию о текущих слияниях, показывает, какие файлы затронул конфликт слияния, помогает при бисекции и во многих других ситуациях. Если вы не до конца понимаете, что происходит в репозитории, всегда имеет смысл проверить `git status`: вполне возможно, что ваш репозиторий находится в промежуточном состоянии, которое нужно сбросить или зафиксировать перед тем, как продолжать работу.

После того как мы перечислили команды, которые вам предстоит запускать особенно часто, давайте повторим важные термины, которые традиционно сбивают с толку начинающих пользователей Git.

Термины и понятия Git

Очень важно хотя бы в общих чертах понимать терминологию, которая принята в Git. Хотя иногда возникает путаница из-за того, что в других продуктах те же самые термины употребляются в другом смысле, стоит разобраться, что они означают в Git, потому что это поможет вам действовать гораздо увереннее, например, когда вы будете устранять неполадки или анализировать сообщения об ошибках.

Предлагаем вашему вниманию обзор ключевых понятий Git.

Репозиторий

Репозиторий — это, по сути, проект, то есть корневой каталог для кода, которым управляет система контроля версий. В этом каталоге находится подкаталог `.git`. В репозитории размещается ваш исходный код, а также история его изменений.

Чистый репозиторий

Это репозиторий, в котором нет выгруженного кода, а есть только данные из каталога `.git`. На серверах, которые предоставляют хостинг для репозиториев, например на GitHub, GitLab, sourcehut, или на ваших корпоративных экземплярах Gogs или Gitea, как правило, есть каталог *`имя_проекта`*`.git` только с теми данными, которые в обычном (клонированном) репозитории находятся в каталоге *`имя_проекта`*`/.git`.

Ветки

Если сравнивать первый коммит с семечком, из которого вырастает новый репозиторий, то проект можно представлять в виде многочисленных веток. В нем есть основная ветка (см. далее), а также часто одна или больше дополнительных веток, которые отражают различные направления развития проекта.

Например, среди дополнительных веток может быть ветка старшей версии, к которой применяются исправления дефектов, но которая никогда не сливается с основной веткой. Бывают и экспериментальные ветки, которые могут сливаться или не сливаться с основной. Наконец, бывают ветки для новых функций или исправлений, которые пока находятся в разработке, но будут слиты с основной веткой по готовности, — в общем, варианты могут быть самыми разными.

Основная ветка

Основная ветка (она обычно называется `main` или `master`) — это ветка, которая назначается по умолчанию, когда репозиторий инициализируется или клонируется. В разных проектах она может содержать либо последнюю версию кода, которая находится в разработке, либо последнюю стабильную версию.

HEAD

Это самый свежий коммит в ветке, который также можно рассматривать как ее «верхушку». В командной строке `HEAD` часто используется вместе со смещением по истории коммитов.

Например, `HEAD~2` означает третий коммит с конца; таким образом, следующая команда выведет журнал трех последних коммитов:

```
git log HEAD~2
```

В сценариях и повседневной работе `HEAD` также можно считать синонимом текущей ветки, потому что это ее верхний коммит.

Теги

В отличие от веток, тегами помечаются отдельные коммиты, например, чтобы создать особую версию кодовой базы, на которую потом можно будет ссылаться.

Поверхностная выгрузка

Так называются выгрузки, которые не содержат или почти не содержат истории. Поверхностные выгрузки полезны, если из Git нужно выгрузить только код, а не целый репозиторий или его историю. Однако с такими выгрузками не работают команды и инструменты, которые оперируют историей.

Слияние

Слияние заключается в том, что код из одной ветки интегрируется в другую. Эта операция нужна во многих ситуациях: например, чтобы включить ветку той или иной новой функциональности в основную ветку, получить изменения из удаленной ветки или извлечь код из буфера. Большинство слияний происходят полностью в автоматическом режиме, хотя при конфликте слияний может понадобиться, чтобы вмешался человек.

Коммит слияния

Это коммит, который образовался в результате слияния кода. В этом случае само слияние становится коммитом. Если нужно обработать конфликт слияния, то в коммит слияния добавятся соответствующие изменения. При этом не стоит

вносить в коммит другие правки (например, дополнительные исправления дефектов), кроме случаев, когда без них нельзя обойтись технически. В коммит слияния нужно включать только те изменения, которые необходимы для самого́ слияния.

Если конфликтов нет и Git выполняет слияние в автоматическом режиме, то разработчик обычно просто подтверждает соответствующий коммит, не внося вручную никаких изменений в код или комментарий коммита.

Конфликт слияния

Когда Git не может выполнить автоматическое слияние, возникает конфликт слияния, который нужно разрешить в ручном режиме, обычно с помощью специальных инструментов. Конфликты могут возникать, когда код выгружается из репозитория или буфера, когда происходит слияние веток или во время любых других операций, которые затрагивают текущий выгруженный код. Каждый конфликт слияния нужно разрешить и закрепить результат коммитом. Команда `git status` обычно подсказывает, что делать.

Буфер

Иногда нужно законсервировать незавершенные изменения, чтобы вернуться к работе над ними позже. В Git для этого есть механизм, который называется буфером или локальным хранилищем (stash). Он устроен по принципу стека, благодаря чему из него удобно извлекать изменения с помощью команды `git stash` и применять их по порядку.

Запрос на принятие изменений

Git — это распределенная система управления версиями, в которой у каждого разработчика есть своя полная копия репозитория и множество разработчиков могут независимо друг от друга трудиться над одним и тем же проектом.

Допустим, разработчик по имени Стив внес какие-то изменения в код в своем репозитории. Он хочет, чтобы другой разработчик по имени Сара интегрировала эти изменения в кодовую базу перед предстоящим выпуском программного продукта. Стив обращается к Саре с запросом о том, чтобы она включила эти изменения в свой репозиторий, — это и есть запрос на принятие изменений, или пул-реквест.

Впрочем, во многих компаниях и проектах в качестве распределенной системы управления версиями используется не Git, а центральный официальный репозиторий кода, из которого и в который все загружают изменения. В таких проектах «запросом на принятие изменений» обычно называют запрос на то, чтобы добавить код в этот официальный репозиторий (а иногда — только в его основную ветку).

ПРИМЕЧАНИЕ

Поскольку запросы, которые обновляют единую центральную версию кодовой базы, плохо сочетаются с децентрализованной природой Git, в Git для них нет специального термина. Разные продукты, которые поддерживают такие запросы, называют их по-разному: в GitHub это «запросы на внесение изменений», в Launchpad — «предложение для слияния», а в GitLab — «запрос на слияние».

Выборочное извлечение коммитов

Иногда имеет смысл получить только некоторые изменения (коммиты) из другой ветки. Типичный пример — исправление дефектов в ветке, которая находится в разработке (например, это ветка с новыми функциями, которую планируется добавить к выпуску стабильной ветки). В таких случаях коммиты можно извлекать выборочно: в отличие от слияния, в котором участвует целая ветка, такой метод позволяет указать, какие именно коммиты следует добавить.

Бисекция

Команда `git bisect` позволяет быстро найти, какой коммит вызвал то или иное изменение; это обычно помогает узнать, какой коммит привел к тому или иному дефекту. Чтобы запустить бисекцию, укажите заведомо «плохой» коммит, в котором есть дефект, и заведомо «хороший», в котором его нет. После этого Git подберет коммиты, которые стоит проверить на наличие дефекта. Вот пример:

```
git bisect start
git bisect bad
git bisect good a0634a0
Bisecting: 675 revisions left to test after this (roughly 10 steps)[1]
```

Бисекция начинается в первой строке. Во второй строке мы сообщаем Git, что текущая версия — «плохая», потому что в ней наблюдается дефект. Допустим, мы знаем, что коммит `a0634a0` — «хороший», поэтому указываем его на третьей строке. Конечно, с таким же успехом это может быть не коммит, а тег или ветка. После этого Git сообщает, сколько версий нужно проверить.

На этой стадии нужно запускать тесты на тот дефект, с которым мы работаем. Если дефект есть, введите `git bisect bad`, если нет — `git bisect good`, и так в цикле. В конце концов вы доберетесь до конкретного коммита, с которого начался дефект.

Чтобы выйти из этого режима и вернуть `HEAD` туда, где он был до начала поиска, запустите `git bisect reset`.

---

[1] *Бисекция: после этого осталось проверить 675 ревизий (примерно 10 шагов). — Примеч. пер.*

Если вы ищете не дефект, а какое-то другое изменение, слова `good` (хороший) и `bad` (плохой) могут выглядеть неудачными. В таких случаях вместо них можно использовать `old` (старый) и `new` (новый). Правда, смешивать разные схемы нельзя: либо `good`/`bad`, либо `old`/`new`.

Чтобы ускорить процедуру бисекции, можно указать конкретные файлы и каталоги, которые могут отвечать за искомый эффект. Например, если вы точно знаете, что он произошел из *`каталога_1`* или *`каталога_2`*, можно сузить поиск таким образом:

```
git bisect start -- каталог_1 каталог_2
```

Тогда Git будет учитывать только те коммиты, которые внесли изменения в эти каталоги.

Эту процедуру можно рационализировать дальше: например, указать несколько «хороших» комментариев или даже подключить тестовый сценарий, который будет автоматически искать нужный коммит в зависимости от кода завершения. Если вам нужно прочесать множество коммитов, взгляните также на справочную страницу `man git-bisect`.

Перебазирование

Команда `git rebase` — распространенный способ поддерживать историю коммитов в простом линейном виде; для этого заданный набор изменений (например, ветка новой функциональности) воссоздается, то есть заново воспроизводится на основе нового базового коммита, а не того базового коммита, на который эти изменения наращивались изначально.

Поскольку разработка обычно ведется в распределенном режиме, настоящая история коммитов может выглядеть так:

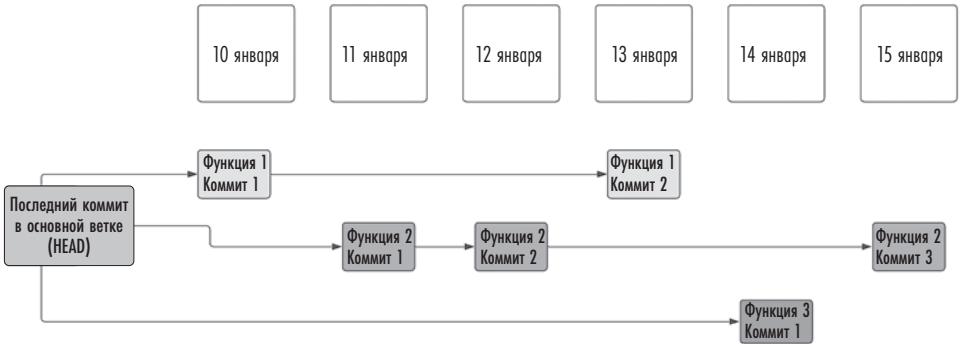


Рис. 14.1. Настоящая история коммитов

Когда в истории ветки новой функциональности так причудливо переплетаются многочисленные «сюжетные линии», это больше путает, чем помогает. Поэтому

Git позволяет перебазировать ветку, чтобы линеаризировать ее коммиты при слиянии.

Сначала сливается *Функция 1*, так что ее коммиты наращиваются на исходный базовый коммит. На этом этапе история выглядит так:

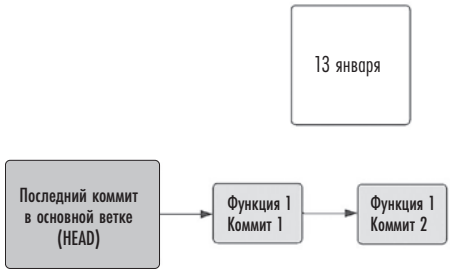


Рис. 14.2. История Git после перебазирования и слияния коммитов Функции 1 от 13 января

Затем перебазируется и сливается *Функция 3*, и история становится такой:

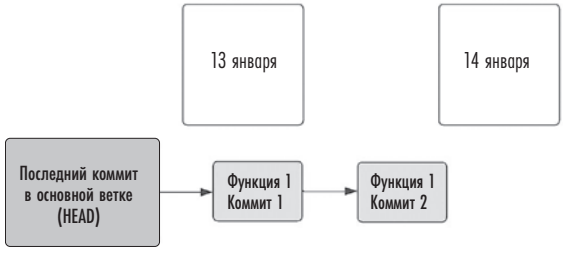


Рис. 14.3. История Git после перебазирования и слияния коммитов Функции 3 от 14 января

Наконец, Функция 2 перебазируется таким образом, что ее базовым коммитом становится коммит Функции 3 от 14 января. Вот у нас и получилась ясная и прямолинейная история:

Рис. 14.4. История Git после перебазирования и слияния коммитов Функции 2 от 15 января

GitHub и другие централизованные хостинговые платформы Git умеют автоматически выполнять эти процедуры при слиянии, так что вам редко придется заниматься перебазированием вручную в командной строке. Но на всякий случай покажем, как это делается:

1. Создайте новую ветку и добавьте коммит:

```
git checkout -b dave/myfeature
git commit -m "внесены изменения"
```

2. Если базовая ветка называется `main` и вы добавили несколько коммитов с тех пор, как начали разрабатывать другую ветку, ее можно перебазировать относительно `main`:

```
git rebase main
```

В результате история Git перестроится так, чтобы перебазировать коммиты вашей ветки на последний коммит `main`, как показано на диаграмме выше. Поскольку при этом меняется существующая история, может понадобиться принудительно выгружать изменения в официальный репозиторий (например, на GitHub), из-за чего другие разработчики могут столкнуться с конфликтами. Не забывайте об этом, когда занимаетесь перебазированием.

После того как мы рассмотрели основные понятия, с которыми вы столкнетесь при работе с Git, давайте поговорим о том, как составлять эффективные комментарии к коммитам.

Как писать комментарии к коммитам

Чтобы с коммитами (и историей) Git было удобно работать, придерживайтесь главного правила: «Одно изменение — один коммит, один коммит — одно изменение».

Нередко бывает так, что вы работаете над одним большим изменением, но вносите в код также несколько небольших (и не связанных по смыслу) правок и улучшений. В общем случае все эти дополнительные изменения стоит выгружать отдельно. Хорошо, когда каждый коммит сосредоточен ровно на одном эффекте, которого вы пытаетесь добиться: внести маленькую поправку, исправить опечатку, сменить стиль, добавить функцию (одну!) и т. д. Даже если вы сделали несколько разнородных изменений одновременно, впоследствии имеет смысл разбить их на отдельные коммиты. Чем мельче коммиты, тем удобнее ориентироваться в репозитории.

В пользу этого совета говорит множество аргументов, и вот один из самых практичных: если ваши коммиты отражают маленькие точечные изменения, то их

легко выборочно извлекать или откатывать, когда понадобится (даже если вы не рассчитывали на это, когда вносили изменения). Маленькие сфокусированные изменения также помогают разобраться в кодовой базе тем, кто запускает `git blame`[1].

Как писать хорошие комментарии

Такие расплывчатые советы, как «используя `git commit`, пишите лаконичные комментарии», часто сбивают с толку. Чтобы понять, чем хорошие комментарии отличаются от плохих, стоит представлять себе, как люди обычно используют Git. Подобно многим распределенным системам управления версиями, Git позволяет отправлять изменения по электронной почте, и в результате комментарии к коммитам сами в некотором смысле подобны электронным письмам. Первая строка считается темой, и в ней обычно ожидают увидеть краткую сводку изменений. За ней следует пустая строка, а затем — более подробный обзор того, что изменилось.

Такая схема допускает множество интерпретаций, однако за прошедшие годы были выработаны неофициальные правила, с которыми в целом согласны большинство разработчиков. Естественно, в тех или иных проектах и организациях могут быть свои традиции, однако большинство проектов с открытым исходным кодом придерживаются таких принципов:

- Первая строка (тема комментария) — короткая: это сводка длиной не более 72 символов.
- Тема начинается с глагола в повелительном наклонении[2] (`Добавить...`, `Исправить...` и т. д.).
- Тема начинается с прописной буквы.
- Если самой по себе темы недостаточно, вставьте пустую строку и добавьте расширенную сводку (тело комментария).
- В теле комментария объясните, *почему* и/или *зачем* вы внесли это изменение. Это весьма пригодится будущим читателям, которые пользуются `git blame`.
- Объясните, каким образом вы пришли к своему решению (или к реализации) и почему это важно. Это особенно актуально для сложных коммитов, а также для таких, которые не сразу понятны, если взглянуть на код. Ваши объяснения окажут неоценимую помощь тем, кто впоследствии будет отслеживать дефек-

---

[1] Подкоманда `blame` (досл. *винить*) выводит сведения о последней ревизии каждой строки файла, в частности, кто и когда последний раз вносил в нее изменения. — *Примеч. науч. ред.*

[2] Рекомендации в книге относятся к комментариям на английском языке и не обязательно применимы к другим языкам. — *Примеч. науч. ред.*

ты, удалять ненужный код, выполнять рефакторинг или просто разбираться в коде.

- Не пишите в комментарии к коммиту то, что лучше написать в комментариях к коду.
- Ориентируйтесь на рецензента или будущего читателя, у которого нет никакого контекста. Добивайтесь, чтобы такой читатель легко понял изменения в коде.

Следуя этим принципам, вы сможете сопровождать коммиты понятными и практичными комментариями. А теперь давайте немного поговорим о том, как дополнительно облегчить себе работу с Git.

Системы с графическим интерфейсом

Хотя эта книга ориентирована на то, чтобы привить вам навыки работы с командной строкой, заметим, что бывают инструменты с графическим интерфейсом, которые иногда обеспечивают дополнительное удобство.

Например, `tig` и `gitk` — это графические программы просмотра репозиториев, чей интерфейс для Git похож на инструменты большинства IDE. Чтобы попробовать эти программы, просто перейдите в репозиторий с помощью `cd` и запустите `gitk` или `tig`. Скорее всего, предварительно вам придется установить эти программы через систему управления пакетами, потому что в большинстве дистрибутивов Linux (а также в macOS) их нет по умолчанию.

Полезные псевдонимы оболочки

Вот несколько полезных псевдонимов для распространенных команд Git. Если вы используете Bash, можете включить их в свой файл `~/.bash_aliases`:

```
alias gpo='git push origin $(git branch | grep "*" | cut -d " " -f2)'
alias gp='git pull'
alias gs='git status'
alias gd='git diff'
alias gds='git diff --staged'
```

В самом деле, если вы по сто раз в день запускаете `git status`, то сэкономите себе немало времени, если вместо этого будете набирать `gs`. Если вам удобнее настроить псевдонимы как-то иначе, конечно, ничто не мешает это сделать.

Давайте теперь перейдем к тому, как применить на практике все, что мы узнали о Git. Выполним небольшой проект — свой собственный частный сервер Git под управлением Linux.

GitHub на коленке

В этом разделе мы продемонстрируем, как развернуть удаленный репозиторий Git для собственного использования. Все, что для этого понадобится, — учетная запись SSH на удаленном узле и двоичный файл Git (то есть сама команда `git`) на вашем локальном компьютере. Если Git уже установлен на удаленном узле, то вам даже не потребуются права `root`.

Цель этого учебного проекта — в том, чтобы вы освоили понятия Git, которые относятся к взаимодействию с операционной системой. Вряд ли именно так стоит налаживать Git в среде реальной эксплуатации, но здесь вы убедитесь, что в работе с Git по-прежнему нет никакой черной магии. Как и все в Linux, Git — это всего лишь файлы (в нашем случае — файлы на удаленном компьютере и туннель SSH).

Общие соображения

В зависимости от того, есть ли у вас права `root` на удаленном узле и планируете ли вы делиться репозиторием с другими людьми, вам может понадобиться создать отдельного пользователя для службы Git с совместным доступом. Эта операция полностью факультативна.

Мы будем аутентифицироваться с помощью учетной записи SSH, поэтому если вы откроете совместный доступ к репозиторию Git, то у других пользователей будут права той же учетной записи. Если вы опасаетесь, что ваши коллеги могут что-то набедокурить под этим аккаунтом, то в удаленной системе стоит завести для этого проекта отдельного пользователя (например, с именем `git`).

В этом проекте предполагается, что вы уже умеете настраивать SSH и подключаться к серверу. Если вы не до конца уверены в этих процедурах, перечитайте предыдущую главу 13 «Безопасный удаленный доступ по протоколу SSH».

Шаг 1. Подключитесь к серверу

Войдите в удаленную систему под учетной записью, которую вы хотите связать со своим репозиторием (например, `git` или ваша собственная учетная запись)[1]:

```
ssh myuser@example.com
```

[1] Не забудьте в этой и последующих командах заменить `myuser` на настоящее имя пользователя, а `example.com` — на имя или IP-адрес сервера, к которому у вас есть доступ. — *Примеч. науч. ред.*

Шаг 2. Установите Git

Прежде всего проверьте, не установлен ли уже Git на сервере:

```
git version
```

Эта команда выведет версию Git, если он установлен. Если вы увидите сообщение об ошибке наподобие `command not found`, значит, Git не установлен.

Установите пакет `git` с помощью обычной системы управления пакетами. Например, в Ubuntu для этого служит такая команда:

```
apt install git
```

Шаг 3. Инициализируйте репозиторий

Теперь можно инициализировать новый чистый репозиторий. В нашем случае он будет называться `my-project`[1], и его можно разместить в любом месте файловой системы. Для простоты настроим его в домашнем каталоге текущего пользователя:

```
git init --bare my-project.git
```

Эта команда создаст каталог `my-project.git`. Это не обычный файл, а структура каталогов, которую Git рассматривает как репозиторий. Здесь мы не будем вдаваться в подробности, однако в обозримом будущем вам вряд ли понадобится менять что-то в этой процедуре.

Хотите верьте, хотите нет, но мы сделали все, что нужно! Теперь можно отключиться от сервера (если вы подключались по SSH, нажмите Ctrl+D).

Шаг 4. Клонируйте репозиторий

Хотя репозиторий пока абсолютно пуст, его уже можно клонировать. После того как вы разорвали соединение, запустите на локальном компьютере такую команду:

```
git clone myuser@example.com:my-project.git
```

Как уже упоминалось, здесь мы предполагаем, что вы создали репозиторий в домашнем каталоге пользователя `myuser`. После доменного имени `example.com` (вместо имени здесь может быть IP-адрес вашего сервера, если вы не настроили DNS) через двоеточие указан путь относительно этого домашнего каталога. Чтобы указать абсолютный путь, начните его с косой черты. То есть команда

[1] *Мой проект. — Примеч. пер.*

`git clone myuser@example.com:/home/myuser/my-project.git` клонирует тот же самый каталог.

Git выведет предупреждение о том, что вы клонировали пустой репозиторий. В данном случае это именно то, чего мы добивались, так что все в порядке.

Шаг 5. Внесите изменения в проект и выгрузите их

Теперь можно переключиться в клонированный каталог и начать работать над проектом:

```
d my-project
echo "Мой персональный проект" >> README
git add README
git commit -m 'самый первый коммит'
```

После того как у нас есть первый коммит, его можно выгрузить на сервер. У первого коммита есть небольшая особенность: поскольку репозиторий абсолютно пуст, в нем еще нет ни одной ветки, даже основной. Git предупредит об этом, если просто запустить `git push`. Поэтому когда вы выгружаете самый первый коммит в новый репозиторий, убедитесь, что вы указали нужную ветку:

```
git push origin master
```

Вот и все!

Теперь вы сами, как и любой пользователь с доступом к вашей учетной записи SSH, можете клонировать репозиторий, а также выгружать обновления в него и из него. Вы даже можете настроить хуки и выполнить много других интересных манипуляций. Git — это чрезвычайно мощный инструмент с широкой палитрой функций. Чтобы освоиться с ними, может потребоваться время, так что считайте эту главу всего лишь отправной точкой.

Git позволяет реализовывать самые разные решения, и мы то и дело с большим воодушевлением узнаем о том, как наши коллеги по цеху применяют Git для нестандартных или уникальных проектов. Смело экспериментируйте и добивайтесь успехов!

Итоги

В этой главе вы узнали об основных понятиях, командах и рабочих процессах, которые нужны, чтобы эффективно использовать Git. Надеемся, что теперь вам стали ближе некоторые продвинутые возможности и неочевидные термины этой системы управления версиями, а также что вы примете к сведению наши

советы о таких «мягких навыках», как умение писать хорошие комментарии к коммитам.

Мы продемонстрировали псевдонимы командной строки, которые каждый рабочий день экономят нам сотни нажатий на клавиши. Хотелось бы, чтобы вам они пригодились не меньше и чтобы вы привыкли назначать псевдонимы для часто используемых команд, которые сложно запомнить и/или напечатать.

Наконец, мы надеемся, что вы поупражнялись в рамках проекта «GitHub на коленке». Чтобы запустить все команды, нужно всего несколько минут, но если вы выделите свободный вечер и попробуете все это на практике (арендуете виртуальную машину Linux на пару часов, настроите там удаленный репозиторий и выгрузите несколько коммитов), то в полной мере почувствуете, насколько мощными и эффективными могут быть ваши навыки, если с их помощью решать задачи из реального мира.

15

Контейнерная виртуализация приложений с помощью Docker

За последние десять лет контейнеры Docker стали чем-то вроде стандартного формата упаковки для веб-приложений и современных микросервисов. Если ваша программа запускается в контейнере, то она находится в предельно облегченной изолированной среде с абстракциями файловой системы Linux, процессов, пользователей и сетевых служб, и она надежно отделена от управляющей системы. Контейнерные образы также отличаются высокой переносимостью: их легко перемещать с компьютера разработчика в тестовое или предпродакшен-окружение, а затем на продакшен-сервер. Эта технология решает многие проблемы, от которых программное обеспечение и инфраструктура страдали десятилетиями.

В некотором смысле контейнеры похожи на пакеты Linux, которые вы уже умеете устанавливать из репозиториев. В первом приближении можно считать, что контейнерный образ — это сжатый архив (такой, как файл `.tar.gz`), в котором находится ваше приложение, а также необходимые конфигурационные файлы и зависимости. Этот «пакет», то есть образ, можно запустить в Docker. Прорывной возможностью контейнеризации стало то, что в контейнере все необходимые ресурсы аккуратно упакованы в виде единого артефакта и его можно запустить в любой системе Linux, где установлена среда контейнерной виртуализации (такая, как Docker).

Эта глава вполне могла бы превратиться в отдельную книгу, потому что Docker и контейнеры Linux — весьма обширные темы. Однако, как и в других главах этой книги, мы сосредоточиваемся только на основных принципах и практических навыках, которые необходимы, чтобы применять инфраструктуру на основе Docker к своим приложениям.

В этой главе мы рассмотрим такие темы:

- Какие проблемы разработки ПО и административные вопросы решают контейнеры.
- Что такое контейнеры и чем они похожи на пакеты Linux.
- Чем образы Docker отличаются от контейнеров Docker.
- Что вам как разработчику достаточно знать и уметь для использования Docker.
- Как собирать свои собственные контейнерные образы с помощью Docker-файлов (вам предстоит упаковать в контейнер реальное веб-приложение на Python).
- Продвинутые темы: чем контейнеры отличаются от виртуальных машин и как Linux абстрагирует контейнеры благодаря пространствам имен.
- Какие полезные приемы, хитрости и передовые практики мы вынесли из собственного опыта работы с контейнерной виртуализацией.

Давайте же приступим!

Как контейнеры выступают в качестве пакетов

Docker стал стандартным инструментом для того, чтобы упаковывать программное обеспечение в случаях, когда нужна готовая, заведомо работоспособная система. В контейнер Docker обычно входит как ПО, ради которого он создается, так и облегченная операционная система Linux в качестве среды выполнения. Она предоставляет библиотеки и инструменты, а также ряд других компонентов (например, конфигурационные файлы), которые позволяют ей функционировать автономно и независимо от той системы, где выполняется контейнер. Основная задача такого решения — гарантировать, что приложение будет успешно запускаться на компьютере разработчика, в тестовой и продакшен-среде, а также в других окружениях и чтобы при этом не надо было заботиться о таких подробностях, как версии операционной системы или установленные библиотеки.

Важно помнить о том, что операционная система и библиотеки при этом никуда не исчезают. В библиотеках еще могут попадаться дефекты, а все упакованные зависимости нужно обновлять из соображений безопасности и по другим причинам. Однако у потребителей упакованного ПО, будь то конечные пользователи, системные операторы или системы координации, теперь есть единообразный пакет, с которым не нужно беспокоиться о системных зависимостях. Хотя особенности запуска и конфигурации программного обеспечения по-прежнему зависят от него самого, выполняется оно стандартизованным способом (в контейнере).

В общих чертах это означает, что все особые настройки среды, например зависимости, которые нужно установить, теперь описываются в рамках Docker-файла.

Если вы создали работоспособный контейнерный образ и ему не требуется особая конфигурация, то он будет запускаться в любой системе, где работает Docker, а точнее, где поддерживаются образы OCI.

ПРИМЕЧАНИЕ

Проект OCI (Open Container Initiative[1]) вырабатывает стандарты, которые описывают формат образов, процедуру выполнения контейнеров в Linux и другие технические условия. Иногда аббревиатуру «OCI» употребляют как синоним Docker в том смысле, что разработчик может создать образ с помощью Docker, хотя для того, чтобы запускать и координировать этот образ, Docker как таковой может не потребоваться.

Лучший способ познакомиться с Docker — установить его на своем компьютере и попробовать выполнить разные команды, чем мы сейчас и займемся.

Как установить Docker

В первую очередь загрузите и установите Docker для рабочего стола (Docker Desktop). Для этого следуйте инструкции по адресу docs.docker.com/get-docker.

Существует также превосходное официальное учебное пособие по Docker для начинающих, однако мы предлагаем вам перейти к нему только после того, как вы прочтете эту главу. Здесь мы рассмотрим основные понятия, но будем уделять внимание не столько конкретным флагам командной строки, сколько тому, как вам предстоит применять соответствующие команды и процессы в профессиональной разработке ПО.

После того как вы установили Docker, давайте создадим наш первый контейнер.

Введение в Docker

Образ Docker ведет себя подобно пакету: это статический артефакт, который можно сохранять, размещать в файловой системе и перемещать с места на место. Он становится **контейнером**, когда его выполняют на том или ином компьютере. Эта разница важна, потому что разработчики иногда путаются в соответствующих терминах. Образ Docker — это неизменяемый базовый ресурс, из которого запускается контейнер — выполняющийся процесс в пространстве имен. Иными

[1] *Инициатива открытых контейнеров. — Примеч. пер.*

словами, образы — это заранее заготовленные шаблоны, из которых во время выполнения развертываются действующие контейнеры.

Образы неизменяемы по своей природе. Например, если вы загрузите образ вебсервера `nginx` и запустите его, то любые изменения, которые вы произведете в полученном контейнере, никак не скажутся на исходном образе. Это часто бывает сюрпризом для тех, кто привык иметь дело с долгоживущими виртуальными машинами, которые можно настроить один раз, а затем многократно запускать и останавливать, причем между запусками они будут сохранять свое внутреннее состояние.

Контейнеры Docker — другое дело. В идеале контейнер должен быть эфемерным и не сохранять состояния, в то время как образ, который его породил, — это стабильный шаблон, на его основе можно развернуть сколько угодно контейнеров в различных средах выполнения.

Далее мы продемонстрируем типичный рабочий процесс Docker на примере; это поможет вам познакомиться с самыми важными командами. Не старайтесь запоминать все команды; позже в этой главе мы рассмотрим их подробнее. Здесь же мы просто объясним, что происходит на каждом этапе, чтобы вы не удивлялись тому, что увидите на экране, когда будете запускать очередной микросервис.

Прежде всего давайте с помощью команды `docker run` запустим контейнер `nginx`, а в нем — командную оболочку Bash (`/bin/bash`) в интерактивном режиме (флаги `-it`):

```
→ ~ docker run -it nginx /bin/bash
```

В результате вы увидите приглашение командной строки из контейнера, который был запущен из образа `nginx`. Теперь оболочка Bash в контейнере подключена к нашему терминалу:

```
root@e96107c9a58e:/#
```

Давайте создадим файл `test.txt` и убедимся, что он появился:

```
root@e96107c9a58e:/# echo "Файл в контейнере " >> test.txt
root@e96107c9a58e:/# cat test.txt
Файл в контейнере
```

Чтобы выйти из оболочки, запустите команду `exit` или нажмите Ctrl+D:

```
root@e96107c9a58e:/#
exit
```

Контейнер завершил работу, и мы вернулись в свою обычную оболочку. Теперь посмотрим на эффект, который сбивает с толку многих начинающих пользовате-

лей Docker: давайте заново запустим первую команду, чтобы снова развернуть контейнер `nginx` из образа Docker, и проверим наш файл:

```
→ ~ docker run -it nginx /bin/bash
root@c3b4d95ab9e6:/# cat test.txt
cat: test.txt: No such file or directory
```

Караул! Отправляйте сообщение об ошибке! Docker сломался!

На самом деле нет. Это просто не тот же самый контейнер, в котором вы создали файл `test.txt`. Если присмотреться внимательнее, можно заметить, что имя узла в приглашении комадной строки во втором контейнере не такое же, как в первом. Дело в том, что команда `docker run` каждый раз запускает новый контейнер из указанного образа. Контейнеры предназначены для того, чтобы выполниться, завершить работу и бесследно исчезнуть.

Впрочем, первый контейнер все еще с нами:

```
→ ~ docker ps -a
CONTAINER ID  IMAGE  COMMAND     CREATED STATUS     PORTS  NAMES
c3b4d95ab9e6  nginx  "/docker-…" 14 m    Exited (1) 6 m    agitated_hofstadter
e96107c9a58e  nginx  "/docker-…" 14 m    Exited (0) 14 m   nervous_gould
```

Чтобы избавиться от ненужного контейнера, можно запустить команду `docker rm` с идентификатором того контейнера[1], который вы хотите удалить:

```
→ ~ docker rm c3b4d95ab9e6
c3b4d95ab9e6
```

Если контейнер остановлен, его можно снова запустить командой `docker start`:

```
→ ~ docker start e96107c9a58e
e96107c9a58e
```

После этого вы увидите активный контейнер в списке процессов Docker:

```
→ ~ docker ps -a
CONTAINER ID  IMAGE  COMMAND      CREATED  STATUS       PORTS   NAMES
e96107c9a58e  nginx  "/docker-…"  18 m     Up 1 second  80/tcp  nervous_gould
```

Чтобы запустить команду внутри этого контейнера, можно вызвать `docker exec`. Давайте снова откроем Bash и подключимся к ней в интерактивном режиме (`-it`). При этом мы увидим измененное состояние файловой системы (то есть наш файл `test.txt`):

---

[1] На самом деле в качестве идентификатора достаточно указать начальную, отличающуюся от других идентификаторов часть. В данном случае: docker rm c. — *Примеч. науч. ред.*

```
→ ~ docker exec -it e96107c9a58e /bin/bash
root@e96107c9a58e:/# cat test.txt
Файл в контейнере
```

Впрочем, долговременно поддерживать контейнеры — не лучший подход. Если вы останавливаете и запускаете контейнеры, изменяя их состояние, вместо того чтобы каждый раз развертывать новый контейнер из образа, то рискуете столкнуться с теми же проблемами, которые Docker призван решать.

Давайте предотвратим все эти проблемы и принудительно удалим этот активный контейнер навсегда:

```
→ ~ docker rm -f e96107c9a58e
e96107c9a58e
```

Можно было запустить `docker stop`, а затем `docker rm`, но если запустить `docker rm` с флагом `-f`, то контейнер будет остановлен и принудительно удален в рамках одной операции.

Этот пример показывает, как Docker заботится о том, чтобы состояние неизменяемых контейнеров не отклонялось от образа, который служит источником истины для начального окружения вашего приложения. Если вам захочется внести изменения в образ, это не получится сделать напрямую, потому что образы неизменяемы.

Изменения должны быть *явными* и *намеренными*: это важно, чтобы создавать и запускать надежное программное обеспечение. Как этого добиться? Можно взять исходный образ, внести изменения в контролируемом и воспроизводимом режиме (а не в стиле «зайти на сервер по SSH и попробовать такие-то команды»), а затем сохранить изменения, создав новый образ. Встречайте всемогущие Docker-файлы!

Как создавать образы с помощью Docker-файлов

Если вам когда-нибудь приходилось создавать новый образ Docker или модифицировать существующий, например, для веб-приложения, которое вы разрабатываете, вы уже наверняка плотно работали с Docker-файлами (файлы с именем `Dockerfile`). См. официальную документацию по адресу docs.docker.com/engine/reference/builder.

У многих современных программ уже есть официальный Docker-файл (или на худой конец Docker-файл от стороннего поставщика с открытым исходным кодом). Даже если вам нужно настроить эти файлы под себя перед тем, как их использовать, рекомендуем прежде всего свериться с примерами в документации к тому

программному обеспечению, которое вы запускаете в контейнере. Когда для этого ПО выйдет крупное обновление, код из таких примеров сломается с меньшей вероятностью, чем ваш собственный Docker-файл.

Кроме того, некоторые фреймворки или среды разработки, например Spring Boot (Java), умеют генерировать образы Docker в ходе сборки.

Хотя и может случиться так, что вам самим никогда не придется работать с Docker-файлами, это маловероятно, так что лучше разобраться в том, как они устроены.

Рассмотрим очень простой Docker-файл из проекта HTTP-сервера `echo` (github.com/hashicorp/http-echo). Здесь создается образ Docker, в который упакован исполняемый двоичный файл на Go, функционирующий как простой веб-сервер:

```
FROM alpine

ADD "https://curl.haxx.se/ca/cacert.pem" "/etc/ssl/certs/ca-certificates.crt"
ADD "./pkg/linux_amd64/http-echo" "/"
RUN apk add curl
ENTRYPOINT ["/http-echo"]
```

Вот что делает этот код:

1. Берет Alpine Linux[1] в качестве базового образа, на основе которого можно строить текущий образ.
2. Загружает сертификаты и добавляет их к образу (фактически при этом изменяется файловая система итогового контейнера).
3. Копирует исполняемый файл `http-echo` из сборочного каталога в контейнерный образ.
4. Запускает команду установки пакета в Alpine, чтобы установить программу `curl`.
5. Указывает, какой исполняемый файл или какую команду запускать, когда запускается контейнер из этого образа.

Каждая строка Docker-файла начинается с *инструкции*, которая записывается прописными буквами и которую анализатор Docker-файлов умеет выполнять. В этом файле используется небольшое подмножество инструкций — `FROM`, `ADD`, `RUN` и `ENTRYPOINT`.

Вот полный список инструкций, доступных, когда вы создаете новые образы Docker с помощью Docker-файлов:

---

[1] Чаще всего за основу берется Alpine Linux благодаря его компактности — *Примеч. науч. ред.*

Инструкция	Буквальный перевод	Описание
`ARG`	Аргумент	Объявляет аргумент времени сборки — по сути, это переменная, которую можно использовать на стадии сборки
`ENV`	Окружение	Переменные окружения, которые действуют во время сборки и останутся в окружении контейнера во время выполнения (а не только сборки!). Задаются в формате *имя=значение*
`FROM`	Откуда	Базовый образ
`CMD`	Команда	Команда по умолчанию (или аргументы `ENTRYPOINT` по умолчанию), которая будет запускаться в контейнере, когда он сам запускается. Ее можно переопределить, но в Docker-файле может быть только одна инструкция `CMD`. Если таких инструкций больше одной, действует только последняя
`ADD`	Добавить	Гибкая инструкция, которая копирует файлы и каталоги, добавляя их в файловую систему образа. Кроме того, с помощью `ADD` можно копировать файлы извне образа или загружать их по удаленным адресам URL (по протоколу HTTP), а также выполнять такие сложные операции, как расширение скобок, распаковка, разархивирование и многое другое. В предыдущем примере Docker-файла инструкция `ADD` выступала как аналог `curl`, чтобы загрузить сертификат удостоверяющего центра
`COPY`	Копировать	Просто копирует файлы и каталоги; это не такая сложная, волшебная и мощная инструкция, как `ADD`
`LABEL`	Пометить	Добавляет метаданные образа в формате *имя=значение*
`EXPOSE`	Обнародовать	Информирует потребителей образа о том, какие сетевые протоколы и порты прослушивает соответствующий контейнер
`ENTRYPOINT`	Точка входа	Указывает, какую команду запускать в контейнере, когда он сам запускается. Чтобы контейнер мог обмениваться сигналами с окружающим миром, задавайте эту инструкцию в безоболочечной форме: `ENTRYPOINT ["`*исполняемый_файл*`", "`*параметр_1*`", "`*параметр_2*`", …]`

Инструкция	Буквальный перевод	Описание
`RUN`	Запустить	Запускает команду с аргументами. • Как и `ENTRYPOINT`, используется в оболочечной форме `RUN` *`команда аргумент_1 аргумент_2 …`* и в безоболочечной форме `RUN ["`*`команда`*`", "`*`аргумент_1`*`", "`*`аргумент_2`*`", …]` • Каждая инструкция `RUN` запускает новый слой образа. (Мы еще не говорили про слои, но это полезно знать) • С помощью `RUN --mount` можно временно монтировать файловые системы к контейнеру во время сборки, не копируя сами файлы в слой образа • `RUN --network` позволяет управлять сетевым контекстом, а `RUN --security` — привилегированными контейнерами
`WORKDIR`	Рабочий каталог	Задает рабочий каталог для последующих инструкций в Docker-файле; аналог команды `cd` в операционных системах семейства Unix
`SHELL`	Оболочка	Задает командную оболочку, которая будет интерпретировать команды во время сборки образа Docker вместо оболочки по умолчанию. Команды должны быть в безоболочечной форме
`STOPSIGNAL`	Стоп-сигнал	Задает системный вызов, который этот контейнер должен интерпретировать как сигнал завершения. По умолчанию это `SIGTERM`, как и в других процессах Linux
`VOLUME`	Том	Определяет, какие тома управляющей системы будут монтироваться в контейнере
`USER`	Пользователь	Переключает пользователя, от имени которого запускаются последующие команды сборки. Позволяет многократно менять пользователя во время сборки
`ONBUILD`	По факту сборки	Определяет инструкцию, которая будет вызываться, когда этот образ используется как база для другого образа
`HEALTHCHECK`	Проверка работоспособности	Функции для проверки работоспособности. Скорее всего, вам не придется их использовать, потому что у планировщика вашего контейнера есть собственная аналогичная функциональность

Совсем скоро мы перейдем к практическому комплексному примеру того, как все это можно задействовать в небольшом проекте, но давайте сперва немного подробнее рассмотрим команды, которые фигурировали ранее в этой главе.

Команды для управления контейнерами

В этом разделе мы продемонстрируем несколько более сложных, но не менее важных команд вместе с примерами того, как их можно вызывать. Вполне вероятно, что вы уже сталкивались или еще столкнетесь с такими командами при работе с Docker.

docker run

Рассмотрим более сложный пример вызова команды `docker run`, которую мы уже применяли:

```
docker run --rm --name mywebcontainer -p 8080:80 \
  -v /tmp:/usr/share/nginx/html:ro -d nginx
```

Вот за что отвечает каждый аргумент этой команды:

`--rm`	Зачистить (удалить) контейнер после того, как он завершит работу
`--name mywebcontainer`	
	Присвоить контейнеру человекочитаемое имя `mywebcontainer`
`-p 80 80:80`	Отобразить порт `8080` управляющей системы на порт `80` в контейнере. Сначала указывается внешний порт (с точки зрения среды, в которой запускается контейнер), а потом — внутренний (с точки зрения контейнера). Например, параметр `-p 4000:80` отобразит порт `80` в контейнере на `localhost:4000`
`-v /tmp:/usr/share/nginx/html:ro`	
	Монтирует том: каталог `/tmp` в управляющей системе монтируется как `/usr/share/nginx/html` внутри контейнера. Суффикс `:ro` обеспечивает, чтобы смонтированный том был доступен только для чтения (то есть чтобы изнутри контейнера нельзя было изменить файлы на этом томе)
`-d`	Запускает контейнер как службу (в фоновом режиме)
`nginx`	Образ, из которого развертывается контейнер

Если вы хотите, чтобы в контейнере по адресу `http://localhost:8080` открывалась веб-страница, добавьте в каталог `/tmp` файл `index.html`:

```
cat <<EOF > /tmp/index.html
<!doctype html>
<h1>Всем привет</h1>
<p>Это мой контейнер</p>
</html>
EOF
```

Поскольку каталог `/tmp` управляющей системы отображается внутри контейнера на каталог `/usr/share/nginx/html`, к которому `nginx` обращается за файлами HTML, веб-сервер немедленно распознает файл `index.html` и начнет его предоставлять.

Итак, благодаря томам можно запускать приложения, которые сохраняют состояние, в контейнерах, которые его не сохраняют.

docker image list

Чтобы посмотреть, какие образы загружены на ваш локальный компьютер, можно запустить `docker images` (или `docker image list`, если вам так больше нравится).

Если вы собирали и применяли много образов Docker, этот список может оказаться весьма внушительным!

```
$ docker image list
REPOSITORY   TAG           IMAGE ID       CREATED        SIZE
nginx        latest        51086ed63d8c   10 days ago    142MB
vault        latest        22fdc6314051   2 months ago   207MB
golang       1.19-alpine   d0f5238dcb8b   2 months ago   352MB
```

docker ps

Команда `docker ps`[1] во многом подобна `ps` в Linux. С ее помощью можно посмотреть, какие контейнеры запущены в вашей системе, а также уточнить контекст: каковы идентификаторы контейнеров, какие команды в них выполняются, когда они были созданы и сколько времени работают, как в них отображаются порты и т. д.

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND                  CREATED
STATUS              PORTS                NAMES
2aca849eef73    nginx    "/docker-entrypoint.…"    About a minute ago
Up About a minute    0.0.0.0:80->80/tcp    mywebcontainer
```

docker exec

Когда вы разрабатываете контейнерный образ, может возникнуть необходимость переключиться в контейнер и запустить там те или иные команды. Чтобы от-

[1] Если вы укажете флаг “-a” (--all), будут выведены все контейнеры, в том числе остановленные. — *Примеч. науч. ред.*

крыть интерактивную оболочку в активном контейнере, применяйте команду `docker exec`:

```
docker exec -it mywebcontainer /bin/bash
```

В случае с ранее запущенным контейнером `nginx` эта команда запустит оболочку Bash в среде контейнера. Когда контейнер завершит работу, пропадут все изменения состояния, которые вы внесли: новые файлы, настройки ядра и т. д. Следующая команда `docker run` просто развернет новый контейнер на основе того же самого базового образа.

docker stop

Чтобы остановить контейнер, запустите команду `docker stop` *`имя_контейнера`*. Если у контейнера нет человекочитаемого имени, можно указать его идентификатор:

```
docker stop mywebcontainer
```

Если контейнер был запущен с параметром `--rm` (как в случае с нашим контейнером `nginx`), то контейнер будет удален и его состояние (если оно отличалось от состояния базового образа) исчезнет.

Если контейнер запускался без `--rm`, то его состояние сохранится в файловой системе, и его можно будет повторно запустить командой `docker start` *`имя_контейнера`*.

Проект с использованием Docker: контейнер для приложения на Python/Flask

В этом упражнении мы контейнеризируем небольшую веб-службу, которая написана на языке Python с использованием веб-фреймворка Flask. Это очень характерный пример, и Python прекрасно подходит для контейнерной виртуализации, потому что многие проекты на этом языке знамениты своим сумбурным управлением пакетами и зависимостями. Все файлы, которые нужны для этого проекта, вы создадите сами, заодно попрактикуетесь в использовании текстового редактора с интерфейсом командной строки!

1. Как создать приложение

Прежде всего создайте новый каталог `dockerpy` и переключитесь в него:

```
mkdir dockerpy && cd dockerpy
```

Теперь давайте разработаем крошечное приложение на Python. В этом примере мы пишем код в vim, но вы можете использовать и любой другой текстовый редактор, который вам нравится, например nano.

```
vim echo_server.py
```

Наберите такой код:

```
from flask import Flask, request
import os

app = Flask(__name__)

@app.route(,/')
def echo():
    return {
        "method": request.method,
        "headers": dict(request.headers),
        "args": request.args
    }

@app.route('/health')
def health():
    return {"status": "healthy"}

if __name__ == "__main__":
    env_port = os.environ.get("PORT", 8080)
    app.run(host='0.0.0.0', port=env_port)
```

Это и есть все наше веб-приложение. Оно просто извлекает некоторые данные из входящего запроса и в зависимости от них отправляет ответ клиенту.

Сохраните файл и выйдите из редактора (Esc, `:x`).

Создайте файл `requirements.txt` с одной-единственной строкой:

```
Flask>=3.0.0
```

Наконец, создайте Docker-файл:

```
vim Dockerfile
```

Наберите такой код:

```
# Используем официальный базовый образ Python
FROM python:3.12-slim

# Задаем рабочий каталог внутри контейнера
WORKDIR /app

# Копируем перечень зависимостей в контейнер
COPY requirements.txt .
```

```
# Устанавливаем зависимости Python
RUN pip install --no-cache-dir -r requirements.txt

# Копируем сценарий в контейнер
COPY echo_server.py .

# Настраиваем проверку работоспособности так, чтобы контейнер
# принудительно завершал работу, если он не прослушивает внутренний порт
HEALTHCHECK --interval=30s --timeout=5s \
  CMD curl --fail http://localhost:8080/health || exit 1

# Открываем порт для приложения
EXPOSE 8080
ENV PORT=8080
CMD ["python", "echo_server.py"]
```

Это все, что нам нужно. Сейчас в вашем каталоге `dockerpy` должны быть три файла:

- `Dockerfile`
- `echo_server.py`
- `requirements.txt`

2. Как создать образ Docker

Чтобы собрать новый образ Docker, запустите команду `docker build`. Флаг `-t` нужен для того, чтобы пометить контейнер человекочитаемым именем:

```
docker build -t dockerpy .
```

Обратите внимание на точку (`.`) в конце команды: она заставляет Docker считать контекстом сборки текущий каталог.

3. Как развернуть контейнер из образа

Ранее в этой главе вы уже встречались с командой `docker run`. Запустите ее, чтобы развернуть контейнер из образа, который вы только что собрали:

```
docker run --rm -d -p 8080:8080 --name my-dockerpy dockerpy
# Эта команда выводит идентификатор вашего нового контейнера
```

Здесь фигурируют дополнительные аргументы:

`--rm`	Заставляет Docker удалять контейнер после того, как он завершит работу. Благодаря этому старые контейнеры не засоряют файловую систему, как в одном из примеров ранее в этой главе
`-d`	Запускает контейнер в режиме службы, чтобы он не занимал терминал на переднем плане

`-p`	Задает перенаправление портов: слева от двоеточия указан порт управляющей системы, на который он отображается, а справа — порт контейнера. Если бы приложение в контейнере работало на порте `1234` и вы хотели бы отобразить его на порт `80` на управляющем компьютере, этот аргумент выглядел бы так: `-p 80:1234`
`--name`	Задает для контейнера имя, по которому его будет легко найти в выводе команды `docker ps`

Теперь ваше приложение запущено в контейнере и к нему можно обратиться из браузера или из командной строки. Давайте применим команду `curl`, чтобы отправить запрос к этой веб-службе:

```
curl localhost:8080 {"args":{},"headers":{"Accept":"*/*",
"Host":"localhost:8080","User-Agent":"curl/8.1.2"},"method":"GET"}
```

Это должно вызвать чувство облегчения у тех, кто настрадался с хитросплетениями зависимостей, которые невозможно воспроизвести (это известный феномен в Python, Ruby и некоторых других языках). Раньше вам приходилось таскать весь этот громоздкий багаж вместе со своим приложением — из локальной среды разработки в среду непрерывной интеграции и тестирования, затем в предпродакшен-окружение и, наконец, в продакшен, но теперь все зависимости упакованы в единый артефакт, который заведомо содержит одно и то же, где бы вы его ни запускали.

Мы до сих пор не использовали еще одну команду — `docker exec`, которая позволяет запускать команды внутри активного контейнера. Это может пригодиться, если по какой-то причине вам во что бы то ни стало нужно изучать или модифицировать действующий контейнер:

```
docker exec -it my-dockerpy /bin/sh
```

Эта команда запускает в контейнере оболочку `/bin/sh` и подключает ее к вашей командной строке. (Большинство эксплуатационных контейнеров оснащены только минималистичной оболочкой `/bin/sh`, а не чем-то более функциональным вроде Bash.)

Наконец, давайте остановим сервер последней командой, которую мы здесь рассматриваем, — `docker kill`:

```
docker kill my-dockerpy
```

Эта команда передает процессу сигнал `SIGKILL` (код 9), который принудительно завершает процесс, не давая ему возможности корректно завершить работу, как в случае `SIGTERM` (код 15).

Чем контейнеры отличаются от виртуальных машин

Вы уже получили представление о рабочих процессах, с помощью которых можно создавать образы Docker и работать с ними. Однако будет нелишним также ориентироваться в том, чем контейнеры отличаются от виртуальных машин. Эта разница может оказаться важной, когда вы устраняете технические неполадки, а кроме того, о ней часто спрашивают на собеседованиях, чтобы оценить, насколько хорошо вы понимаете принципы, лежащие в основе контейнерной виртуализации.

Виртуальная машина позволяет запускать полноценную операционную систему (например, Linux, Windows или DragonFly BSD) поверх другой операционной системы (управляющей). Виртуальные машины функционируют независимо от управляющей системы. Фактически, когда вы запускаете Docker в macOS, он прозрачно задействует виртуальную машину с окружением Linux, которое нужно для Docker.

В результате в виртуальной машине работает настоящая операционная система, такая как Linux, которая, в свою очередь, использует подсистему инициализации `systemd`. Благодаря этому службами и процессами можно управлять так же, как если бы виртуальная машина была обычным компьютером. В контексте повседневного использования виртуальные машины ведут себя так же, как реальные. Но контейнеры обычно используются по-другому.

Контейнер Docker, как правило, содержит одно приложение, причем часто ему соответствует один процесс. Если в контейнере выполняется несколько процессов, в основном это происходит из-за того, что целевое приложение породило дочерние процессы, — так обычно поступают веб-серверы или исполнители команд. Поскольку общепринятый подход заключается в том, чтобы в контейнере выполнялся всего один процесс и чтобы контейнер завершал работу как можно скорее после выхода из процесса, любое внутреннее управление процессами было бы напрасной тратой ресурсов.

Вместо этого задачи, которые в полноценном Linux выполняет подсистема инициализации, передаются из контейнеризованной среды выполнения внешним системам, которые *управляют самими контейнерами*, например Kubernetes или Nomad.

В этой новой модели контейнерам отводится та же роль, которую в традиционных ОС играют процессы, а системы координации контейнеров берут на себя различные функции операционной системы и планировщика.

Процесс с PID 1, который в обычном Linux соответствует подсистеме инициализации, — это тот процесс, который задает инструкция `CMD` или `ENTRYPOINT`. Как

правило, это главный процесс программы, которая запущена в контейнере, и других процессов в нем обычно нет. Бывают случаи, когда контейнеры используются по-другому, однако здесь описана наиболее стандартная схема. Она особенно уместна, когда нужно просто упаковать в контейнер службу, которая будет запускаться в продакшен-среде. Из этого правила бывают исключения: в первую очередь в случае, когда какая-то служба была создана до широкого распространения контейнеров, — но в таких случаях вы, вероятнее всего, будете осведомлены об этом и воспользуетесь базовым образом, созданным специально для этой цели.

Репозитории образов Docker

В этой главе мы весьма активно задействовали образ `nginx`. Но откуда конкретно он взялся? По умолчанию Docker пытается загружать образы из Docker Hub (hub.docker.com) — центрального репозитория общедоступных образов Docker. Docker Hub работает примерно так же, как репозиторий пакетов Linux: он предоставляет для всеобщего использования образы, которые в нем размещены. Здесь можно найти бóльшую часть популярного серверного программного обеспечения, и использовать это ПО так же легко, как в нашем примере с `nginx`.

Однако не все приложения находятся в открытом доступе, и образы Docker нередко размещаются в частных репозиториях. Список репозиториев постоянно меняется, так что мы не будем перечислять их здесь, а лишь заметим, что они все работают так же, как Docker Hub.

Контейнеры: советы от наученных горьким опытом

Когда вы начнете собирать собственные контейнеры, то избежите многих проблем, если примете на вооружение передовые практики, которые описаны в официальной документации Docker: docs.docker.com/get-started/09_image_best.

Кроме того, мы собрали небольшой список самых вопиющих ошибок контейнерной виртуализации вместе с советами о том, как не наступать на те же грабли. Этот раздел — итог многих бессонных ночей, дорогостоящих аварий и нашего обучения на собственном горьком опыте.

Не используйте большие образы

Пока у вас еще мало опыта работы с Docker, используйте минималистичные базовые образы, такие как Scratch или Alpine. Чтобы развернуть большинство приложений, вам ни к чему большие образы и дистрибутивы вроде Ubuntu. Если вам

нужны зависимости на этапе сборки, лучше всего избавиться от них или использовать промежуточные сборочные контейнеры.

Легкие минималистичные образы не только быстрее загружаются и потребляют меньше ресурсов, но и гораздо проще в управлении. Когда в образе нет ненужных вам программ и библиотек, то у вас меньше сущностей, которые надо поддерживать в актуальном состоянии, меньше простора для атак злоумышленников и меньше докучливых предупреждений от служб проверки безопасности контейнеров.

Обращайте внимание на стандартную библиотеку C

Следите за тем, с какой стандартной библиотекой C (она же *libc*) вы работаете. Многие дистрибутивы используют `glibc`, однако некоторые (например, Alpine Linux) — `musl` или другие реализации. Эти библиотеки, а значит и программы, собранные с их помощью, могут быть несовместимы друг с другом. Например, пользователям Alpine приходится самим компилировать менее популярные инструменты. Если ваши проекты зависят от тех или иных библиотек, которые доступны в виде пакетов для выбранного базового образа, вы можете столкнуться с несовместимостью. Конечно, такие же проблемы могут возникнуть, когда вы обновляете, откатываете или полностью заменяете базовые образы.

Имейте в виду, что по мере того как Alpine и musl уверенно набирают популярность, подобные неприятности становятся менее вероятными (или их по крайней мере легче одолеть с помощью поисковой системы). Если ваш проект вовсе не зависит от библиотек C, то вам повезло: скорее всего, упомянутые хлопоты вам не грозят. Также, если вы будете статически компилировать код, это позволит вашему приложению меньше зависеть от окружающей системы, а следовательно, от базового образа.

Продакшен-среда — не ваш компьютер: не полагайтесь на внешние зависимости

Плохо, если ваше приложение зависит от локально смонтированных томов или других локальных контейнеров. Скорее всего, оно будет развернуто в окружении, которое существенно отличается от вашего компьютера. Если в вашей системе контейнер с базой данных размещен поблизости от контейнера с приложением, это не значит, что эти контейнеры будут так же соседствовать на компьютере в среде реальной эксплуатации.

То же самое касается томов с данными: по поводу этих точек соприкосновения вам имеет смысл продумать решение вместе со специалистами по DevOps. Скорее всего, вам понадобится подключаться к системе обнаружения служб, службам проверки работоспособности и совместно используемым томам с помощью планировщика или другого инструмента DevOps.

Немного теории: контейнеры и пространства имен

Если вам интересно, как вся эта контейнерная магия работает «под капотом», или вы просто опасаетесь, что однажды вам придется устранять неполадки в контейнерном окружении в ситуации аврала, — вам стоит получить представление о том, что такое пространства имен. Можете спокойно пропустить этот раздел, если вам неинтересно, как устроена абстракция контейнеров в Linux.

Пространство имен — это многозначный термин, которым называют разные вещи в разных областях IT. В контексте контейнерной виртуализации в Linux это понятие проще всего объяснить на примере утилиты `chroot`[1]. Эта старинная программа для Unix и родственных систем позволяет сменить корневой каталог (путь `/`) в файловой системе.

Использовать эту утилиту очень просто: команда `chroot` *`путь/к/каталогу`* назначает указанный путь новым корневым каталогом (`/`). Это не только дает установщикам ОС возможность переключаться в устанавливаемую систему, чтобы запускать команды, но и позволяет наладить элементарные пространства имен. Вообще говоря, этот механизм используется во многих программах и в конфигурации различных дистрибутивов Linux, чтобы усилить безопасность, ведь `chroot` фактически исключает участки файловой системы из текущей области видимости, и все файлы за пределами нового корневого каталога становятся недоступными. Таким образом, если злоумышленник воспользуется уязвимостью, которая позволяет удаленно выполнять код на веб-сервере, работающем в окружении `chroot`, то не пострадают ни сама система, ни любые файлы, находящиеся за пределами этого каталога.

Технические примитивы, с помощью которых реализованы контейнеры в Linux и других операционных системах, существенно изменились за последнее десятилетие и, скорее всего, продолжат меняться. К счастью, низкоуровневая реализация контейнеров не так уж критически важна для вас как для разработчика; она гораздо важнее для операторов виртуализации или для тех, кто создает соответствующие системы.

Абстракция контейнеров опирается на такие технические решения:

- пространства имен в файловой системе (например, настроенные с помощью `chroot`);
- пространства имен для пользователей и процессов; благодаря этим пространствам имен изнутри контейнера нет доступа к процессам, которые находятся

[1] Сокр. *change root* (*сменить корневой каталог*). — *Примеч. пер.*

снаружи. Например, пользователь `root` и процесс с PID 5 в контейнере отображаются на непривилегированного пользователя и процесс с другим идентификатором вне пространства имен контейнера;

- механизмы, которые позволяют группировать ресурсы и отслеживать их потребление, например `cgroups`;
- сетевая виртуализация с пространствами имен, из-за которой контейнер не может напрямую обратиться к сетевому интерфейсу, но зато при этом удается избежать ситуаций, когда номера портов перекрываются. Например, можно запустить два разных контейнера, каждый из которых открывает порт `8080`. Вы не увидите сообщения об ошибке вроде «Этот порт уже используется», потому что сетевые стеки контейнеров не зависят друг от друга.

Как контейнеры участвуют в операциях технического обслуживания

Хотя это не книга для системных администраторов или специалистов по надежности сайтов, вам не помешает знать, как контейнеры вписываются в более общий контекст информационных систем. Основная идея состоит в том, что контейнеры рассматриваются как «функции», которые не сохраняют состояния, обрабатывают входные данные (веб-запросы или сообщения HTTP от других служб) и вырабатывают выходные данные (ответы на веб-запросы, побочные эффекты и протоколы, которые направляются в `stdout`). В грамотно настроенной среде технического обслуживания контейнеры ведут себя примерно так же, как процессы в Linux или функции в языках программирования.

Контейнеры обычно распределяются по узлам с помощью стороннего слоя инструментов, таких как Kubernetes, Nomad и др. Если сравнивать контейнеры с процессами, то эти инструменты играют роль планировщика операционной системы, где в качестве системы выступает распределенная структура, а не отдельный узел.

Те же инструменты обычно перехватывают выходные данные контейнеров и перенаправляют их к средствам протоколирования, таким как Logstash, Graylog и Datadog. Метрики от всех активных контейнеров извлекаются и передаются таким инструментам, как Prometheus, для анализа и устранения неполадок.

Итоги

В этой главе мы обзорно рассмотрели самые важные понятия, которые понадобятся вам, чтобы работать с Docker и контейнерами в целом. Конкретные технологии могут меняться (например, в зависимости от того, какой планировщик

контейнеров сейчас в моде или как лучше всего обрабатывать поток журналов), но мы старались сосредоточиться на базовой теории и практических навыках, которые нужны каждому современному разработчику.

Мы надеемся, что вам удалось извлечь из этой главы несколько ключевых идей. Прежде всего, мы хотели бы, чтобы вы получили общее представление о том, какие проблемы решает контейнерная виртуализация, — в основном благодаря тому, что она позволяет управлять сложностью и инкапсулирует зависимости в единый артефакт.

Также важно понимать, чем образы отличаются от контейнеров, и научиться составлять свои собственные Docker-файлы с нуля с помощью официальной документации.

Мы рассчитываем, что некоторые продвинутые темы — например, чем контейнеры отличаются от виртуальных машин и как устроены пространства имен, — помогут вам устранять неполадки или проходить собеседования. Кроме того, будут полезны передовые методы, которые мы затронули.

Наконец, чтобы закрепить полученные навыки, мы рекомендуем попрактиковаться на материале этой главы и поместить в контейнер одно из ваших собственных приложений. Вы многому научитесь, и вам будет гораздо легче приступить к работе, если эта глава будет свежа у вас в памяти.

16

Журналы приложений

Добро пожаловать в мир журналов Linux! Разработчикам ПО крайне важно разбираться в том, как устроены журналы и протоколирование в Linux, и владеть соответствующими инструментами (в частности, `systemd` и `journald`). Предлагаем вашему вниманию обзор знаний и навыков, которые пригодятся вам в профессиональной деятельности.

Журнал — это протокол событий, которые происходят в приложении или операционной системе. Журналы устроены довольно гибко, и почти каждое приложение ведет их в своем собственном формате, но в современных системах принят более-менее единообразный подход к тому, как их обрабатывать, как хранить и как извлекать из них данные. Вам как разработчику очень важно ориентироваться в журналах, доступных в Linux, потому что они помогают понять, почему операционная система и приложения, которые в ней функционируют, ведут себя тем или иным образом. Это позволит исправлять ошибки, оценивать производительность приложений и отлаживать их. Журналы — это первая линия обороны в области устранения неполадок, так что будьте готовы досконально в них разобраться.

Эта глава посвящена основам журналов и протоколирования в Unix и Linux, и в ней рассказывается, как разработчики ПО чаще всего обращаются с журналами. Вот о чем пойдет речь:

- Как система и приложения под ее управлением формируют журналы.
- Где хранятся журналы в большинстве современных систем Linux.
- Как было устроено протоколирование в прежних системах (это пригодится в работе со многими действующими системами, которые могут вам встретиться).
- Как искать и просматривать журналы, когда вы устраняете неполадки в приложениях.

- Как налажена централизация журналов в корпоративной среде и в приложениях, которые развертываются в облачных средах.

Кроме того, мы дадим несколько советов о том, как извлечь из структурированных журналов максимальную пользу и как избежать распространенных опасностей, которые подстерегают разработчиков.

Введение в протоколирование и журналы в Linux

Как вы уже наверняка знаете, **журналы** — это просто собрания сообщений о событиях, которые происходят в приложении или операционной системе. Как и для большинства понятий Linux, в этой области есть грубое практическое правило: даже если вы пишете сценарий из двух строк, выводящий дату и время в текстовый файл, это можно считать журналом. Ряд журналов — это просто строки в формате обычного текста, которые записываются в файлы в стандартных расположениях, а другие представляют собой хорошо структурированные двоичные данные, которыми управляет специальная служба вроде `systemd`.

Вы как разработчик наверняка знакомы с **уровнями протоколирования**. Это метки, обозначающие, насколько неотложно то или иное событие, которое произошло в вашем программном обеспечении. В своей профессиональной деятельности вы часто натыкаетесь на сообщения с заголовками вроде «Ошибка», «Информация» или «Диагностическое сообщение». Чуть позже мы поговорим об этих стандартных уровнях протоколирования, а сейчас отметим, что вам следует знать о трех основных *источниках* журнальных сообщений в современной полнофункциональной среде Linux: это **системные, служебные и прикладные журналы** (журналы приложений, которые не относятся к службам). По источнику можно судить о контексте, который важен, чтобы правильно понять то или иное сообщение.

Системные журналы порождаются само́й операционной системой (ядром). В них заносятся сообщения об ошибках, о событиях оборудования, потреблении и ограничениях ресурсов, конфигурации и безопасности, а также о существенных изменениях в состоянии системы.

Служебные журналы формируются службами, которые выполняются в системе. В частности, в Linux сообщения для этих журналов поступают от служб, которыми управляет подсистема инициализации `systemd`; она протоколирует события с помощью службы `journald`. Из служебных журналов можно узнать о работоспособности и состоянии различных служб.

В системах, с которыми вам предстоит иметь дело, системными и служебными журналами чаще всего управляет `journald`. Далее в этой главе мы подробнее поговорим о `systemd` и `journald` (а также `journalctl`).

Приложения, которыми не управляет `systemd`, — это аутсайдеры, чьи события обычно не протоколируются с помощью `journald`. Как правило, в их документации написано, где искать их журналы, хотя порядочные приложения обычно хранят журналы в каталоге вроде `/var/log/`*`название_приложения`*.

По мере того как мы будем углубляться в эту главу, вам станет понятно, почему важно разбираться в командах `journald` и `journalctl`. Однако прежде всего стоит упомянуть о некоторых особенностях журналов в Linux.

Журналы в Linux: не все так просто

К этому моменту вы уже убедились, что системы семейства Unix отличаются невероятной гибкостью. Если вам не нравится, как что-нибудь устроено по умолчанию, вы можете пойти своим путем и перенастроить что угодно и как угодно.

Но эта гибкость оборачивается серьезными затруднениями для тех, кто изучает основы Unix и Linux. Дело в том, что многие параметры — от конфигурации программ до свойств пользователя по умолчанию — можно настраивать в нескольких разных местах, и в новой среде часто нет другого способа узнать, какие соглашения здесь приняты, кроме как спросить коллег (а иногда выяснить методом проб и ошибок).

Это более чем отчетливо проявляется в области протоколирования, которую особенно затронули недавние изменения в том, как бизнес строит свою IT-инфраструктуру. Несколько предыдущих десятилетий компании напрямую покупали, конфигурировали и долговременно поддерживали физические серверы, на которые устанавливалась отдельная операционная система. По мере того как рабочая нагрузка переместилась в облако и обычным делом стало множество операционных систем на одном компьютере (благодаря виртуальным машинам) и даже множество сред в одной операционной системе (благодаря контейнерам), традиционные подходы к протоколированию тоже претерпели изменения.

Какой главный вывод можно из этого сделать? Когда вам нужно выяснить, как работать с журналами на новом месте или в новой команде, стоит спрашивать не о том, как устроено протоколирование в Linux в целом, а о том, как оно организовано именно здесь. Все во многом зависит от того, какие решения принимали создатели программ, которые вы используете в работе. Не забывайте об этом, пока будете изучать эту главу.

Как отправлять сообщения в журнал

Чаще всего службы либо регистрируют свои события с помощью специальных библиотек, либо просто выводят сообщения в `stdout`. Впрочем, в системах семейства Unix также предусмотрена команда, которая отправляет сообщения syslog-серверу. (Далее в этой главе мы подробно поговорим о том, что это такое.) Поскольку в любой системе и `syslogd`, и `systemd` обеспечивают syslog-сервер, сообщения в журнал можно отправлять с помощью единой команды `logger`[1]:

```
logger Привет!
```

Эта команда зарегистрирует сообщение `Привет!`. У команды `logger` много параметров, и она бывает чрезвычайно полезна, когда приходится отлаживать код, вносить сообщения в журнал из сценария командной оболочки или объяснять, как устроены журналы.

Журнал подсистемы инициализации systemd

Если используется подсистема инициализации `systemd`, то за ведение журнала отвечает `journald`[2]. Это первое место, где стоит искать журналы, если вы устраняете неполадки на компьютере под управлением Linux, на котором работает `systemd`. По умолчанию служба `journald` регистрирует весь вывод из процессов, находящихся под ее наблюдением. Все, что направляется в `stderr`, считается сообщением об ошибке. Так что если программное обеспечение не настроено так, чтобы выводить журнальные сообщения не в `stderr`/`stdout` (в зависимости от ПО это может быть либо задано с помощью конфигурации, либо жестко закодировано), то события регистрируются в журнале подсистемы `systemd`.

Чтобы извлекать данные из журналов `journald`, используйте команду `journalctl`. Она позволяет ограничивать запросы определенными службами, периодами времени и промежутками между перезагрузками системы, а также поддерживает примерно те же аргументы, что и команда `tail`. Давайте перейдем от теории к практике и посмотрим, как использовать `journalctl`.

---

[1] *регистратор, система протоколирования. — Примеч. пер.*

[2] *Journal — журнал*; *d* — сокр. *daemon*, как и во всех подобных случаях (см. сноску 2 на с. 71). *— Примеч. пер.*

Некоторые команды journalctl

Основы работы с `journalctl` очень просты. Начнем с ответа на такой вопрос: когда вы устраняете неполадки в приложении, какие операции вам нужно совершать с его журналами?

Прежде всего нужно находить и просматривать текущие журналы. С помощью `journalctl` это можно сделать, но вы быстро убедитесь, что на самом деле вам нужны не все журнальные записи, а только несколько самых свежих. Их можно выбрать с помощью флага `-n`.

Например, чтобы просмотреть последние 100 сообщений в `journald`, запустите такую команду:

```
journalctl -n 100
```

Вы наверняка заметили, что это похоже на работу команды `tail`, с которой вы уже встречались в этой книге. Если вы запускали предыдущую команду, то среди выведенных записей будет ваше сообщение `Привет!`.

Как выводить активные записи журнала для модуля в реальном времени

Возможно, вам также понадобится просматривать поступающие журнальные сообщения в режиме реального времени. Например, если отслеживать, какие сообщения ваше приложение порождает во время запуска, это часто помогает выяснить, в какой конкретно момент что-то пошло не так:

```
journalctl -fu имя_модуля
```

Длинная форма флага `-f` — `--follow`, а флага `-u` — `--unit`[1]; этот флаг обозначает модуль (службу), по которому вы хотите отфильтровать журналы.

Как фильтровать журналы по времени

Даже если вы сузили выборку из журналов до отдельного модуля, который вас интересует, в ней все равно может оказаться слишком много сообщений. Здесь может помочь фильтр по времени, особенно если вы пытаетесь найти связь между известной внешней проблемой (аварией, ошибкой сети и т. д.) и журнала-

[1] Параметр `--unit` употребляется со знаком равенства, например: `--unit=docker.service`. — *Примеч. науч. ред.*

ми приложений непосредственно до или непосредственно после сбоя. Здесь вам пригодятся параметры `--since`[1] и `--until`[2]:

```
journalctl --since "2021-01-01 00:00:00"
```

Некоторые моменты времени можно указывать в сокращенной записи: например, `today`[3]:

```
journalctl --until today
```

Параметры `--since` и `--until` можно совмещать, чтобы узко фильтровать результаты, например:

```
journalctl --since "2021-01-01 00:00:00" --until "1 hour ago"
```

ПРИМЕЧАНИЕ

Когда вы просматриваете временны́е метки в журналах или фильтруете записи по времени, почти всегда имеет смысл запускать команду `journalctl` с параметром `--utc`, благодаря которому дата и время выводятся в UTC. Если вы помогаете специалистам технической поддержки устранять последствия аварии, вы почти всегда оперируете временем по UTC, чтобы не возникало путаницы с часовыми поясами.

Журналы можно фильтровать и по таким дополнительным параметрам, как идентификаторы пользователей и групп.

Как фильтровать журналы по определенному уровню протоколирования

Если вы ищете именно ошибку, то можете настроить `journalctl` так, чтобы отображались только ошибки:

```
journalctl -p[4] err
```

Можно указывать и другие уровни протоколирования; все они перечислены на странице wiki.archlinux.org/title/Systemd/Journal#Priority_level.

---

[1] *Начиная с* (такого-то времени). — *Примеч. пер.*

[2] *До* (такого-то времени). — *Примеч. пер.*

[3] *Сегодня*. — *Примеч. пер.*

[4] Длинная форма — `--priority` (*приоритет*). — *Примеч. пер.*

Как анализировать журналы от предыдущих загрузок

Иногда ситуация выходит из-под контроля, отказывают критически важные компоненты, и из-за этого система перезагружается. В таких случаях часто требуется просматривать журналы от предыдущих загрузок. Параметр `--list-boots`[1] позволяет вывести список всех доступных загрузок:

```
journalctl --list-boots
```

Затем с помощью флага `-b`[2] можно выбрать из списка конкретную загрузку. Допустим, нас интересует загрузка с меткой `2`:

```
journalctl -b -2
```

Флаг `-b` без метки означает текущую загрузку системы.

Сообщения ядра

Во введении мы упомянули, что журнальные сообщения уровня системы порождает операционная система (*ядро* в терминологии Linux). Чтобы просматривать только такие сообщения, используйте флаг `-k`[3] (по историческим причинам его длинная форма — `--dmesg`).

Журналы в контейнерах Docker

Стандартный подход к журналам в контейнерах Docker состоит в том, чтобы просто предполагать, что нас интересует вывод главного процесса контейнера и что он выводит журнальные сообщения в стандартный поток вывода (`stdout`). Системы координации контейнеров (Kubernetes, Nomad и др.), а также различные облачные службы, которые управляют контейнерами, по умолчанию воспринимают `stdout` как поток соответствующих журнальных сообщений и в зависимости от конфигурации перенаправляют его в тот или иной приемник. Мы поговорим об этом подробнее чуть дальше в разделе «Централизованные журналы».

---

[1] *Перечислить загрузки. — Примеч. пер.*

[2] Длинная форма — `--boot` (*загрузка*). *— Примеч. пер.*

[3] Сокр. *kernel* (*ядро*). *— Примеч. пер.*

Введение в Syslog

Протокол syslog[1] может показаться немного архаичным по сравнению с `systemd/journald`, о которых шла речь раньше. Syslog отличается богатой историей: хотя он существует с 1980-х годов, он остается практичным, гибким и широко востребованным средством ведения журналов. Что еще более важно, вы почти наверняка встретитесь с syslog в реальных средах эксплуатации, так что стоит знать его основы, чтобы не растеряться, если придется устранять аварию, когда время играет против вас.

В системах семейства Unix протоколирование syslog часто сводится к тому, что сообщения записываются в файл в каталоге `/var/log`, причем большинство из них — в `/var/log/messages`. Однако имейте в виду: не все, что находится в `/var/log`, поступает туда через syslog. Многие программные компоненты реализуют свои собственные способы протоколирования, вообще не задействуя службу syslog.

Служба syslog принимает все сообщения, которые ей направлены, и записывает их в тот или иной файл в зависимости от различных параметров, — например, от категории сообщения, которые будут перечислены ниже. Почти во всех системах это файл `/var/log/messages`, и если вы практиковались по ходу этой главы, то именно в нем вы увидите сообщение `Привет!`, которое создавали выше.

Syslog — это стандартный протокол для ведения журналов. Хотя на момент написания этой книги соответствующая служба `syslogd` в основном оперирует строковыми сообщениями, текущий стандарт RFC 5424 также предусматривает структурированное протоколирование. Впрочем, оно не так широко поддерживается, поэтому мы ограничимся тем, что вкратце обсудим основы этого протокола в контексте строкоориентированных журналов. Если вы вовсе не работаете с syslog, можете спокойно пропустить этот раздел.

Как уже упоминалось, syslog — это протокол. Хотя чаще всего его применяют приложения (например, базы данных, которые ведут локальные журналы), в средах эксплуатации обычно тоже бывает центральный сервер журналов, куда направляются сообщения. Многие программные продукты, например PostgreSQL или nginx, умеют передавать сообщения по протоколу syslog, а различные механизмы журналов — например, Logstash, Loki, syslogd и syslog-ng — умеют принимать эти сообщения. Обычно для этого используется порт 514 (UDP) или 6514 (TCP).

---

[1] Сокр. *system log* (*системный журнал*). — *Примеч. пер.*

Категории

Поскольку syslog — это очень старый протокол из 1980-х, некоторые его особенности могут выглядеть архаичными. Например, он использует стандартный набор категорий (субъектов), которые обозначают, откуда поступило журнальное сообщение. У каждой категории есть свой код:

Код	Название	Описание
0	`kern`	Сообщения ядра
1	`user`	Сообщения уровня пользователя; их часто порождают процессы
2	`mail`	Почтовая система. Используется в основном для почтовых серверов, SMTP, IMAP и POP3. В этой категории также ведут журналы службы и программы, которые связаны со спамом
3	`daemon`	Системные службы (демоны), особенно те, которые относятся к операционной системе (например, NTP)
4	`auth`	Сообщения безопасности и аутентификации. В этой категории обычно отражаются попытки входа (например, локально или по SSH), но бывают и сообщения от многих других служб
5	`syslog`	Сообщения, которые генерирует syslog на внутреннем уровне. Обычно они относятся к самому syslog
6	`lpr`	Подсистема линейного принтера. Сообщения, связанные с принтером
7	`news`	Подсистема сетевых новостей. Историческая категория, сейчас практически не используется
8	`uucp`	Подсистема UUCP. Историческая категория, сейчас практически не используется
9	`cron`	Подсистема cron: сообщения, относящиеся к заданиям cron. Эта категория бывает очень полезна, когда приходится устранять неполадки в заданиях cron
10	`authpriv`	Сообщения безопасности и аутентификации. Эта категория подобна `auth`, но предназначена для того, чтобы передавать сообщения более ограниченному кругу получателей. Большинство программ в Linux ведут журналы здесь, а не в `auth`
11	`ftp`	Служба FTP; в основном историческая категория. Протоколирует взаимодействие с серверами FTP
12	`ntp`	Подсистема NTP; протоколирует работу протокола NTP, с помощью которого синхронизируется время
13	`security`	Журнал аудита; протоколирует события, которые связаны с безопасностью

Код	Название	Описание
14	`console`	Журнал консоли; протоколирует события, которые относятся к локальной консоли
15	`solaris-cron`	Служба системных часов
16–23	`local0` — `local7`	Локальные категории, то есть локальное ПО. Например, если PostgreSQL записывает в syslog, она по умолчанию заносит сообщения в `local0`

Во многих системах в каталоге `/var/log` бывают файлы, имена которых соответствуют этим категориям. Например, если вам понадобится устранять неполадки в заданиях cron в системе, где не используется `journald`, то нужные журналы вы, скорее всего, найдете в файлах вроде `/var/log/cron`, `/var/cron/log`, `/var/log/messages` и т. п.

Обратите внимание, что нет единого стандарта того, какие именно события нужно протоколировать в каждой категории. Скорее всего, вы встретитесь с ситуациями, когда разные операционные системы или программы похожего назначения расходятся во мнениях о том, в какую категорию заносить журнальные сообщения.

Уровни важности сообщений

Возможно, это понятие вам знакомо лучше. Журнальным сообщениям присваиваются уровни важности по шкале от 0 (самый важный) до 7 (наименее важный):

Код	Аналогичный параметр journalctl	Название	Описание
`0`	`emerg`	Emergency	Система неработоспособна
`1`	`alert`	Alert	Отказ важного компонента и/или потеря данных
`2`	`crit`	Critical	Различные отказы и сбои
`3`	`err`	Error	Обычные ошибки
`4`	`warning`	Warning	Предупреждения
`5`	`notice`	Notice	Необычные события
`6`	`info`	Informational	Штатные события
`7`	`debug`	Debug	Диагностические события

Как и в случае с категориями, от конкретного программного обеспечения зависит, к какому уровню важности относится то или иное сообщение.

Конфигурация и реализации

У syslog есть много реализаций, которые обычно позволяют настраивать фильтрацию, а также сохранять и пересылать сообщения в зависимости от категории и уровня важности. Некоторые реализации поставляются со службами, такими как `syslogd`, `rsyslog` или `syslog-ng`, которые можно настраивать с помощью конфигурационных файлов в каталогах `/etc/syslog.conf` или `/etc/syslog-ng`. В Loki, Logstash и других системах распределенного управления журналами протоколирование конфигурируется по-своему — обычно с помощью трехуровневой структуры, где на одном уровне поступают входные данные, на другом они фильтруются и преобразовываются, а на третьем программа сохраняет или пересылает выходные данные.

Как вести журналы

Разные программные продукты ведут журналы по-разному, и принципы того, какое протоколирование считать «правильным» или «неправильным», во многом зависят от конкретного проекта, а также меняются со временем. Однако в большинстве случаев следует учитывать общие соображения, которые мы сейчас рассмотрим.

Тщательно выбирайте ключевые слова при структурированном протоколировании

Если ваше программное обеспечение порождает структурированные журналы, постарайтесь не только добиться того, чтобы для стандартных понятий, например таких, как запросы или идентификаторы пользователей, использовались привычные ключевые слова, но и избежать коллизий, когда одними и теми же ключевыми словами обозначаются похожие, но все-таки не одинаковые понятия. В зависимости от того, какую базу данных вы используете, вы также можете наткнуться на проблемы с типами, например, в ситуации, когда идентификатор `user` может быть ключом для числового поля, строкового поля или специально созданной структуры вроде объекта JSON.

Чтобы избежать коллизий, иногда удается создавать отдельное пространство имен для каждой службы, а также поддерживать список «глобальных» ключей вместе с их определениями.

Уровень важности

Когда вы разрабатываете ПО, имеет смысл составить внутренний документ о том, по каким критериям присваивать сообщению тот или иной уровень важности. Это поможет избегать ситуаций, когда неудачные попытки входа на общедоступный сайт или ошибки 404, которые возникают из-за того, что поисковые роботы запрашивают несуществующие ресурсы, вызывают сигнал тревоги и будят ваших коллег посреди ночи. Но даже если таких неприятностей не происходит, то вам будет гораздо проще отлаживать код и выявлять проблемы, если вы будете присваивать уровни важности по единой логически выстроенной схеме.

Имеет смысл четко различать такие варианты:

- ситуации, которые *могут* указывать на проблему;
- ситуации, которые не должны происходить, но *могут* произойти;
- ситуации, которые заведомо указывают на дефект или более серьезную проблему.

По мере того как ПО разрастается и к нему добавляются новые службы, журналы склонны усложняться, так что вы не ошибетесь, если с самого начала выработаете четкие правила о том, какие события и когда протоколировать.

Централизованные журналы

В корпоративной среде принято работать с журналами централизованно, что помогает разбираться в проблемных ситуациях. Кроме того, при этом не требуется по отдельности изучать каждый журнал на каждом физическом компьютере, в каждой виртуальной машине и в каждом контейнере. Централизованные службы управления журналами обычно позволяют просто и быстро анализировать огромные объемы журналов, особенно если они ведутся структурированно, а отдельные службы согласованы с единой схемой журналов.

Эти службы управления журналами могут быть либо самостоятельными продуктами, такими как Loki, стек ELK (Elastic Search, Logstash и Kibana) и Graylog, а могут быть управляемыми службами, например, если это версия одного из упомянутых продуктов, которая развернута на хостинге, или если это специализированное облачное решение, например Google Cloud Operations (ранее известное как Stackdriver), AWS CloudWatch или Azure Monitor. Эти системы во многом похожи: они умеют извлекать журналы из файлов либо извлекать их с помощью того или иного API, фильтровать и реорганизовывать их и в итоге сохранять в конечное хранилище, где к ним можно обращаться с запросами.

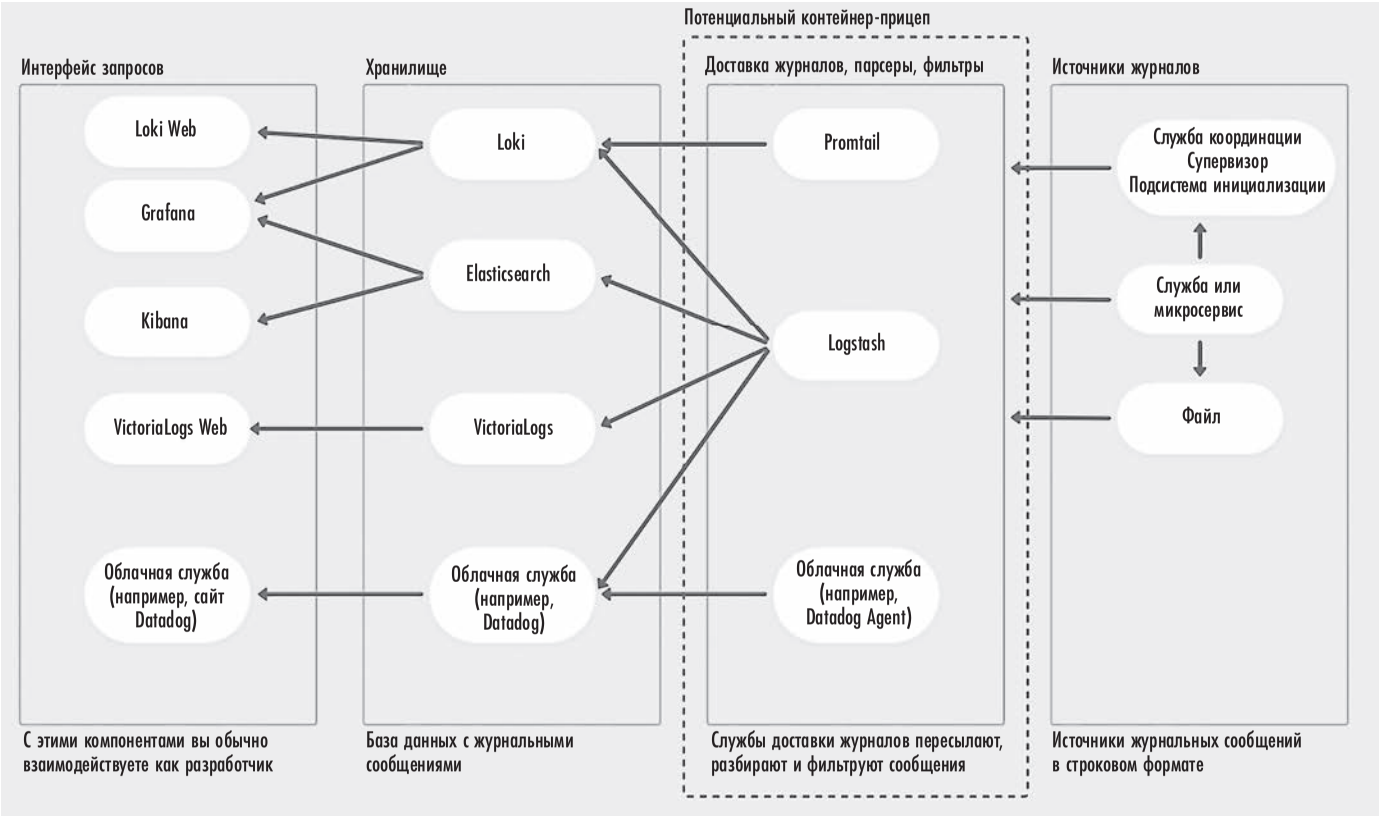


Рис. 16.1. Координация контейнеров с sidecar-контейнером

В микросервисной архитектуре вместе с сообщениями обычно передается контекст, например идентификатор запроса, чтобы можно было легко проследить путь запроса от клиента через различные службы, а это крайне важно для отладки в архитектурах с большим количеством служб.

Как уже упоминалось, у большинства этих систем есть механизмы для того, чтобы передавать журналы центральному серверу или кластеру журналов. К таким продуктам относятся, например, Logstash (от которого происходит буква L в названии стека ELK) и Promtail (который используется вместе с Loki); обычно эти системы умеют получать журналы множеством разных способов. Например, их можно настроить так, чтобы они функционировали как сервер HTTP или syslog-сервер, подключались к `journald`, извлекали данные из любых других служб доставки журналов, обращались к облачным API или просто применяли команду `tail` к файлам. Они запускаются либо в качестве дополнительных системных служб (как, например, Pod в Kubernetes), либо как часть конфигурации Nomad. Поскольку предназначение этих инструментов — централизовать журналы независимо от того, какое программное обеспечение используется, они обычно умеют принимать самые разные входные данные и чрезвычайно гибко настраиваются: например, с их помощью можно создать иерархию, пересылая данные от одних служб доставки журналов к другим. В системах координации контейнеров, таких как Kubernetes или Nomad, эту структуру часто реализуют с помощью sidecar-контейнера, который выполняется рядом с контейнерами ваших приложений и регистрирует журналы из всех контейнеров в соответствующем поде (размещении, узле) перед тем, как доставлять их по назначению.

Как видите, хотя с протоколированием связано множество технологий и продуктов, все они попадают в одну из категорий, описанных выше, так что если вы налаживаете централизованные журналы в своей среде, эта схема поможет вам понять, как все эти компоненты связаны друг с другом.

Итоги

В области протоколирования и журналов в современных средах эксплуатации постоянно происходят изменения. Эта глава была рассчитана на то, чтобы снабдить вас необходимыми знаниями и навыками, благодаря которым вы сможете ориентироваться в этой теме. Мы также надеемся, что знакомство с `syslog` и `journalctl` поможет вам понять низкоуровневые механизмы и направление развития журналов в Linux, так что вам будет проще разобраться в том, как на самом деле работают перспективные решения класса «журналы как служба».

Хотелось бы верить, что вы почувствовали, как навыки, которыми вы овладели в этой главе, обеспечили вам практические, измеримые преимущества в профес-

сиональной деятельности, когда вы разрабатываете, отлаживаете и оптимизируете приложения, чтобы развертывать их. Вы убедились, что стоит овладеть основами `journald`, чтобы быстро выявлять и локализовывать неполадки, независимо от того, относятся ли они именно к вашему приложению или к окружающей системе Linux. А если вы разбираетесь в альтернативных и устаревших механизмах протоколирования в Linux, это может пригодиться, когда вам придется устранять неполадки в системах, которые давно не обновлялись.

Материал этой главы не только поможет решать проблемы, но и облегчит вашу жизнь и позволит вам работать эффективнее. Кроме того, с этими навыками вам будет проще выделиться на рынке труда. Проще говоря, если вы разбираетесь в журналах Linux, вы становитесь более квалифицированным и эффективным разработчиком.

17

Балансировка нагрузки и протокол HTTP[1]

В этой главе мы зайдем немного с другой стороны, так что, как говорится, держитесь за поручень. Во-первых, мы совершим небольшой экскурс в историю протокола HTTP и развеем несколько заблуждений, от которых страдают многие современные веб-разработчики.

Во-вторых, мы продолжим делать акцент на практические навыки и рассмотрим `curl` — один из самых популярных стандартных инструментов для работы с HTTP из командной строки. В частности, вы познакомитесь с основами `curl` в контексте того, как с его помощью устранять распространенные неполадки веб-приложений.

Мы предполагаем, что если вы — веб-разработчик, то уже имеете представление об HTTP. Так что хотя в этой главе у нас нет задачи обучить вас элементарным понятиям, касающимся этого протокола, мы все-таки вкратце затронем некоторые из этих понятий, чтобы освежить ваши знания, если вдруг вы давно не практиковались. А если у вас нет *вообще* никакого опыта работы с HTTP, то обратите внимание, что в интернете имеется множество прекрасных ресурсов с документацией вам в помощь. В частности, мы весьма рекомендуем MDN от компании Mozilla (developer.mozilla.org/docs/Web/HTTP) как достойный источник информации.

Мы больше сосредоточимся не на основах протокола, а на распространенных заблуждениях и подводных камнях, которые связаны с HTTP и тем, как он применяется в реальной практике; эти темы часто оказываются неожиданностью даже

---

[1] Аббр. *HyperText Transfer Protocol* (*протокол передачи гипертекста*). Несмотря на название, с помощью HTTP передаются не только такие данные, которые можно назвать «гипертекстом», но и данные любой другой природы. — *Примеч. науч. ред.*

для действующих разработчиков. Заблуждения в основном происходят из-за того, что разработчики создают веб-приложения в довольно простой локальной среде, но затем запускают их в сложных продакшен-средах, которые существенно отличаются от компьютера, где разрабатывалось и тестировалось приложение.

Различия между условиями, в которых приложение функционирует на стадии разработки на локальном компьютере, и инфраструктурой, которая окружает приложение, когда оно развернуто в предпродакшен или продакшен-среде, порождают немало путаницы и неочевидных дефектов.

В этой главе мы поговорим о самых важных различиях такого рода: вы узнаете о шлюзах, восходящих узлах и других понятиях, которые играют роль в инфраструктуре, окружающей современный сайт или веб-приложение. Затем мы уделим внимание самым распространенным ошибкам из области HTTP, которые виноваты в трудноотлаживаемых проблемах с заголовками, кодами состояния и др. Также речь пойдет о современных механизмах безопасности, которые добавились относительно недавно (в частности, CORS), и об истории HTTP с обзором версий. Наконец, вы узнаете кое-что о том, как балансировать нагрузку: это поможет вам избежать традиционного заблуждения части разработчиков, в голове у которых формируется неверная модель того, по какому маршруту следует запрос пользователя, и эта неверная модель часто приводит к ошибкам проектирования на стыке приложения и инфраструктуры.

Вот какие темы рассматриваются в этой главе:

- Ряд элементарных понятий, необходимых для того, чтобы ориентироваться в более сложной веб-инфраструктуре, о которой мы поговорим позже в этой главе.
- Распространенные заблуждения о состояниях HTTP, которые стоит убедительно развеять, чтобы ваш код стал чище и смог более эффективно обрабатывать разные состояния.
- Заголовки HTTP и сопутствующие проблемы, с которыми вы можете встретиться в своих веб-приложениях.
- С какими версиями протокола HTTP можно столкнуться в реальной практике.
- Как устроена балансировка нагрузки и почему вам как разработчику важно ее понимать, даже если вы не собираетесь заниматься инфраструктурой приложений.
- Как из командной строки с помощью инструмента `curl` устранять неполадки, которые относятся ко всем перечисленным темам.

Единственное, что вам понадобится, чтобы освоить эту главу, — это хотя бы в общих чертах понимать, как работают запросы HTTP и какие существуют инструменты разработчика веб-приложений: например, вам стоит уметь по крайней мере

на начальном уровне работать с консолью браузера и другими средствами разработчика, чтобы устранять простейшие неполадки HTTP.

Давайте начнем с основных понятий, которые вам пригодятся, чтобы решать проблемы с веб-приложениями.

Основные понятия

В следующих разделах будут фигурировать несколько понятий, которые могут оказаться для вас в новинку, так что давайте кратко рассмотрим их здесь.

Шлюз

В современном мире шлюз — это обычно или обратный прокси-сервер HTTP, или балансировщик нагрузки, а зачастую сочетание обоих компонентов. Это может быть сервер HTTP, такой как nginx или Apache, физический балансировщик в классическом смысле или облачная реализация этого же принципа. Кроме того, в качестве шлюза может выступать **CDN**[1]. Таким образом, если код состояния HTTP сообщает об ошибке, которая связана со шлюзом, это значит, что с вами взаимодействует какое-то из перечисленных устройств или приложений.

Восходящий узел

Восходящий узел приложения — это служба, по отношению к которой оно выступает как прокси-узел. Чаще всего это настоящее приложение или служба, например служба HTTP, которую вы разработали. Полезно помнить о том, что прокси-серверы могут выстраиваться по каскадной схеме, так что между первым прокси-сервером и настоящим веб-приложением может располагаться еще один промежуточный прокси-сервер. Например, во многих облачных инфраструктурах присутствует входной балансировщик нагрузки, который обрабатывает и фильтрует входящий трафик, а за ним расположен балансировщик нагрузки приложения, который уже на самом деле анализирует трафик HTTP и маршрутизирует его на нужный серверный пул приложения.

После того как мы разобрали несколько понятий, выходящих за рамки HTTP, вы готовы осваивать следующие разделы этой главы. Давайте подробнее рассмотрим распространенные заблуждения, которые относятся к HTTP, и начнем применять `curl`, чтобы научиться работать с популярными командами оболочки для устранения неполадок.

---

[1] Аббр. *Content Delivery Network* (*сеть доставки контента*) или *Content Distribution Network* (*сеть распространения контента*). — *Примеч. пер.*

Распространенные заблуждения об HTTP

Если вы разрабатываете веб-приложения и API для HTTP, вам стоит вникнуть в определенные тонкости, которые многие специалисты упускают из виду. Давайте рассмотрим несколько ситуаций, где действительно имеет смысл разбираться в предмете чуть глубже, чтобы создавать более надежные приложения. А навыки работы с `curl`, которые мы рассмотрим в этой главе, позволят вам устранять из командной строки даже расплывчато сформулированные неполадки вроде «сайт не открывается».

Состояния HTTP

В этом разделе перечислены распространенные коды состояний HTTP. Мы также представим избранные важные сведения об этих состояниях и развеем мифы, которые могут сбить вас с толку.

Не ограничивайтесь проверкой на 200 OK

Популярный способ убедиться, что запрос завершился успешно и ошибок нет, — проверять, что код состояния равен 200 или даже просто входит в диапазон кодов 2xx. Однако здесь есть несколько нюансов.

Коды состояния в диапазоне 2xx (то есть от 200 до 299 включительно) предназначены для того, чтобы сигнализировать об успешной операции, и многие API возвращают код 204 No Content[1] как признак успеха. Это особенно характерно, когда этот API обычно предоставляет ресурс, который был создан или изменен, а в данном случае этого не происходит (например, при запросах типа `DELETE`).

Для некоторых приложений вполне достаточно проверять, что код ответа находится в диапазоне 2xx. Однако важно понимать, что если приложение придерживается логики вроде «если не 200, выводить в журнал сообщение об ошибке» — это неверный подход. Коды в диапазонах 1xx и 3xx тоже не обозначают ошибки, хотя они и не относятся к 2xx.

Довольно редко приходится наблюдать коды в диапазоне 1xx, если вы не ожидаете их специально, потому что в большинстве случаев «сотые» коды обозначают операции вроде переключения на WebSockets, но от этого они не становятся ошибками. Сервер часто возвращает коды 3xx, чтобы уведомить клиентское ПО о переадресации. Даже если эти коды указывают на то, что на стороне клиента нужно что-то сделать, например обновить адрес ресурса, который был перемещен, это определенно не ошибка как таковая.

---

[1] *Нет содержимого. — Примеч. пер.*

Один из кодов состояния в диапазоне 3xx, который гораздо чаще наблюдается в среде эксплуатации, чем в среде разработки, — это 304 Not Modified[1]. Его легко упустить из виду во время разработки, и он также может возникать из-за того, что изменилась инфраструктура или обновились библиотеки так, чтобы внедрить новые или улучшенные механизмы кэширования.

С учетом сказанного обычно имеет смысл вместо того, чтобы считать успехом только коды 2xx, использовать другой подход: рассматривать как потенциальные ошибки только коды 400 и более. Это не усложняет внутреннюю логику приложения: проверять, что код больше или равен 400, ничуть не менее элегантно, чем проверять, что он находится в диапазоне 2xx.

404 Not Found[2]

Важно не забывать о том, что код состояния 404 Not Found может означать разные вещи в зависимости от приложения. Этот код могут возвращать серверы (в том числе файловые) и шлюзы, но может вернуть и само приложение. Он может означать как то, что не существует маршрута к запрошенному ресурсу, так и то, что не существует самого́ ресурса (например, это пост или комментарий в социальной сети, который был удален).

Вот почему 404 вполне может быть штатным ответом от исправного приложения, которое работает так, как было спроектировано. В некоторых случаях поведение клиента даже может зависеть от этого ответа: например, если нужно убедиться, что тот или иной ресурс *не* существует (в частности, перед тем, как создавать этот ресурс или чтобы уведомить пользователя, что ресурс или его имя уже заняты).

Иными словами, самого по себе кода 404 (без дополнительных сведений о приложении и запросе) недостаточно, чтобы объявлять о проблеме. Как мы только что продемонстрировали, этот код может даже обозначать успех на нескольких уровнях. Этого иногда избегают посредством того, что приложение не применяет этот код на своем уровне и по-другому сигнализирует о том, что ресурс не найден, например в этом случае все-таки возвращает код состояния 200. Какое решение лучше, зависит как от приложения, так и от того, о чем договорились разработчики, а также от того, что предписывают стандарты и какова архитектура вашего приложения.

502 Bad Gateway[3]

Этот код состояния означает, что шлюз не смог интерпретировать данные, которые вернул восходящий узел. Другими словами, шлюз переадресовал запрос и получил

---

[1] *Нет изменений. — Примеч. пер.*

[2] *Ресурс не найден. — Примеч. пер.*

[3] *Ошибка шлюза. — Примеч. пер.*

ответ, который не является полным и действительным ответом HTTP. Это обычно говорит о том, что со службой на восходящем узле что-то не в порядке.

503 Service Unavailable[1]

Код 503 обычно говорит о том, что к восходящему узлу не удалось обратиться по тому порту, на который настроен шлюз. На практике это может означать, что веб-приложение отказало, или что оно не прослушивает запросы HTTP, или что оно прослушивает не тот порт, или что его блокирует какое-то правило брандмауэра или неправильное правило маршрутизации, — но может быть еще миллион других причин.

504 Gateway Timeout[2]

Если шлюз подключается к восходящему серверу, то в какой-то момент истекает время, которое выделено на это соединение. Это важно, потому что зависшие процессы расходуют ресурсы и на стороне шлюза, и на стороне службы. Как правило, такие тайм-ауты случаются только тогда, когда этого не ожидалось, и при этом никакие данные не читаются и не записываются.

Если восходящий шлюз обрабатывает долговременный запрос, который задерживается потому, что ждет результата вычислений или еще чего-то подобного, можно увеличить предельное время, в течение которого запрос может обрабатываться. Однако обычно более эффективны другие решения: например, обрабатывать запрос асинхронно или начинать записывать данные раньше (как в случае потоковых данных).

В пользу этих решений говорит тот аргумент, что если задано предельное время ожидания, то пока оно не истечет, обе стороны расходуют ресурсы, причем запрашивающая сторона даже не знает, собирается ли приложение вообще возвращать ответ. Так что если веб-служба неисправна, то ни шлюз, ни клиент об этом не узнают. В результате шлюз может так и продолжать считать, будто эта служба по-прежнему благополучно работает, вместо того чтобы быстро обнаружить проблему и переключиться на другой экземпляр службы.

Введение в curl: как проверить состояние ответа HTTP

Если бы вам потребовалось изучить всего одну команду, с помощью которой можно устранять неполадки HTTP, — выбирайте `curl`. По мере того как мы будем рассматривать особенности HTTP, в которых стоит разобраться подробнее, мы будем добавлять разделы вроде этого, чтобы продемонстрировать практические команды `curl`, имеющие отношение к текущей теме.

---

[1] *Служба недоступна. — Примеч. пер.*

[2] *Превышено время ожидания ответа от шлюза. — Примеч. пер.*

Простейший вызов `curl` выглядит примерно так:

```
curl https://tutorialinux.com/
```

Это почти то же самое, что вставить URL в адресную строку браузера, — за исключением того, что эта команда не задействует браузер, а выводит ответ сервера непосредственно в командную строку. Мы согласны, что это не самый увлекательный способ посещать сайты, так что давайте сделаем что-нибудь более полезное для вашего следующего сценария: проверим состояние ответа HTTP!

С помощью `curl` можно легко проверить, что веб-сервер функционирует и отвечает на запросы:

```
curl -IsS https://tutorialinux.com/ | head -n 1
HTTP/2 200
```

Из этого вывода можно заключить, что веб-сервер работает и путь `/` на нем отвечает с кодом состояния 200 OK. Также здесь отображается версия протокола HTTP/2, о которой мы поговорим позже. Если конкретнее, то эта команда отправляет запрос `HEAD` (для этого служит флаг `-I` или `--head`) и подавляет сообщения о ходе запроса и об ошибках (флаг `-s` или `--silent`).

Тем не менее, если что-то пойдет не так, вы все-таки увидите сообщения об ошибках благодаря флагу `-S` (`--show-error`):

```
curl -IsS https://tutoriajsdkfksjdhfkjshdflinux.com/ | head -n 1
curl: (6) Could not resolve host[1]: tutoriajsdkfksjdhfkjshdflinux.com
```

Наконец, чтобы отбросить заголовки ответа HTTP и вывести только код состояния (первую строку ответа), мы используем конвейер: `| head -n 1`.

Впрочем, чтобы устранять неполадки, часто требуется изучить заголовки. Давайте обсудим несколько особенностей заголовков HTTP, а затем попробуем проанализировать их с помощью `curl`.

Заголовки HTTP

Регистр не важен

Заголовки HTTP не зависят от регистра символов. Некоторые программы учитывают эту особенность, в результате чего определенные шлюзы могут модифицировать и «нормализовывать» эти заголовки. К счастью, когда разработчики пишут веб-приложения, им редко приходится иметь дело непосредственно с необработанными заголовками. Гораздо чаще они пользуются веб-библиотеками, которые абстрагируют бóльшую часть этой сложности и сами управляются с фор-

[1] *Не удается разрешить имя узла. — Примеч. пер.*

матом заголовков. Тем не менее вам все равно имеет смысл проверить и нормализовать заголовки, например перевести в нижний регистр все заголовки, которые формирует ваше приложение. Так можно избежать ситуаций, когда одни и те же заголовки ответа по ошибке добавляются по несколько раз с разным регистром символов.

Пользовательские заголовки

Если ваше приложение формирует свои собственные заголовки HTTP, обратите внимание на то, каким префиксом вы их помечаете. Раньше пользовательские заголовки часто начинались с префикса `X-`, например `X-My-Header`. Сейчас это считается неудачной практикой — см. RFC 6648, который объявляет такие префиксы устаревшими. Вместо этого рекомендуется придумать собственный префикс, например название проекта, продукта, компании или аббревиатуру из этих названий. Это поможет предотвратить ситуации, когда ваш заголовок будут использовать другие разработчики, ошибочно принимая его за официальную часть стандарта HTTP.

Как просматривать заголовки HTTP с помощью curl

Флаг `-I`, который мы представили в предыдущем примере использования `curl`, позволяет просматривать заголовки ответа, благодаря чему удается выявить проблемы кэша, несоответствие типов содержимого и другие проблемы. Посмотрим, что сервер `tutorialinux` скажет своими заголовками:

```
curl -I https://tutorialinux.com/

HTTP/2 200
server: nginx/1.24.0
date: Sat, 21 Oct 2023 16:37:12 GMT
content-type: text/html; charset=UTF-8
vary: Accept-Encoding
x-powered-by: PHP/8.2.11
link: <https://tutorialinux.com/wp-json/>; rel="https://api.w.org/"
strict-transport-security: max-age=31536000; includeSubdomains; preload
```

На сервере не обнаружено никаких проблем, хотя эти заголовки дают нам повод задуматься о том, как можно улучшить конфигурацию nginx. Например, демонстрировать названия программных продуктов и номера версий — это неудачная идея с точки зрения безопасности, и никому, кто принимает с этого сервера данные в формате HTML или JSON, не стоит знать о том, что на бэкенде работает PHP.

Версии HTTP

Чтобы объяснить новые возможности HTTP, которые вы скоро увидите на практике, мы обзорно представим вам историю этого протокола. HTTP существует

довольно давно и в значительной мере эволюционировал за это время, особенно в последние годы, когда в моду вошли веб-приложения. Основные принципы и примитивы в целом остались такими же, как на заре HTTP, но некоторые приемы, методы оптимизации и особенности поведения заметно изменились. Если вы знаете, с какой версией протокола работаете, то сможете эффективнее устранять неполадки или предотвращать их, а также уменьшить ненужную или контрпродуктивную оптимизацию и избавиться от «костылей».

HTTP/0.9

Вряд ли вы когда-нибудь встретите эту версию HTTP. Это самый минималистичный HTTP, который только можно себе представить. HTTP/0.9 позволял отправлять серверу запросы GET и принимать то, что сейчас называется *телом* ответа HTTP. Протокол не отправлял и не возвращал никаких заголовков — даже версии или кода состояния.

HTTP/1.0 и HTTP/1.1

Версии протокола HTTP/1.0 и HTTP/1.1 были уже гораздо ближе к тому, что современные разработчики понимают под HTTP. HTTP/1.0 ввел номер версии и заголовки, а HTTP/1.1 представил различные методы и немало расширений, в основном в виде заголовков.

В HTTP/1.1 также появился режим конвейерной обработки, который стал применяться по умолчанию. В этом режиме множественные запросы можно передавать по одному и тому же соединению TCP. Еще одно важное дополнение — заголовок Host, благодаря которому для одного и того же сервера или IP-адреса можно использовать несколько доменных имен.

Например, с HTTP/1.1 веб-сервер можно настроить так, чтобы он откликался на запросы по именам http://example.org/, http://www.example.org/, http://forum.example.org/ и http://blog.example.org/ без необходимости заводить отдельный IP-адрес для каждого из этих имен.

HTTP/1.1 также внедрил множество расширений: кэширование, сжатие, различные механизмы аутентификации, согласование содержимого и даже такие технологии, как WebSockets. Все эти возможности до сих пор широко используются в интернете.

HTTP/2

Вы наверняка видели множество статей, которые восхваляют достоинства HTTP/2. Эта версия стала огромным и неоднозначным шагом в новом направлении развития HTTP. В то время как HTTP/1.1 был текстовым протоколом, который давал возможность всем желающим создавать полноценные и действительные

запросы в терминале или текстовом редакторе, HTTP/2 функционирует в двоичном формате, а также работает с потоками данных, позволяющими наладить облегченный вариант соединения TCP.

Двоичный формат и сжатие заголовков означают, что теперь нужны специальные инструменты для того, чтобы взаимодействовать с сервером или клиентом HTTP. Однако общие принципы остались такими же, как в прежних версиях HTTP, так что вы как разработчик сможете заметить разницу только в особых ситуациях.

Несмотря на то что в HTTP/2 добавилось немало абсолютно новых возможностей, многие из них редко задействуются в веб-приложениях с пользовательским интерфейсом и даже не всегда поддерживаются в браузерах.

В браузерах HTTP/2 обычно работает только с HTTPS, хотя официальный стандарт данной версии протокола этого не требует.

В большинстве случаев веб-приложения с HTTP/2 становятся производительнее, например, благодаря тому, что через одно и то же соединение TCP передается много потоков данных, особенно когда параллельно совершается много запросов, скажем, для статических файлов и AJAX. Это упраздняет некоторые прежние методы оптимизации вроде CSS-спрайтов или объединения нескольких файлов в один. Более того, с HTTP/2 подобные трюки могут принести вред, потому что они требуют передавать избыточные данные.

Чтобы приспособить для HTTP/2 приложения, которые создавались для HTTP/1.1, их может понадобиться доработать, потому что, например, возможность поддерживать активное соединение может вызвать непредсказуемые побочные эффекты. По этим и другим причинам имеет смысл тестировать веб-приложения, прежде чем переделывать их под HTTP/2. Иногда даже бывают случаи, когда разработчики обнаруживают, что после перехода на HTTP/2 их приложения начинают медленнее загружаться или потреблять больше ресурсов.

Из этого следует, что стоит тестировать приложения и наблюдать за тем, как они работают в среде реальной эксплуатации, чтобы понять, в чем выражаются различия между версиями HTTP. Поскольку многие преимущества HTTP/2 проявляются, только когда веб-приложение запущено в полноценном браузере, то результаты простого нагрузочного теста в командной строке могут отличаться от опыта реальных пользователей, которые открывают приложение в браузере. Например, одна из типичных ошибок заключается в том, что разработчики иногда не учитывают возможности конвейерной обработки HTTP/1.1.

Впрочем, большинство реальных сайтов выиграют от перехода на HTTP/2. При этом многие компании продолжают использовать HTTP/1.1 или gRPC в своих внутренних HTTP API для микросервисов.

HTTP/3 и QUIC

Протокол HTTP/3 опирается на достижения HTTP/2 и переносит его принципы на транспортный протокол QUIC[1]. Он основан на UDP и применяется вместо TCP, который использовался со всеми предыдущими версиями HTTP.

Также в отличие от прежних версий HTTP/3 задействует облегченные потоки данных, вместо того чтобы устанавливать новые соединения TCP. Потоки инициализируются не в контексте существующего соединения TCP, а с помощью QUIC, который специально предназначен для того, чтобы их поддерживать.

В распространенных сценариях использования HTTP у QUIC есть немало преимуществ перед TCP. В частности, поскольку в основе протокола лежит UDP[2] (который примитивнее, зато быстрее, чем TCP), QUIC позволяет предотвратить ситуации, когда все соединение «подвисает» в ожидании, пока поступит один-единственный пакет, даже если он предназначен для другого потока. Еще QUIC оптимизирован для того, чтобы как можно быстрее устанавливать начальное соединение, в ходе которого также инициализируется TLS, чтобы защитить канал связи между клиентом и сервером. В QUIC также изначально закладывалась расширяемость и поддержка будущих версий, и вскоре после того, как этот протокол был стандартизирован, многие расширения тоже начали стандартизироваться и внедряться.

Поскольку QUIC основан на UDP и разработан так, чтобы не *окостеневать*, отпадает необходимость во многих традиционных формах промежуточных узлов и шлюзов.

ПРИМЕЧАНИЕ

Окостенение (ossification) *протокола* заключается в том, что промежуточные узлы (или другие компоненты, которые взаимодействуют с протоколом) требуют, чтобы протокол сохранял определенную форму, в результате чего его становится трудно развивать и изменять (например, добавляя расширения).

Давайте посмотрим, как эти элементарные понятия HTTP вписываются в более крупную инфраструктуру, в которой будет функционировать большинство ваших

---

[1] Название протокола QUIC вдохновлено словом *quick* (*быстрый*), хотя официально не считается аббревиатурой. Первоначально предлагалось расшифровывать его как *Quick UDP Internet Connections* (*быстрые интернет-соединения по UDP*), но от этой идеи отказались. — *Примеч. науч. ред.*

[2] Аббр. *User Datagram Protocol* (*протокол пользовательских датаграмм*). — *Примеч. пер.*

веб-приложений. Соответствующие архитектуры обычно не состоят из одного веб-сервера и клиента (такого, как ваш браузер или `curl`); чаще всего в них задействовано несколько уровней коммуникации по протоколу HTTP, из-за чего простые проблемы могут наслаиваться друг на друга так, что с ними становится труднее бороться.

Как обычно, мы познакомим вас с ключевыми понятиями, в которых необходимо ориентироваться, а затем продемонстрируем практические примеры того, как устранять неполадки в более сложной веб-инфраструктуре с помощью `curl` в командной строке.

Балансировка нагрузки

Балансировка нагрузки заключается в том, что нагрузка, которая предназначена для той или иной службы, распределяется между многими ее экземплярами. Хотя этот механизм широко применяется не только для HTTP и веб-служб, HTTP остается одним из наиболее характерных контекстов для современной балансировки нагрузки.

Важно понимать, как балансируется нагрузка в веб-приложениях, потому что от этого зависит, как дефекты и ошибки проявляются в среде эксплуатации. Например, в своей локальной среде разработки вы обычно имеете дело с одним клиентом (вашим браузером или другим потребителем в форме API) и одним сервером (то есть веб-приложением или службой, которую вы разрабатываете). Но в реальном мире между клиентом и приложением часто бывает много промежуточных серверов, которые принимают и ретранслируют ваш трафик HTTP и могут попутно вносить свои собственные дефекты и ошибки.

Задача этого раздела — в том, чтобы вы в общих чертах поняли, из каких факторов и компонентов складывается приложение в продакшен-среде, даже если эти компоненты не относятся к коду, который вы пишете.

Чтобы балансировать нагрузку HTTP, перед приложением обычно помещают прослойку инфраструктуры, которая проксирует запросы HTTP. Как правило, эта прослойка реализуется одним из указанных ниже способов:

1. Служба шлюза, такая как сервер HTTP, который поддерживает соответствующую функциональность (например, nginx или Apache).
2. Выделенная служба (например, HAProxy или relayd).
3. Облачная служба (балансировщик нагрузки GCP, ELB или ALB от AWS и пр.).
4. Аппаратный балансировщик нагрузки.

Иногда специалисты по техническому обслуживанию используют специально разработанную службу или решение на основе DNS, особенно в контексте регио-

нализованной балансировки, которая часто применяется в качестве дополнительной прослойки перед одним из перечисленных механизмов. Системы координации контейнеров и выделенные средства обнаружения служб обычно тоже предоставляют еще один механизм балансировки нагрузки.

Чтобы говорить о балансировке нагрузки, необходимо освоить еще несколько понятий: в основном они относятся к тому, как конкретно балансировщики распределяют запросы от клиентов между серверами, на которых работают экземпляры вашего веб-приложения.

Управлять сеансами и cookie становится сложнее, потому что запросы в рамках одного продолжительного сеанса не обязательно будут направляться к одному и тому же серверу. Если один сервер из пула вашего приложения откажет, это может создать проблемы: например, у пользователей оборвется соединение или пропадут данные. Если вы отлаживаете собственное веб-приложение, сможете ли вы воспроизвести проблему, которая проявляется всего на одном сервере из сотен или тысяч?

Если вы понимаете, как устроена современная балансировка нагрузки, это поможет избежать подобных недостатков проектирования или ужасов отладки. Несколько следующих разделов призваны выработать у вас ментальную модель, которая позволит предотвращать эти проблемы.

Закрепленные сеансы, cookies и детерминированное хеширование

Когда вы настраиваете балансировку нагрузки для служб HTTP, один из первых вопросов, который нужно решить, заключается в том, требует ли служба закрепленных сеансов. Закрепленный (так называемый «липкий») сеанс заключается в том, что на время сеанса клиент привязывается к одному конкретному серверу приложения. Это часто необходимо, если приложение хранит состояние сеанса на само́м сервере.

Это одна из причин, по которым лучше всего разрабатывать приложения без состояния, а точнее такие, которые записывают состояние на выделенный слой данных. Работа этих приложений не нарушится, если один запрос клиента будет обработан одним сервером, а другой — другим. К счастью, в современных реалиях, особенно если используется облачная инфраструктура, закрепленные сеансы обычно не нужны. Тем не менее о них не стоит забывать, особенно когда вы устраняете неполадки, которые необъяснимым образом проявляются только в среде эксплуатации с балансировкой нагрузки.

Хотя HTTP позволяет закреплять сеансы по-разному, чаще всего это делается с помощью cookies. Это могут быть cookies приложения (например, сеансовые cookies), о которых осведомлен балансировщик нагрузки, или специальные cookies, которыми балансировщик управляет сам.

Впрочем, если закреплять сеансы так, что балансировщику нагрузки приходится хранить дополнительное состояние, это вызывает новые проблемы. Если балансировщик должен помнить, какие IP-адреса соответствуют тем или иным серверам приложения, то что произойдет, если это состояние потеряется оттого, что балансировщик откажет или его заменят? Как видите, в этом случае мы просто переложили проблему состояния с сервера приложения на балансировщик нагрузки и понадеялись, что ничего плохого от этого не произойдет. Но как гласит пословица, одной надеждой сыт не будешь.

Один из хитрых способов закрепить сеанс и не столкнуться с затруднениями из-за того, что состояние приходится хранить на балансировщике нагрузки, состоит в том, чтобы балансировать нагрузку с помощью хеширования IP-адресов. При этом вычисляется хеш-сумма IP-адреса клиента, которая используется, чтобы отображать IP-адреса запросов на экземпляры службы. Пока IP-адрес клиента не меняется, сеанс будет «закреплен» за конкретным экземпляром приложения.

В результате один или несколько балансировщиков нагрузки будут детерминированным образом сопоставлять IP-адреса с серверами приложения, и им не потребуется передавать или сохранять состояние. Серверы можно менять сколь угодно часто, и каждый новый сервер будет работать с тем же отображением, что и все остальные, потому что они все используют один и тот же алгоритм хеширования, а значит, будут сопоставлять каждый IP-адрес с одним и тем же сервером приложения.

Круговая балансировка нагрузки

Если закреплять сеансы не нужно, то нагрузка обычно балансируется по круговому принципу: каждое новое соединение или запрос направляется на следующий экземпляр. На языке математики это означает, что выбирается экземпляр, номер которого равен остатку от деления количества запросов на количество экземпляров.

Другие механизмы

Итак, вы получили представление о том, как балансировка нагрузки HTTP работает на практике. Конечно, существует много других механизмов: например, можно распределять нагрузку в зависимости от потребления ресурсов. Однако стоит хорошо представлять себе, чем оборачивается дополнительная сложность: «интеллектуальные» алгоритмы балансировки нагрузки, как правило, не обходятся без подводных камней.

Например, балансировщики на основе потребления ресурсов далеко не всегда правильно обрабатывают короткие всплески нагрузки, в результате чего одни экземпляры оказываются недогруженными, а другие — перегруженными. Это часто происходит, если реальная нагрузка характеризуется всплесками и измеря-

ется в неподходящие моменты времени, что не дает возможности выровнять эти всплески.

Если добавить еще один уровень сложности, чтобы их выравнивать, это может привести к другим неприятностям: например, множественные всплески начнут накладываться друг на друга. Если у вас возникает искушение отклониться от проторенного маршрута более стандартных алгоритмов, убедитесь, что ваша команда тщательно продумала архитектуру и контекст приложения в реальных условиях.

Высокая доступность

Хотя нагрузку балансируют в первую очередь ради того, чтобы обеспечить быстрый отклик, балансировщик обычно следит за тем, какие службы доступны. Он может применять средства проверки работоспособности, чтобы убедиться, что серверы, к которым он посылает запросы, функционируют в штатном режиме. Это означает, что балансировка нагрузки также помогает добиться высокой доступности и часто служит неотъемлемой частью архитектур с нулевым временем простоя, когда службу можно заменить (например, когда развертывается новая версия) так, чтобы клиенты не заметили никаких перебоев.

Для этого можно предусмотреть механизм элегантного завершения работы экземпляров по такой схеме: клиентские соединения не обрываются, а остаются активными до тех пор, пока не будут полностью обслужены, зато новые соединения маршрутизируются только на обновленные экземпляры. Когда завершается последний сеанс, необновленный экземпляр можно полностью отключить.

Проверки работоспособности позволяют балансировщику нагрузки выяснять, исправна ли та или иная служба. Безусловно, в первую очередь проверяется, можно ли установить соединение. Однако в микросервисных архитектурах порой бывает так, что служба не может правильно отвечать на запросы, если недоступна внешняя зависимость (например, другая служба). Это тоже можно отслеживать с помощью выделенной конечной точки состояния.

Во многих приложениях и командах специалистов по инфраструктуре принят специальный путь вроде `/healthcheck`[1], код состояния которого показывает, можно ли отправлять запросы этой службе. В некоторых более сложных средах этот код может даже сообщать о том, *какого рода* запросы готов принимать экземпляр.

Когда компетентные разработчики приложений работают вместе с платформенной командой и специалистами по надежности сайтов, пути для проверки работоспособности порой удается наладить так, чтобы они сигнализировали о ситуациях, в которых требуются действия со стороны инфраструктуры: например, если эк-

[1] *Проверка работоспособности. — Примеч. пер.*

земпляр неисправен и его нужно заменить. Если эти пути хорошо продуманы, они обычно передают дополнительный контекст и сведения о проблеме, которые облегчают отладку в среде эксплуатации.

По мере того как инфраструктура, которая поддерживает веб-приложение, растет и усложняется, количество потенциальных узких мест возрастает экспоненциально, и конкретные риски зависят от архитектуры и приложения. Один из классов проблем, вероятность которых возрастает по мере того, как в веб-инфраструктуру добавляются новые уровни проксирования и маршрутизации, — это циклическая переадресация и другие ошибки переадресации.

К счастью, эти неполадки прекрасно помогает выявить такой инструмент командной строки, как `curl`.

Как устранять неполадки переадресации с помощью curl

Как мы только что упомянули, переадресация может быть характерным симптомом дефектов, проблем и прочего неожиданного поведения веб-приложения и инфраструктуры, которая его окружает. Чтобы следовать за переадресацией, запускайте curl с флагом `-L` (или `--location`[1]):

```
curl -IL http://www.tutorialinux.com/
HTTP/1.1 301 Moved Permanently
Server: nginx/1.24.0
Date: Sun, 22 Oct 2023 22:58:02 GMT
Content-Type: text/html
Content-Length: 169
Connection: keep-alive
Location: https://tutorialinux.com/

HTTP/2 200
server: nginx/1.24.0
date: Sun, 22 Oct 2023 22:58:02 GMT
content-type: text/html; charset=UTF-8
vary: Accept-Encoding
x-powered-by: PHP/8.2.11
link: <https://tutorialinux.com/wp-json/>; rel="https://api.w.org/"
strict-transport-security: max-age=31536000; includeSubdomains; preload
```

Как видите, на исходный запрос сервер отвечает кодом 301 Moved Permanently[2] и сообщает новый адрес: `https://tutorialinux.com/`. Команда `curl` следует по переадресованному маршруту и отправляет запрос на новый адрес, который отвечает кодом 200 OK.

---

[1] *Местоположение. — Примеч. пер.*

[2] *Перемещено в новое постоянное расположение. — Примеч. пер.*

Эта переадресация работает так, как нужно, но аналогичная команда `curl` позволяет также выявлять циклическую переадресацию в приложении и устранять неполадки кэширования и маршрутизации в балансировщиках нагрузки с многоуровневой структурой.

Однако иногда, чтобы устранить неполадки в приложении, вам требуется пойти еще дальше и отправить ему данные. И здесь `curl` тоже будет полезен!

Как тестировать API с помощью curl

Если вы умеете тестировать API с помощью `curl`, этот навык может пригодиться чаще, чем вам кажется. Такое тестирование особенно актуально для API, которые работают в формате JSON и принимают данные POST: в этом случае можно отправить тестовые данные на конечный узел, чтобы убедиться, что сервер возвращает ожидаемый результат:

```
curl --header "Content-Type: application/json" \
     --request POST \
     --data '{"имя_1":"значение_1","имя_2":"значение_2"}' \
     http://localhost:4000/api/v1/endpoint
```

Эта команда запускается с несколькими флагами, которые стоит запомнить:

Короткая форма	Длинная форма	Буквальный перевод	Описание
`-H`	`--header`	Заголовок	Строка заголовка запроса. Чтобы передать несколько заголовков, можно просто повторить этот аргумент
`-X`	`--request`	Запрос	Тип запроса HTTP (по умолчанию используется GET)
`-d`	`--data`	Данные	Передаваемые данные (в случае запросов типа POST или PATCH)

Если вы передаете данные с помощью параметра `--data`, не забывайте о том, что в командной строке по-прежнему действуют экранирующие символы Bash, так что сложные данные может оказаться удобнее передать в виде файла:

```
curl -X POST -d "@путь/к/файлу" https://localhost/api/v1/endpoint
```

Не забудьте поставить символ `@` в начале пути к файлу. Если вы передаете сложные данные, прочтите документацию по параметрам `--data-raw`, `--data-binary` и `--data-urlencode`. В зависимости от того, каких данных ожидает ваше веб-приложение, вам также может понадобиться отправить те или иные дополнительные заголовки.

Теперь вы знаете, как интерактивно взаимодействовать с веб-приложениями, которые вы отлаживаете, отправляя им произвольные данные с помощью `curl`. Осталось рассмотреть последний практический прием, который относится к TLS (протокол, с помощью которого шифруется трафик в современном HTTPS). Нельзя сказать, что с TLS связаны какие-то особые заблуждения в контексте веб-приложений, но здесь есть специфический камень преткновения, с которым тоже помогает справиться `curl`.

Как принимать и отображать недействительные сертификаты TLS с помощью curl

У команды `curl` есть параметр `-k` (`--insecure`[1]), который позволяет принимать с сервера недействительные сертификаты TLS и продолжать запрос. Это может пригодиться, когда нужно устранять неполадки на неправильно сконфигурированном сервере:

```
curl --insecure -v https://www.tutorialinux.org/
```

В результате `curl` будет работать так, будто сертификат TLS действителен, даже если это не так. Очевидно, это создает риски безопасности, так что параметр `--insecure` стоит применять только для отладки, чтобы `curl` продолжала работу и в том случае, когда сертификат не удается проверить и в штатном режиме запрос был бы прекращен.

Давайте вкратце рассмотрим последнюю тему из области HTTP, о которой стоит иметь представление, если вы пишете или отлаживаете веб-приложения: речь идет о CORS.

CORS

Браузерная технология CORS[2] предназначена для того, чтобы обрабатывать ресурсы, которые поступают с другого домена, например изображения, видеозаписи, код на HTML и JavaScript или даже ответы AJAX. Прежде чем загружать ресурсы из стороннего источника, браузер запрашивает у него разрешение на эту операцию. Это называется предварительным запросом.

Предварительный запрос — это запрос типа `OPTIONS`; он рассчитан на ответ с заголовками HTTP, которые сообщают, разрешен ли такой запрос. Чаще всего этот ответ поступает с кодом состояния 204 No Content и содержит только заголовки. Если таких заголовков нет или если они не подтверждают, что запрос разрешен, браузер не будет отправлять дальнейшие запросы к этому ресурсу.

---

[1] *Небезопасный. — Примеч. пер.*

[2] Аббр. *Cross-Origin Resource Sharing* (*совместное использование ресурсов между разными источниками*). *— Примеч. пер.*

Вот как может происходить такая коммуникация. Браузер открывает сайт `https://www.example.org` и спрашивает, можно ли отправить запрос типа POST к ресурсу `/api/test` на этом сайте:

```
OPTIONS /api/test
Origin: https://www.example.org
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-Custom-Header, Content-Type
```

Разрешающий ответ будет выглядеть так:

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://www.example.org
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-Custom-Header, Content-Type
Access-Control-Allow-Max-Age: 3600
```

Поскольку это означает, что желаемый запрос разрешен, браузер теперь может отправить сам этот запрос:

```
POST /api/test
Origin: https://www.example.org
Content-Type: application/json
X-Custom-Header: foobarbaz
```

А если запрос не разрешен, то в ответе не будет никакого специального кода или указания на то, что запрос отклонен; в нем просто не будет разрешающего заголовка `Access-Control-Allow-Origin`. В этом случае клиент понимает, что запрос не авторизован, и регистрирует ошибку.

Ошибки такого рода вы могли наблюдать в технической консоли своего браузера, где они выглядят примерно так:

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the
remote resource at https://www.example.org. (Reason: something)[1].
```

Мы представили вам краткое введение в CORS, потому что веб-разработчикам важно понимать эту тему. Хотя здесь мы не работали с командной строкой, разработчикам бывает полезно ориентироваться в CORS и анализировать журналы ошибок своих веб-клиентов на предмет соответствующих сообщений. Чтобы глубже разобраться в этой теме, мы рекомендуем обратиться к статье на сайте MDN: developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

---

[1] *Запрос между источниками заблокирован: принцип одинакового источника не позволяет читать удаленный ресурс по адресу https://www.example.org. (Причина: такая-то). — Примеч. пер.*

Итоги

В этой главе мы рассмотрели темы, в которых нужно разбираться, чтобы избежать распространенных заблуждений, ошибок и досадных дефектов проектирования, проявляющихся, когда веб-приложение покидает компьютер разработчика и начинает взаимодействовать с реальным миром через сложную инфраструктуру. Вы узнали о некоторых ее компонентах — шлюзах и восходящих узлах, которые служат посредниками при доступе к вашему приложению.

Также вы получили представление о самых распространенных ошибках разработчиков в области HTTP и теперь благодаря этому сможете избежать трудноотлаживаемых проблем с заголовками, неверными или неинформативными кодами состояния и не только. Наконец, вы узнали о том, что такое CORS и как протокол HTTP эволюционировал до своего нынешнего состояния.

Но, возможно, еще важнее, что ваша квалификация разработчика повысилась благодаря тому, что вы овладели таким инструментом командной строки, как `curl`, и научились применять его в сочетании с теоретическими знаниями HTTP.

Материал этой главы позволит вам быстро и эффективно устранять неполадки с веб-приложениями: например, выявлять циклическую переадресацию на неисправном сайте WordPress, диагностировать неочевидную проблему кэширования, анализируя заголовки, которые вернуло приложение на Ruby-on-Rails, или в четыре часа утра отправлять запросы типа POST с определенными данными в формате JSON, чтобы воспроизвести проблему в продакшене (и проверить ее решение).

Эту книгу мы выпустили вместе с Orion soft

Orion soft создает ПО для ИТ-инфраструктуры Enterprise-бизнеса. Главные принципы разработки, которым отвечает каждый продукт компании, — стабильность, простота использования и безопасность.

В продуктовую экосистему Orion soft входит zVirt — платформа виртуализации серверов, дисков и сетей с одной из крупнейших баз инсталляций в России.

Orion soft стремится постоянно развивать как свои компетенции, так и российское ИТ-сообщество, собирая и распространяя лучшие современные практики разработки.

Linux для разработчиков

‹packt›

Разработчики всегда стремятся подняться на новый уровень мастерства, но большинство полностью теряется, когда дело доходит до командной строки Linux.

С помощью этой книги вы сделаете следующий важный шаг в своей карьере. Большую часть навыков, которые вы получите после ее прочтения, можно сразу же применить на практике, чтобы стать более эффективным разработчиком. Книга написана специально для программистов, а не для системных администраторов Linux. Каждая глава даст достаточно теоретических знаний, чтобы понять, что вы делаете, прежде чем переходить к практическим командам, которые вы сможете использовать в своей повседневной работе в качестве разработчика ПО.

По мере прочтения вы быстро освоите основы работы Linux и освоитесь с командной строкой. Овладев основными навыками, вы разберетесь, как применять их в различных контекстах, с которыми столкнетесь как разработчик ПО: создание образов Docker и работа с ними, автоматизация скучных задач сборки с помощью сценариев оболочки и устранение неполадок в продакшен-средах.

К концу книги вы сможете с комфортом пользоваться Linux и командной строкой и применять приобретенные навыки в повседневной работе. Это позволит вам экономить время, быстро устранять неполадки и стать мастером работы с командной строкой, к которому обращается вся команда.

- Изучите полезные приемы и инструменты работы с командной строкой, которые упрощают разработку ПО, тестирование и устранение неполадок.
- Поймете, как на самом деле работают Linux и среды командной строки.
- Создадите мощные настраиваемые инструменты и сэкономите тысячи строк кода с помощью утилит Linux, ориентированных на разработчиков.
- Получите практический опыт работы с Docker, SSH и сценариями командной оболочки, что поможет вам стать более эффективным разработчиком.
- Почувствуете себя уверенно при поиске в журналах и устранении неполадок на серверах Linux.
- Освоите нюансы работы с командной строкой, которые ставят в тупик других разработчиков.

SPRINT book

ISBN:978-601-08-4837-5