Иан Милл
Эйдан Хобсон Сейерс

Docker на практике

Docker in Practice

SECOND EDITION

IAN MIELL
AIDAN HOBSON SAYERS

MANNING
Shelter Island

Docker на практике

ИАН МИЛЛ
ЭЙДАН ХОБСОН СЕЙЕРС

Перевод с английского Беликов Д.А.

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2020

УДК 004.451Docker
ББК 32.972.1
М57

М57 Иан Милл, Эйдан Хобсон Сейерс
Docker на практике / пер. с англ. Д.А. Беликов. – М.: ДМК Пресс, 2020. – 516 с.: ил.

ISBN 978-5-97060-772-5

Данная книга научит вас надежным, проверенным методам, используемым Docker, таким как замена виртуальных машин, использование архитектуры микросервисов, эффективное моделирование сети, производительность в автономном режиме и создание процесса непрерывной доставки на базе контейнеров. Следуя формату «проблема/решение» в стиле поваренной книги, вы изучите реальные варианты использования Docker и узнаете, как применить их к собственным проектам.

Издание предназначено разработчикам, использующим Docker в своем рабочем окружении.

УДК 004.451Docker
ББК 32.972.1

Original English language edition published by Manning Publications. Copyright © 2019 by Manning Publications. Russian language edition copyright © 2020 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-61729-480-8 (англ.)
ISBN 978-5-97060-772-5 (рус.)

Copyright © 2019 by Manning Publications Co.
© Оформление, перевод на русский язык, издание, ДМК Пресс, 2020

Содержание

Предисловие

В сентябре 2013 года, просматривая сайт *Hacker News*, я наткнулся на статью в *Wired* о новой технологии под названием «Docker». Когда я ее читал, я все больше волновался, осознавая революционный потенциал этой платформы.

Компания, на которую я работал более десяти лет, изо всех сил пыталась поставлять программное обеспечение достаточно быстро. Подготовка среды была дорогостоящим, трудоемким, ручным и неэлегантным занятием. Непрерывной интеграции почти не существовало, а настройка среды разработки требовала терпения. Поскольку название моей должности включало в себя слова «менеджер DevOps», я был очень заинтересован в решении этих проблем!

Я привлек пару мотивированных коллег (один из них теперь мой соавтор) через список рассылки компании, и наша команда разработчиков трудилась над тем, чтобы превратить бета-инструмент в бизнес-преимущество, снижая высокую стоимость виртуальных машин и предлагая новые способы мышления относительно сборки и развертывания программного обеспечения. Мы даже собрали и открыли исходный код инструмента автоматизации (ShutIt) для удовлетворения потребностей нашей организации в доставке.

Docker дал нам упакованный и обслуживаемый инструмент, решавший многие проблемы, которые были бы фактически непреодолимыми, если бы мы взяли их решение на себя. Это был открытый исходный код в лучшем виде, который позволил нам принять вызов, используя свое свободное время, преодолевая технический долг и ежедневно обучаясь. Обучаясь не только Docker, но и непрерывной интеграции, доставке, упаковке, автоматизации и тому, как люди реагируют на быстрые и разрушительные технологические изменения.

Для нас Docker – удивительно обширный инструмент. Везде, где вы запускаете программное обеспечение с использованием Linux, Docker может влиять на него. Это затрудняет написание книги на эту тему, так как ландшафт настолько же широк, как и само программное обеспечение. Задача усложняется необычайной скоростью, с которой экосистема Docker производит решения для удовлетворения потребностей, возникающих в результате таких фундаментальных изменений в производстве программного обеспечения. Со временем форма проблем и решений стала нам знакомой, и в книге мы постарались передать данный опыт. Это позволит вам найти решения для ваших конкретных технических и бизнес-ограничений.

Выступая на встречах, мы поражены тем, насколько быстро Docker стал эффективным в организациях, готовых использовать его. Эта книга отражает то, как мы использовали Docker, переходя от настольных компьютеров через конвейер DevOps вплоть до производства. Следовательно, книга признана не всеми, но, будучи инженерами, мы считаем, что безупречность иногда уступает место практичности, особенно когда речь идет об экономии денег! Все в этой книге основано на реальных уроках в этой области, и мы надеемся, что вы извлечете пользу из нашего с трудом завоеванного опыта.

Иан Милл

Благодарности

Эта книга не могла быть написана без поддержки, жертв и терпения самых близких нам людей. Особо следует отметить Стивена Хэзлтона (Stephen Hazleton), чьи неустанные усилия, направленные на то, чтобы сделать Docker полезным для наших клиентов, наполнили содержимым большую часть книги.

Некоторые соавторы и сотрудники Docker были достаточно любезны, чтобы прочитать книгу на разных этапах, и оставили много полезных отзывов, включая следующих людей, которые ознакомились с рукописью: Бенуа Бенедетти, Буркхард Нестманн, Чэд Дэвис, Дэвид Моравек, Эрнесто Карденас Кангауала, Фернандо Родригес, Кирк Братткус, Петуру Радж, Скотт Бейтс, Стивен Лембарк, Стюарт Вудворд, Тисен Беннетт, Валмики Аркиссандас и Уил Мур III (Benoit Benedetti, Burkhard Nestmann, Chad Davis, David Moravec, Ernesto Cárdenas Cangahuala, Fernando Rodrigues, Kirk Brattkus, Pethuru Raj, Scott Bates, Steven Lembark, Stuart Woodward, Ticean Bennett, Valmiky Arquissandas Wil Moore III). Хосе Сан Леандро (José San Leandro) выступил в качестве нашего технического корректора, и мы благодарны ему за его острый взгляд.

Наконец, это издание также многим обязано редакционной команде Manning, которая старалась изо всех сил подтолкнуть нас к тому, чтобы сделать книгу не только достаточно хорошей, но и лучшей, какой она могла быть. Мы надеемся, что гордость за свою работу отразилась на нас.

Иан Милл (IAN MIELL). Спасибо Саре, Исааку и Рэйчел за то, что они мирились с программированием по ночам, отцом, приклеенным к экрану ноутбука, и вечным «Docker то, Docker это, Docker бла-бла», и моим родителям за поддержку с раннего возраста ставить под сомнение статус-кво. И за покупку мне этого Спектрума.

Эйдан Хобсон Сейерс (AIDAN HOBSON SAYERS). Спасибо Моне за поддержку и стимул, родителям за их мудрость и мотивирующие слова, а также моему соавтору за это роковое сообщение по электронной почте: «Кто-нибудь пробовал этот Docker?».

Об этой книге

Docker, пожалуй, самый быстрорастущий программный проект в мире. Его код был открыт в марте 2013 года, к 2018 году он получил почти 50 000 звезд на GitHub и свыше 14 000 ответвлений. Он принял значительное количество запросов на принятие изменений от таких компаний, как Red Hat, IBM, Microsoft, Google, Cisco и VMWare.

Docker достиг этой критической массы, откликнувшись на насущную потребность многих программных организаций: способность создавать программное обеспечение открытым и гибким способом, а затем надежно и последовательно развертывать его в различных контекстах. Вам не нужно изучать новый язык программирования, покупать дорогостоящее аппаратное обеспечение или делать многое в плане установки или настройки, чтобы компактно собирать, поставлять и запускать приложения с помощью Docker.

Второе издание *Docker in Practice* знакомит вас с реальными примерами использования Docker с применением техник, которые мы брали в различных контекстах. Там, где это возможно, мы пытались объяснить эти методы, не требуя знания других технологий, прежде чем приступать к чтению. Мы предполагаем, что читатели знакомы с основными методами и концепциями, такими как способность разрабатывать структурированный код и знание процессов разработки и развертывания программного обеспечения. Кроме того, предполагаются знания основных идей управления исходным кодом и базовые знания основ сети, таких как TCP/IP, HTTP и порты. Менее важные направления мы будем объяснять по мере продвижения.

Начиная с краткого изложения основ Docker в первой части, во второй части мы заостряем внимание на использовании Docker при разработке на одном компьютере. В третьей части мы переходим к использованию Docker с конвейером DevOps, рассказывая о непрерывной интеграции, непрерывной доставке и тестировании.

В четвертой части речь идет о том, как запускать контейнеры Docker масштабируемым образом с помощью оркестровки.

В последней части описывается, как запустить Docker в производстве, особое внимание уделяется вариантам стандартных производственных операций, а также тому, что может пойти не так и как с этим бороться.

Docker – это настолько широкий, гибкий и динамичный инструмент, что не отставать от его стремительного развития – задача не для слабонервных. Мы постарались дать вам понимание критических концепций с помощью реальных приложений и примеров с целью предоставить возможность критически оценить будущие инструменты и технологии в экосистеме Docker с уверенностью. Мы надеялись сделать книгу приятной экскурсией по множеству увиденных нами способов, которыми Docker делает жизнь проще и даже веселее.

Погружение в Docker познакомило нас со многими интересными методиками программного обеспечения, охватывающими весь его жизненный цикл, и мы надеемся, что вы разделите этот опыт.

Дорожная карта

Эта книга состоит из 16 глав, разделенных на 5 частей.

Часть 1 закладывает основы для остальной книги, знакомя вас с Docker и предлагая выполнять некоторые основные команды Docker. Глава 2 посвящена знакомству с архитектурой Docker «клиент-сервер» и способам ее отладки, что может быть полезно для выявления проблем с нетрадиционными настройками Docker.

Часть 2 посвящена знакомству с Docker и максимально эффективному его использованию на вашем собственном компьютере. Аналогия с концепцией, которая, возможно, вам известна, – виртуальные машины – используется в качестве основы для главы 3, чтобы показать более простой способ по-настоящему приступить к использованию Docker. В главах 4, 5 и 6 подробно описывается несколько техник, которые мы ежедневно применяем для создания образов, их запуска и управления самим Docker. В последней главе этой части более подробно рассматривается тема создания образов с помощью методов управления конфигурацией.

В третьей части мы рассмотрим использование Docker в контексте DevOps – от его использования для автоматизации сборок и тестирования программного обеспечения до перемещения вашего собранного программного обеспечения в различные места. Эта часть завершается главой о виртуальной сети Docker, которая представляет Docker Compose и охватывает ряд более сложных тем, связанных с сетевой средой, таких как сетевое моделирование и сетевые плагины Docker.

Часть 4 исследует тему оркестровки контейнеров. Мы отправимся в путешествие от одного контейнера на одном хосте к платформе на основе Docker, работающей в «центре обработки данных в качестве операционной системы». Глава 13 – это расширенное обсуждение областей, которые необходимо учитывать при выборе платформы на основе Docker, и она также служит руководством к тому, что думают архитекторы предприятия при внедрении таких технологий.

Часть 5 охватывает ряд тем для эффективного использования Docker в рабочей среде. В главе 14 рассматривается важная тема безопасности, объясняется, как заблокировать процессы, происходящие внутри контейнера, и как ограничить доступ к внешнему демону Docker. В последних двух главах подробно рассматривается ключевая практическая информация для запуска Docker в производстве. Глава 15 демонстрирует, как применять классические знания системного администратора в контексте контейнеров, от логирования до ограничений ресурсов, а глава 16 рассматривает проблемы, с которыми вы можете столкнуться, и предоставляет шаги для отладки и решения.

В приложениях содержатся подробные сведения об установке, использовании и настройке Docker различными способами, в том числе на виртуальной машине и в Windows.

О коде

Исходный код для всех инструментов, приложений и образов Docker, которые мы создали для использования в этой книге, доступен на GitHub в организации «docker-in-practice»: https://github.com/docker-in-practice. Образы на Docker Hub под пользователем «dockerin-practice» (https://hub.docker.com/u/dockerinpractice/) обычно представляют собой автоматические сборки из одного из репозиториев GitHub. Там, где мы чувствовали, что читателю может быть интересно дальнейшее изучение исходного кода, лежащего в основе методики, в обсуждение методики была включена ссылка на соответствующий репозиторий. Исходный код также доступен на сайте издателя по адресу www.manning.com/books/docker-in-practice-second-edition.

Значительное количество листингов кода в книге иллюстрирует терминальную сессию, которой должен следовать читатель, вместе с соответствующим выводом команд. Следует отметить пару вещей относительно этих сессий:

- длинные терминальные команды могут использовать символ продолжения строки оболочки (\), чтобы разделить команду на несколько строк. Хотя это будет работать в вашей оболочке, если вы напечатаете его, вы также можете опустить его и набрать команду в одной строке;
- если раздел вывода не предоставляет дополнительную полезную информацию для обсуждения, он может быть опущен, а вместо него вставлено многоточие ([...]).

Книжный форум

Приобретение *Docker in Practice* (2-е изд.) включает в себя бесплатный доступ к частному веб-форуму, организованному Manning Publications, где вы можете оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, перейдите на страницу www.manning.com/books/docker-in-practice-second-edition. Вы также можете узнать больше о форумах Manning и правилах поведения по адресу https://forums.manning.com/forums/about.

Обязательство Manning перед своими читателями состоит в том, чтобы обеспечить место, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Это не обязательство какого-либо конкретного участия со стороны автора, чей вклад в форум остается добровольным (и не оплачивается). Мы предлагаем вам задать автору несколько сложных вопросов, чтобы его интерес не пропал.

Об иллюстрации на обложке

Изображение на обложке *Docker in Practice* (2-е изд.) называется «Человек из Сельце, Хорватия». Иллюстрация взята из репродукции альбома хорватских традиционных костюмов середины XIX века Николы Арсеновича, изданного Этнографическим музеем в Сплите, Хорватия, в 2003 году. Иллюстрации были получены от услужливого библиотекаря из Этнографического музея, который расположен в римском ядре средневекового центра города: руинах дворца императора Диоклетиана примерно 304 г. н. э. Книга включает в себя превосходные цветные иллюстрации людей из разных регионов Хорватии, сопровождаемые описаниями костюмов и повседневной жизни.

За последние 200 лет изменились дресс-код и образ жизни, а разнообразие регионов, столь богатое в то время, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных деревушках или городах, разделенных всего несколькими милями. Возможно, мы обменяли культурное разнообразие на более пеструю личную жизнь – конечно, в пользу более яркой и быстро развивающейся технологической жизни.

Manning приветствует изобретательность и инициативу компьютерного бизнеса с обложками книг, основанными на богатом разнообразии региональной жизни двухсотлетней давности, возвращенными к жизни иллюстрациями из старых книг и коллекций, подобных этой.

Часть 1

Основы Docker

Первая часть этой книги состоит из глав 1 и 2, которые знакомят вас с использованием Docker и его основами.

Глава 1 объясняет происхождение Docker, а также его основные понятия, такие как образы, контейнеры и слои. Наконец, вы займетесь практикой, создав первый образ с помощью файла Dockerfile.

Глава 2 знакомит с некоторыми полезными приемами, которые помогут вам глубже понять архитектуру Docker. Беря каждый основной компонент по очереди, мы рассмотрим взаимоотношения между демоном Docker и его клиентом, реестром Docker и Docker Hub.

К концу первой части вы освоите базовые концепции Docker и сможете продемонстрировать некоторые полезные приемы, заложив прочную основу для оставшейся части книги.

Глава 1

Знакомство с Docker

О чем рассказывается в этой главе:

- что такое Docker;
- использование Docker и как он может сэкономить вам время и деньги;
- различия между контейнерами и образами;
- слои Docker;
- сборка и запуск приложения с использованием Docker.

Docker – это платформа, которая позволяет «создавать, поставлять и запускать любое приложение повсюду». За невероятно короткое время она прошла большой путь и теперь считается стандартным способом решения одного из самых дорогостоящих аспектов программного обеспечения – развертывания.

До появления Docker в конвейере разработки обычно использовались комбинации различных технологий для управления движением программного обеспечения, такие как виртуальные машины, инструменты управления конфигурацией, системы управления пакетами и комплексные сети библиотечных зависимостей. Все эти инструменты должны были управляться и поддерживаться специализированными инженерами, и у большинства из них были свои собственные уникальные способы настройки.

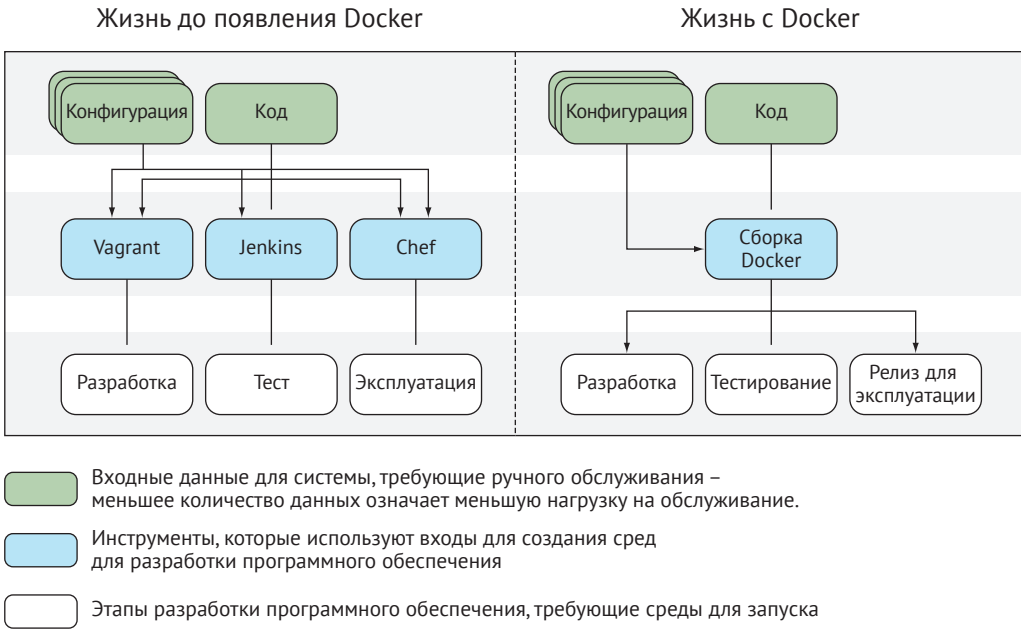


Рис. 1.1 ❖ Как Docker облегчил бремя обслуживания инструментов

Docker изменил все это, позволив различным инженерам, вовлеченным в этот процесс, эффективно говорить на одном языке, облегчая совместную работу. Все проходит через общий конвейер к одному выходу, который можно использовать для любой цели, – нет необходимости продолжать поддерживать запутанный массив конфигураций инструментов, как показано на рис. 1.1.

В то же время нет необходимости выбрасывать существующий программный стек: если он работает – можно упаковать его в контейнер Docker как есть, чтобы другие могли его использовать. В качестве бонуса вы увидите, как были созданы эти контейнеры, поэтому, если нужно покопаться в деталях, вы можете это сделать.

Данная книга предназначена для разработчиков среднего уровня, обладающих рядом знаний о Docker. Если вы знакомы с основами, не стесняйтесь переходить к последующим главам. Цель этой книги – раскрыть реальные проблемы, с которыми сталкивается Docker, и показать, как их можно преодолеть. Но сначала мы немного расскажем о самом Docker. Если вы хотите более подробно изучить его основы, обратите внимание на книгу Джеффа Николоффа *Docker в действии* (Manning, 2016).

В главе 2 вы познакомитесь с архитектурой Docker более подробно, с помощью ряда методов, демонстрирующих его мощь. В этой главе вы узнаете, что такое Docker, поймете, почему он важен, и начнете его использовать.

1.1. Что такое Docker, и для чего он нужен

Прежде чем приступить к практике, мы немного обсудим Docker, чтобы вы поняли его контекст, то, откуда пришло название «Docker», и почему мы вообще его используем!

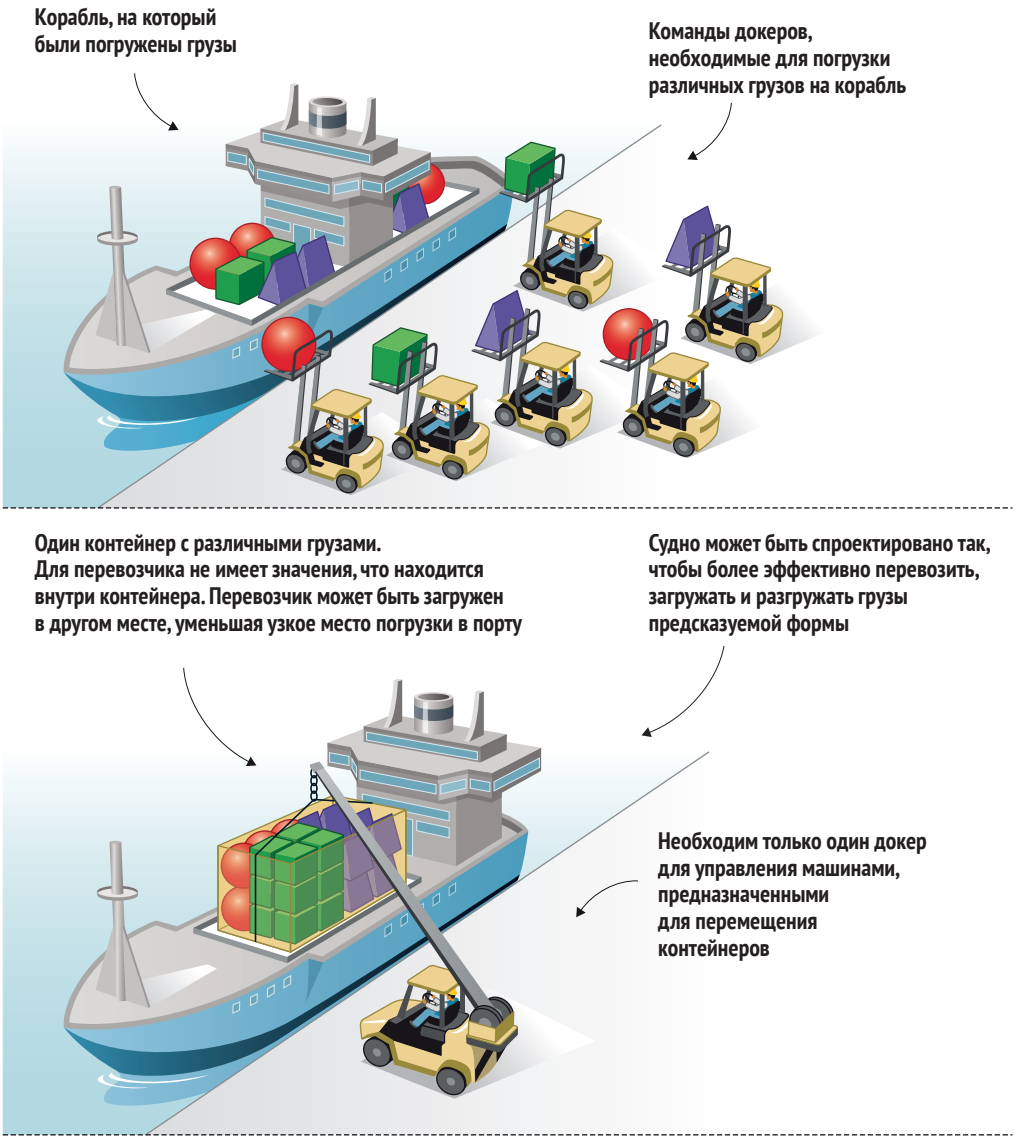


Рис. 1.2 ❖ Доставка до и после стандартизованных контейнеров

1.1.1. Что такое Docker?

Чтобы понять, что такое Docker, проще начать с метафоры, чем с технического объяснения, а метафора у Docker – мощная. *Докером (docker)* назывался рабочий, который переносил товары на суда и обратно, когда те стояли в портах. Там были ящики и грузы разных размеров и форм, и опытные докеры ценились за то, что они в состоянии вручную поместить товары на корабли экономически эффективным способом (см. рис. 1.2). Нанять людей для перемещения этих грузов было недешево, но альтернативы не было.

Это должно быть знакомо всем, кто работает в сфере программного обеспечения. Много времени и интеллектуальной энергии тратится на перенос программного обеспечения нечетной формы в метафорические корабли разного размера, заполненные другим программным обеспечением нечетной формы, чтобы их можно было продавать пользователям или предприятиям где бы то ни было.

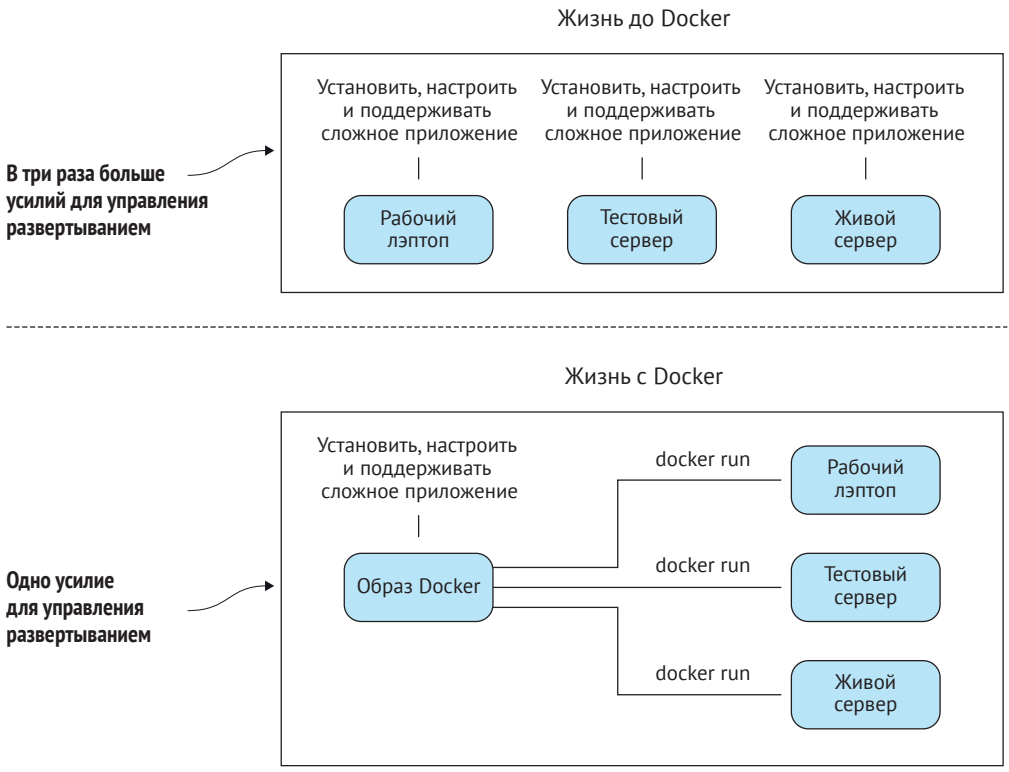


Рис. 1.3 ❖ Доставка программного обеспечения до и после Docker

На рис. 1.3 показано, как можно сэкономить время и деньги с помощью концепции Docker. До появления Docker развертывание программного обеспечения в различных средах требовало значительных усилий. Даже если вы не выполняли сценарии вручную для подготовки программного обеспечения

на разных компьютерах (а многие так и делают), вам все равно приходилось бороться с инструментами управления конфигурацией, которые регулируют состояние в быстро растущих средах, испытывающих нехватку ресурсов. Даже когда эти усилия были заключены в виртуальных машинах, много времени тратилось на управление развертыванием этих машин, ожидание их загрузки и управление накладными расходами на использование ресурсов, которые они создавали.

В Docker процесс конфигурирования отделен от управления ресурсами, а процесс развертывания тривиален: запустите `docker run`, и образ среды будет извлечен и готов к запуску, потребляя меньше ресурсов и не мешая другим средам.

Вам не нужно беспокоиться о том, будет ли ваш контейнер отправлен на компьютер с Red Hat, Ubuntu или образ виртуальной машины CentOS; до тех пор пока там есть Docker, с ним все будет хорошо.

1.1.2. Чем хорош Docker?

Возникают некоторые важные практические вопросы: зачем вы используете Docker и для чего? Короткий ответ на вопрос «почему» заключается в том, что при минимальных усилиях Docker может быстро сэкономить вашему бизнесу много денег. Некоторые из этих способов (и далеко не все) обсуждаются в следующих подразделах. Мы видели все эти преимущества не понаслышке в реальных условиях работы.

Замена виртуальных машин

Docker может использоваться для замены виртуальных машин во многих ситуациях. Если вас волнует только приложение, а не операционная система, Docker может заменить виртуальную машину, а вы можете оставить заботу об операционной системе кому-то другому. Docker не только быстрее, чем виртуальная машина, предназначенная для запуска, он более легок в перемещении, и благодаря его многоуровневой файловой системе вы можете легче и быстрее делиться изменениями с другими. Он также прочно укоренен в командной строке и в высшей степени пригоден для написания сценариев.

Прототипирование программного обеспечения

Если вы хотите быстро поэкспериментировать с программным обеспечением, не нарушая существующую настройку и не проходя через трудности, связанные с подготовкой виртуальной машины, Docker может предоставить «песочницу» за миллисекунды. Освобождающий эффект этого процесса трудно понять, пока вы не испытаете его на себе.

Упаковка программного обеспечения

Поскольку образ Docker фактически не имеет зависимостей для пользователя Linux, это отличный способ для упаковки программного обеспечения. Вы можете создать свой образ и быть уверенным, что он может работать на любом современном компьютере с Linux, – подумайте о Java без необходимости создания виртуальной машины Java.

Возможность для архитектуры микросервисов

Docker упрощает декомпозицию сложной системы на серию составных частей, что позволяет более детально рассуждать о своих сервисах. Это может позволить вам реструктурировать свое программное обеспечение, чтобы сделать его части более управляемыми и подключаемыми, не затрагивая целое.

Моделирование сетей

Поскольку вы можете запустить сотни (даже тысячи) изолированных контейнеров на одном компьютере, смоделировать сеть очень просто. Это может отлично подойти для тестирования реальных сценариев по разумной цене.

Возможность производительности полного стека в автономном режиме

Поскольку можно связать все части вашей системы в контейнеры Docker, вы можете реализовать их оркестровку для запуска на своем ноутбуке и работать в пути даже в автономном режиме.

Сокращение неизбежных расходов на отладку

Сложные переговоры между различными командами относительно поставляемого программного обеспечения – обычное явление в данной отрасли. Мы лично пережили бесчисленные дискуссии по поводу испорченных библиотек, проблемных зависимостей, неправильного применения обновлений или их применения в неверном порядке (или даже их отсутствия), невоспроизводимых ошибок и т. д. Вероятно, вы тоже. Docker позволяет вам четко указать (даже в форме сценария) шаги для отладки проблемы в системе с известными свойствами, что значительно упрощает воспроизведение ошибок и среды и обычно позволяет отделить ее от среды хоста.

Документирование зависимостей программного обеспечения и точки взаимодействия

Создавая образы в структурированном виде, готовые к перемещению в различные среды, Docker заставляет вас явно документировать свои программные зависимости с базовой отправной точки. Даже если вы решите не использовать Docker повсюду, эта документация может помочь вам установить программное обеспечение в других местах.

Возможность непрерывной доставки

Непрерывная доставка – это парадигма доставки программного обеспечения, основанная на конвейере, который перестраивает систему при каждом изменении, а затем осуществляет доставку в производство (или «вживую») посредством автоматизированного (или частично автоматизированного) процесса.

Поскольку вы можете более точно контролировать состояние среды сборки, сборки Docker более воспроизводимы, чем традиционные методы сборки программного обеспечения. Это делает реализацию непрерывной доставки намного проще. Стандартные технологии непрерывной доставки, такие как развертывание Blue/Green (где «живое» и «последнее» развертывания

поддерживаются в реальном времени) и развертывание Phoenix (где целые системы создаются заново в каждом релизе), становятся тривиальными благодаря реализации воспроизводимого Docker-ориентированного процесса сборки.

Теперь вы немного знаете, как Docker может вам помочь. Прежде чем мы углубимся в реальный пример, давайте рассмотрим несколько основных концепций.

1.1.3. Ключевые концепции

В этом разделе мы рассмотрим некоторые ключевые концепции Docker, которые проиллюстрированы на рис. 1.4.

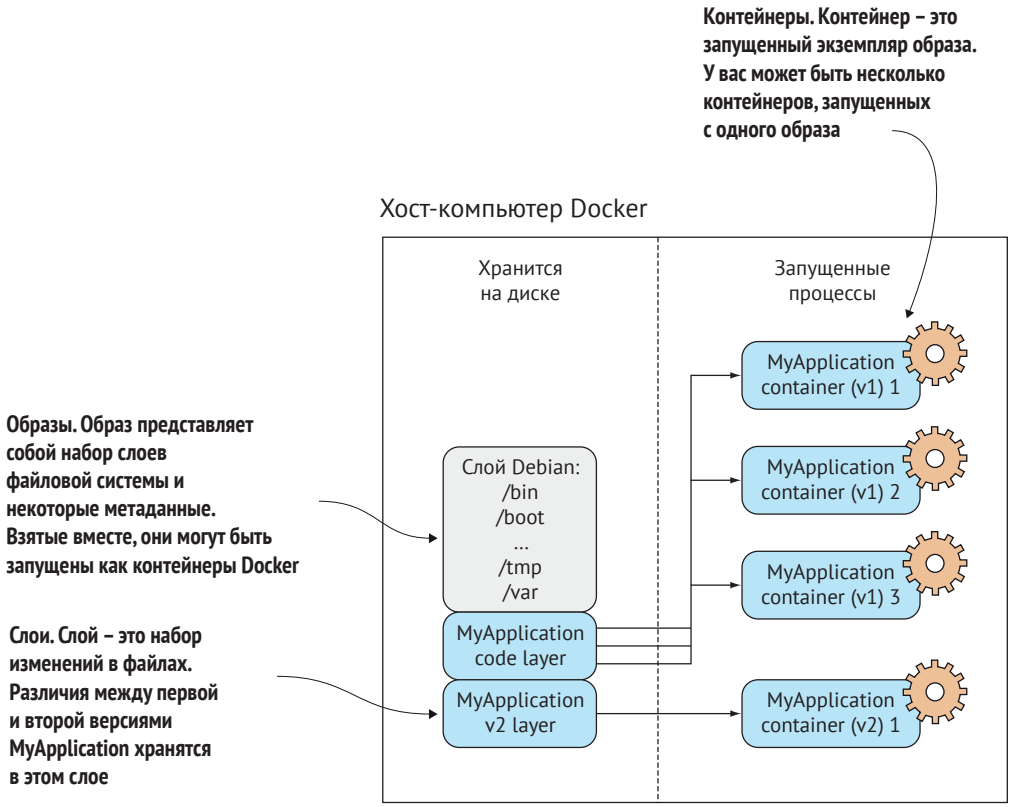


Рис. 1.4 ❖ Базовые концепции Docker

Прежде чем запускать команды Docker, лучше всего разобраться с понятиями образов, контейнеров и слоев. Говоря кратко, *контейнеры* запускают системы, определенные *образами*. Эти *образы* состоят из одного или нескольких *слоев* (или наборов различий) плюс некоторые метаданные Docker.

Давайте посмотрим на некоторые основные команды Docker. Мы превратим образы в контейнеры, изменим их и добавим слои к новым образам, которые сохраним. Не волнуйтесь, если все это звучит странно. К концу главы все станет намного понятнее.

Ключевые команды Docker

Основная функция Docker – создавать, отправлять и запускать программное обеспечение в любом месте, где есть Docker. Для конечного пользователя Docker – это программа с командной строкой, которую они запускают. Как и git (или любой другой инструмент управления исходным кодом), у этой программы есть подкоманды, которые выполняют различные операции. Основные подкоманды Docker, которые вы будете использовать на своем хосте, перечислены в табл. 1.1.

Таблица 1.1 ❖ Подкоманды Docker

Команда	Назначение
`docker build`	Собрать образ Docker
`docker run`	Запустить образ Docker в качестве контейнера
`docker commit`	Сохранить контейнер Docker в качестве образа
`docker tag`	Присвоить тег образу Docker

Образы и контейнеры

Если вы не знакомы с Docker, это может быть первый раз, когда вы встречаете слова «контейнер» и «образ» в данном контексте. Это, вероятно, самые важные концепции в Docker, поэтому стоит потратить немного времени, чтобы убедиться, что разница ясна. На рис. 1.5 вы увидите иллюстрацию этих концепций с использованием трех контейнеров, запущенных из одного базового образа.

Один из способов взглянуть на образы и контейнеры – это рассматривать их как программы и процессы. Точно так же процесс может рассматриваться как «выполняемое приложение», контейнер Docker может рассматриваться как образ, выполняемый Docker.

Если вы знакомы с принципами объектно-ориентированного программирования, еще один способ взглянуть на образы и контейнеры – это рассматривать образы как классы, а контейнеры – как объекты. Точно так же, как объекты представляют собой конкретные экземпляры классов, контейнеры являются экземплярами образов. Вы можете создать несколько контейнеров из одного образа, и все они будут изолированы друг от друга так же, как и объекты. Что бы вы ни изменили в объекте, это не повлияет на определение класса, – это принципиально разные вещи.

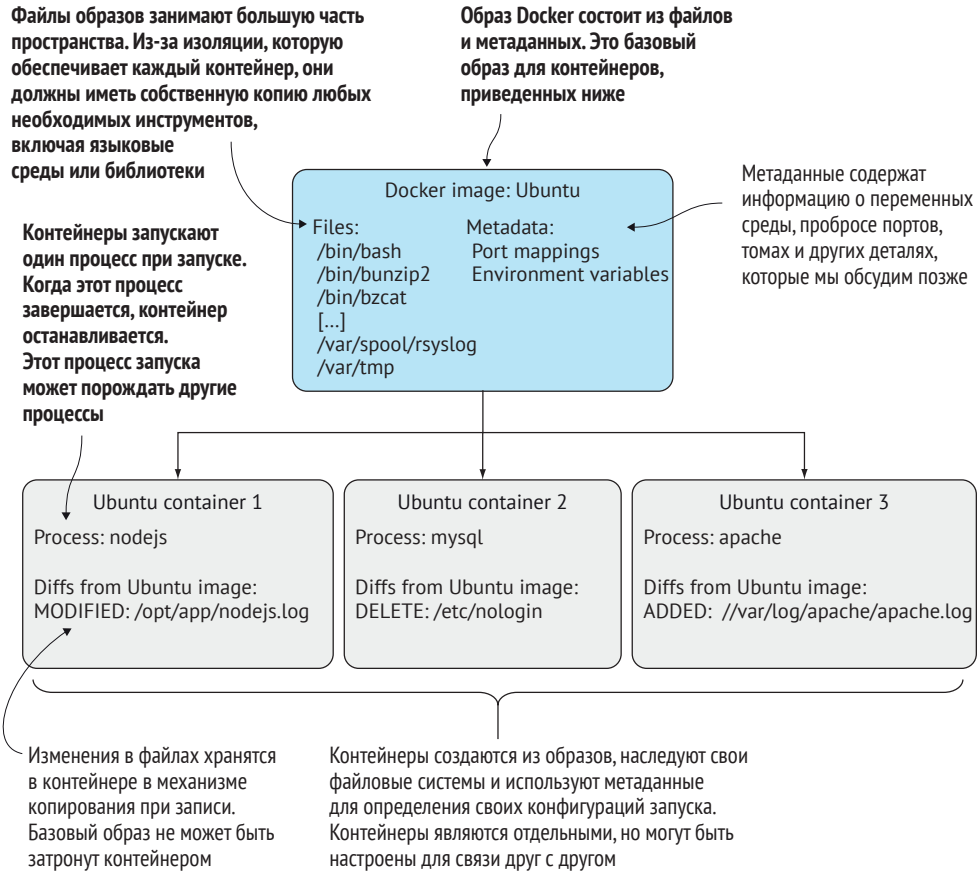


Рис. 1.5 ❖ Образы и контейнеры Docker

1.2. Создание приложения Docker

Теперь мы перейдем к практике, создав простой образ приложения в формате to-do с помощью Docker. В ходе этого процесса вы встретитесь с некоторыми ключевыми функциями Docker, такими как файлы Dockerfiles, повторное использование образов, отображение портов и автоматизация сборки. Вот что вы узнаете в течение следующих 10 минут:

- как создать образ Docker с помощью Dockerfile;
- как присвоить тег образу Docker для удобства пользования;
- как запустить свой новый образ Docker.

Приложение в формате to-do – это приложение, которое помогает вам отслеживать то, что вы хотите сделать. Приложение, создаваемое нами, будет хранить и отображать короткие строки информации, которые можно пометить как выполненные, представленные в простом веб-интерфейсе. На рис. 1.6 показано, чего мы добьемся благодаря этому.

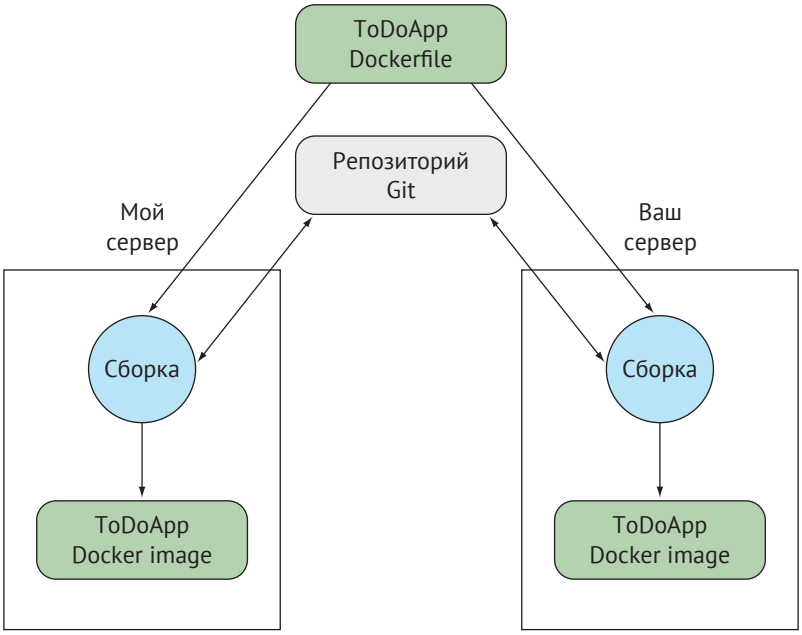


Рис. 1.6 ❖ Создание приложения Docker

Детали приложения не важны. Мы продемонстрируем, что из одного короткого файла Dockerfile, который мы собираемся вам дать, вы можете гарантированно создавать, выполнять, останавливать и запускать приложение как на своем, так и на нашем хосте, не беспокоясь об установке или зависимостях. Это ключевая часть того, что предлагает Docker, – надежно воспроизводимые, легко управляемые и совместно используемые среды разработки. Это означает, что не нужно следовать более сложным или неоднозначным инструкциям по установке, в которых можно потеряться.

> **ПРИМЕЧАНИЕ.** Это приложение будет использоваться неоднократно на протяжении всей книги, и оно довольно полезно для экспериментов и демонстраций, так что стоит с ним ознакомиться.

1.2.1. Способы создания нового образа Docker

Существует четыре стандартных способа создания образов Docker. Они перечислены в табл. 1.2

Таблица 1.2 ❖ Методы создания образов Docker

Метод	Описание	Источник
Команды Docker/ «От руки»	Запустите контейнер с помощью `docker run` и введите команды для создания образа в командной строке. Создайте новый образ с помощью `docker commit`	См. метод 15
Dockerfile	Выполните сборку из известного базового образа и укажите ее с помощью ограниченного набора простых команд	Будет обсуждаться в скором времени
Dockerfile и инструмент управления конфигурацией	То же самое, что и Dockerfile, но вы передаете контроль над сборкой более сложному инструменту управления конфигурацией	См. метод 55
Стереть образ и импортировать набор файлов	Из пустого образа импортируйте файл TAR с необходимыми файлами	См. метод 11

Первый вариант «от руки» подойдет, если вы проверяете концепцию, чтобы увидеть, работает ли ваш процесс установки. В то же время вы должны вести записи о предпринимаемых вами шагах, чтобы при необходимости иметь возможность вернуться к нужной вам точке.

В какой-то момент вы захотите определить шаги для создания своего образа.

Это опция Dockerfile (и та опция, которую мы будем использовать здесь).

Для более сложных сборок вы можете выбрать третий вариант, особенно когда функции Dockerfile недостаточно сложны для вашего образа.

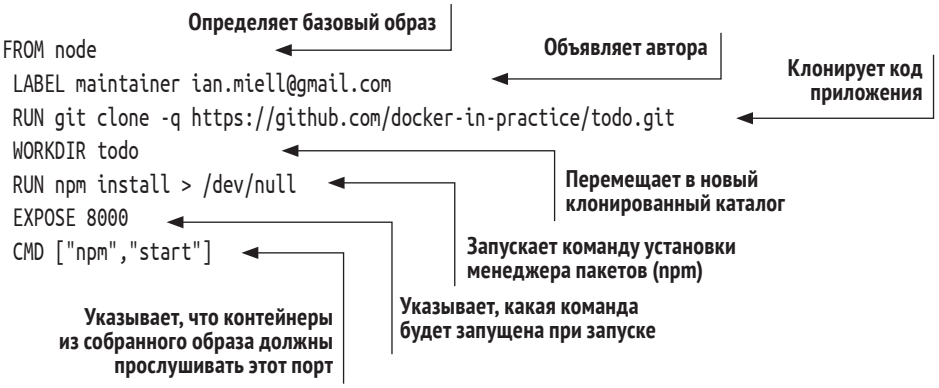
Последний вариант строится из нулевого образа путем наложения набора файлов, необходимых для запуска образа. Это полезно, если вы хотите импортировать набор автономных файлов, созданных в другом месте, но этот метод редко встречается при массовом использовании.

Сейчас мы рассмотрим метод Dockerfile; другие будут изучены позже.

1.2.2. Пишем Dockerfile

Dockerfile – это текстовый файл, содержащий серию команд. В листинге 1.1 приведен файл Dockerfile, который мы будем использовать в этом примере. Заведите новую папку, перейдите в нее и создайте файл с именем «Dockerfile» с таким содержимым:

Листинг 1.1. Файл Dockerfile



Файл Dockerfile начинается с определения базового образа с помощью команды FROM. В этом примере используется образ Node.js, поэтому у вас есть доступ к двоичным файлам Node.js. Официальный образ Node.js называется node.

Далее вы объявляете автора с помощью команды LABEL. В этом случае мы используем один из наших адресов электронной почты, но вы можете заменить его своей ссылкой, поскольку теперь это ваш файл Dockerfile. Эта строка не обязательна для создания рабочего образа Docker, но рекомендуется ее включить. На этом этапе сборка унаследовала состояние node-контейнера, и вы готовы работать над ним.

Затем вы клонируете код приложения с помощью команды RUN. Указанная команда используется для получения кода приложения, выполняя git внутри контейнера. В этом случае Git устанавливается внутри базового образа, но нельзя принимать это как должное.

Теперь вы перемещаетесь в новый клонированный каталог с помощью команды WORKDIR. Это не только меняет каталоги в контексте сборки, но и последняя команда WORKDIR определяет, в каком каталоге вы находитесь по умолчанию, когда запускаете свой контейнер из собранного образа.

Затем вы запускаете команду установки менеджера пакетов Node.js (npm). Она установит зависимости для вашего приложения. В этом примере вывод вас не интересует, поэтому вы перенаправляете его в /dev/null.

Поскольку порт 8000 используется приложением, вы применяете команду EXPOSE, чтобы сообщить Docker, что контейнеры из собранного образа должны прослушивать этот порт.

Наконец, вы используете команду CMD, чтобы сообщить Docker, какая команда будет выполнена при запуске контейнера.

Этот простой пример иллюстрирует несколько ключевых особенностей Docker и файлов Dockerfiles. Dockerfile – это простая последовательность ограниченного набора команд, выполняемых в строгом порядке. Это оказывает воздействие на файлы и метаданные полученного образа. Здесь команда RUN

влияет на файловую систему, проверяя и устанавливая приложения, а команды EXPOSE, CMD и WORKDIR – на метаданные образа.

1.2.3. Собираем образ Docker

Вы определили шаги сборки своего файла Dockerfile. Теперь вы собираетесь создать из него образ Docker, набрав команду, показанную на рис. 1.7. Вывод будет выглядеть примерно так:

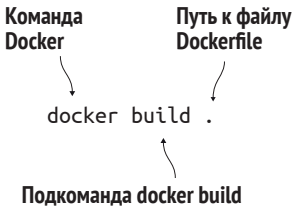


Рис. 1.7 ❖ Команда сборки

```
Sending build context to Docker daemon  2.048kB
Step 1/7 : FROM node
 ---> 2ca756a6578b
Step 2/7 : LABEL maintainer ian.miell@gmail.com
 ---> Running in bf73f87c88d6
 ---> 5383857304fc
Removing intermediate container bf73f87c88d6
Step 3/7 : RUN git clone -q https://github.com/docker-in-practice/todo.git
 ---> Running in 761baf524cc1
 ---> 4350cb1c977c
Removing intermediate container 761baf524cc1
Step 4/7 : WORKDIR todo
 ---> a1b24710f458
Removing intermediate container 0f8cd22fbe83
Step 5/7 : RUN npm install > /dev/null
 ---> Running in 92a8f9ba530a
npm info it worked if it ends with ok
[...]
npm info ok
 ---> 6ee4d7bba544
Removing intermediate container 92a8f9ba530a
Step 6/7 : EXPOSE 8000
 ---> Running in 8e33c1ded161
 ---> 3ea44544f13c
```

Каждая команда приводит к созданию нового образа с выводом его идентификатора

Docker загружает файлы и каталоги по пути, предоставленному команде docker build

Каждый шаг сборки последовательно нумеруется, начиная с 1, и выводится командой

Для экономии места каждый промежуточный контейнер удаляется, перед тем как продолжить

Отладка сборки выводится здесь (и редактируется из этого списка)

```
Removing intermediate container 8e33c1ded161
Step 7/7 : CMD npm start
 ---> Running in ccc076ee38fe
 ---> 66c76cea05bb
Removing intermediate container ccc076ee38fe
Successfully built 66c76cea05bb
```

Окончательный идентификатор образа для этой сборки, готовый к присвоению тега

Теперь у вас есть образ Docker со своим идентификатором («66c76cea05bb» в предыдущем примере, но ваш идентификатор будет другим). Возможно, к нему неудобно обращаться, поэтому вы можете присвоить ему тег для удобства, как показано на рис. 1.8.

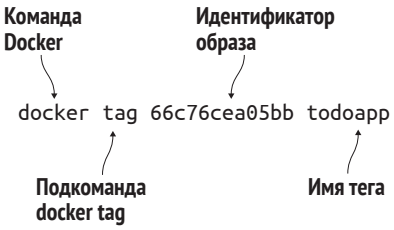


Рис. 1.8 ❖ Подкоманда docker tag

Введите предыдущую команду, заменив 66c76cea05bb сгенерированным для вас идентификатором образа.

Теперь вы можете собрать свою собственную копию образа Docker из файла Dockerfile, воспроизводя среду, определенную кем-то другим!

1.2.4. Запускаем контейнер Docker

Вы собрали свой образ Docker и присвоили ему тег. Теперь вы можете запустить его в качестве контейнера:

Листинг 1.2. Вывод docker run для todoapp

```
$ docker run -i -t -p 8000:8000 --name example1 todoapp
npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm info prestart todomvc-swarm@0.0.1

> todomvc-swarm@0.0.1 prestart /todo
 > make all

npm install
```

Подкоманда docker run запускает контейнер, -p перенаправляет порт контейнера 8000 в порт 8000 на хост-компьютере, --name присваивает контейнеру уникальное имя, а последний аргумент – это образ

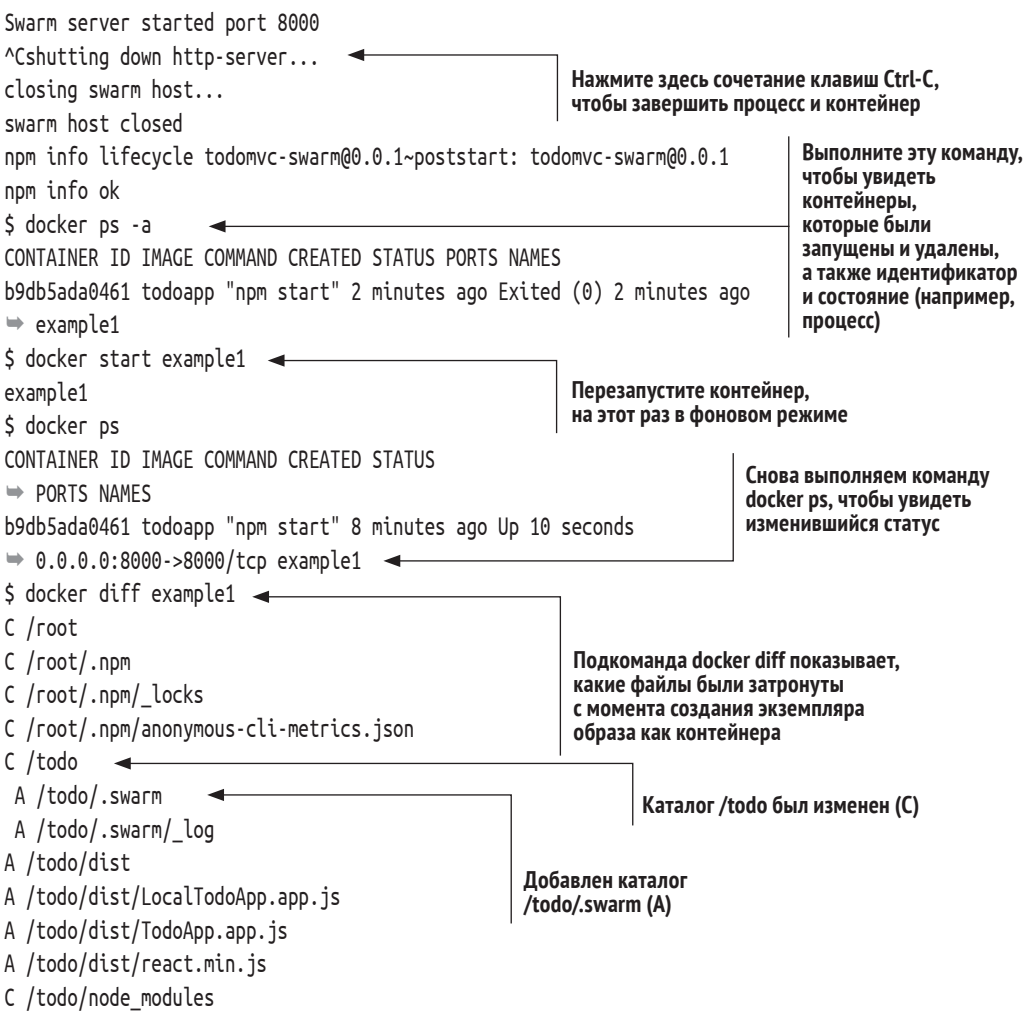
Вывод процесса запуска контейнера отправляется на терминал

```
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm WARN package.json todomvc-swarm@0.0.1 No repository field.
npm WARN package.json todomvc-swarm@0.0.1 license should be a valid SPDX
➥ license expression
npm info preinstall todomvc-swarm@0.0.1
npm info package.json statics@0.1.0 license should be a valid SPDX license
➥ expression
npm info package.json react-tools@0.11.2 No license field.
npm info package.json react@0.11.2 No license field.
npm info package.json node-
     jsx@0.11.0 license should be a valid SPDX license expression
npm info package.json ws@0.4.32 No license field.
npm info build /todo
npm info linkStuff todomvc-swarm@0.0.1
npm info install todomvc-swarm@0.0.1
npm info postinstall todomvc-swarm@0.0.1
npm info prepublish todomvc-swarm@0.0.1
npm info ok
if [ ! -e dist/ ]; then mkdir dist; fi
cp node_modules/react/dist/react.min.js dist/react.min.js

LocalTodoApp.js:9: // TODO: default english version
LocalTodoApp.js:84: fwdList = this.host.get('/TodoList#'+listId);
 // TODO fn+id sig
TodoApp.js:117: // TODO scroll into view
TodoApp.js:176: if (i>=list.length()) { i=list.length()-1; } // TODO
➥ .length
local.html:30: <!-- TODO 2-split, 3-split -->
model/TodoList.js:29: // TODO one op - repeated spec? long spec?
view/Footer.jsx:61: // TODO: show the entry's metadata
view/Footer.jsx:80: todoList.addObject(new TodoItem()); // TODO
➥ create default
view/Header.jsx:25: // TODO list some meaningful header (apart from the
➥ id)

npm info start todomvc-swarm@0.0.1

> todomvc-swarm@0.0.1 start /todo
> node TodoAppServer.js
```



Подкоманда `docker run` запускает контейнер. Флаг `-p` перенаправляет порт контейнера 8000 в порт 8000 на хост-компьютере, поэтому теперь вы можете перейти в своем браузере по адресу http://localhost:8000 для просмотра приложения.

Флаг `-name` присваивает контейнеру уникальное имя, к которому вы можете обратиться позже для удобства. Последний аргумент – это имя образа.

Как только контейнер будет запущен, нажмите сочетание клавиш **Ctrl-C**, чтобы завершить процесс и контейнер. Вы можете выполнить команду `ps`, чтобы увидеть контейнеры, которые были запущены, но не были удалены. Обратите внимание, что у каждого контейнера есть свой идентификатор и статус, аналогичный процессу. Его статус `Exited`, но вы можете перезапустить его. После того как вы это сделаете, обратите внимание, что статус изменился на `Up`, и теперь виден порт, перенаправляемый из контейнера в хост-компьютер.

Подкоманда `docker diff` показывает, какие файлы были затронуты с момента создания экземпляра образа как контейнера. В этом случае каталог todo был изменен (C), и были добавлены другие перечисленные файлы (A). Файлы не были удалены (D) – это другая возможность.

Как видно, тот факт, что Docker «содержит» вашу среду, означает, что вы способны рассматривать ее как сущность, над которой можно предсказуемо выполнять действия. Это дает Docker широкие возможности – влиять на жизненный цикл программного обеспечения от разработки до эксплуатации и обслуживания. Эти изменения – то, о чем пойдет речь в этой книге, показывая вам на практике, что можно сделать с помощью Docker.

Далее вы узнаете о слоях, еще одной ключевой концепции Docker.

1.2.5. Слои Docker

Слои Docker помогают справиться с большой проблемой, которая возникает, когда вы используете контейнеры в широком масштабе. Представьте себе, что произойдет, если вы запустите сотни или даже тысячи приложений, и каждому из них потребуется копия файлов для хранения в каком-либо месте.

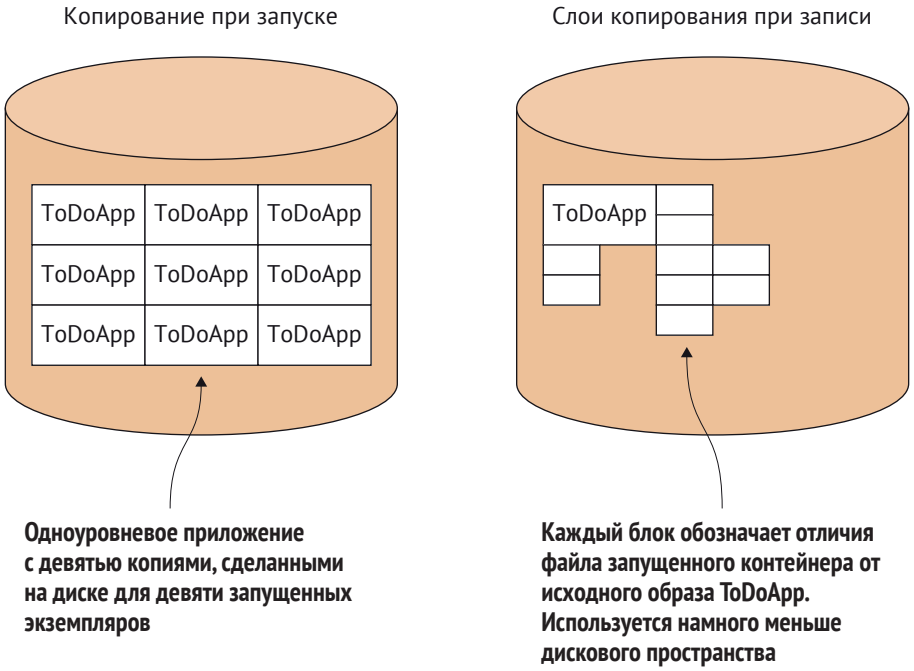


Рис. 1.9 ❖ Слои файловой системы Docker

Как вы можете себе представить, дисковое пространство закончится довольно быстро! По умолчанию Docker внутренне использует механизм копирования при записи, чтобы уменьшить объем требуемого дискового пространства (см. рис. 1.9). Всякий раз, когда работающему контейнеру необходимо

выполнить запись в файл, он записывает изменение, копируя элемент в новую область диска. После выполнения фиксации новая область диска замораживается и записывается как слой со своим собственным идентификатором.

Это отчасти объясняет, как контейнеры Docker могут так быстро запускаться, – им нечего копировать, потому что все данные уже сохранены в виде образа.

> **ПОДСКАЗКА.** Копирование при записи – это стандартная стратегия оптимизации, используемая в вычислительной технике. Когда вы создаете новый объект (любого типа) из шаблона, а не копируете весь требуемый набор данных, вы копируете данные только после их изменения. В зависимости от варианта использования это может сэкономить значительные ресурсы.

На рис. 1.10 показано, что созданное вами приложение содержит три слоя, которые вам интересны. Слои статичны, поэтому, если вам нужно что-то изменить в более высоком слое, можно просто выполнить сборку поверх образа, который вы хотите взять в качестве ссылки. В своем приложении вы создали общедоступный node-образ и многоуровневые изменения сверху.

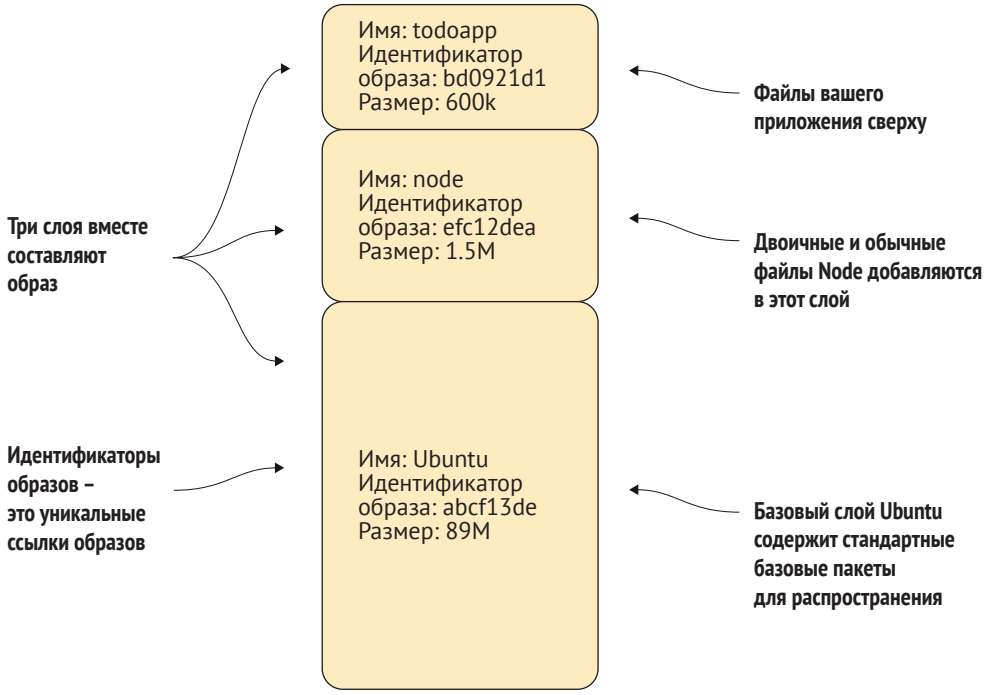


Рис. 1.10 ❖ Концепция слоев в файловой системе todoapp в Docker

Все три слоя могут совместно использоваться несколькими запущенными контейнерами, так же как общая библиотека может совместно использоваться

в памяти несколькими запущенными процессами. Это жизненно важная функция для операций, позволяющая запускать многочисленные контейнеры на основе разных образов на хост-компьютерах, не испытывая нехватки дискового пространства.

Представьте, что вы запускаете свое приложение как сервис для платных клиентов в режиме реального времени. Вы можете расширить свое предложение для большого количества пользователей. Если вы занимаетесь разработкой, то можно запустить много разных сред на локальном компьютере одновременно. Если вы проходите тестирование, можно одновременно выполнять гораздо больше тестов и значительно быстрее, чем раньше. Все это стало возможным благодаря слоям.

Создав и запустив приложение с помощью Docker, вы начали осознавать всю мощь, которую Docker может внести в рабочий процесс. Воспроизведение и совместное использование определенных сред и возможность их размещения в разных местах дают вам гибкость и контроль над разработкой.

Резюме

- Docker – это попытка сделать для программного обеспечения то, что контейнеризация сделала для судоходной отрасли: снизить стоимость локальных различий за счет стандартизации.
- Некоторые из применений Docker включают в себя программное обеспечение для создания прототипов, программное обеспечение для упаковки, снижение затрат на тестирование и отладку сред, а также использование методологий DevOps, таких как непрерывная доставка.
- Вы можете создавать и запускать приложение Docker из файла Dockerfile, используя команды `docker build` и `docker run`.
- Образ Docker – это шаблон для работающего контейнера. Это похоже на разницу между исполняемым файлом программы и запущенным процессом.
- Изменения в запущенных контейнерах можно сохранять и тегировать как новые образы.
- Образы создаются из многоуровневой файловой системы, что уменьшает пространство, используемое образами Docker на вашем хосте.

Глава 2

Постигаем Docker: внутри машинного отделения

О чем рассказывается в этой главе:

- архитектура Docker;
- разбор внутреннего устройства Docker на вашем хосте;
- использование Docker Hub для поиска и загрузки образов;
- настройка собственного реестра Docker;
- как заставить контейнеры общаться друг с другом.

Понимание архитектуры Docker является ключом к более полному его осмыслению. В этой главе дается обзор основных компонентов Docker на вашем компьютере и в сети, а также описываются приемы, которые помогут развить это понимание.

В процессе вы узнаете несколько хитростей, которые помогут вам более эффективно использовать Docker (и Linux). Многие из более поздних и продвинутых методов в этой книге будут основаны на том, что вы видите здесь, поэтому обратите особое внимание на изложенное далее.

2.1. Архитектура Docker

На рис. 2.1 показана архитектура Docker, и она станет центральным элементом этой главы. Мы начнем с обзора высокого уровня, а затем сосредоточимся на каждой части, используя методы, разработанные, для того чтобы укрепить ваше понимание.

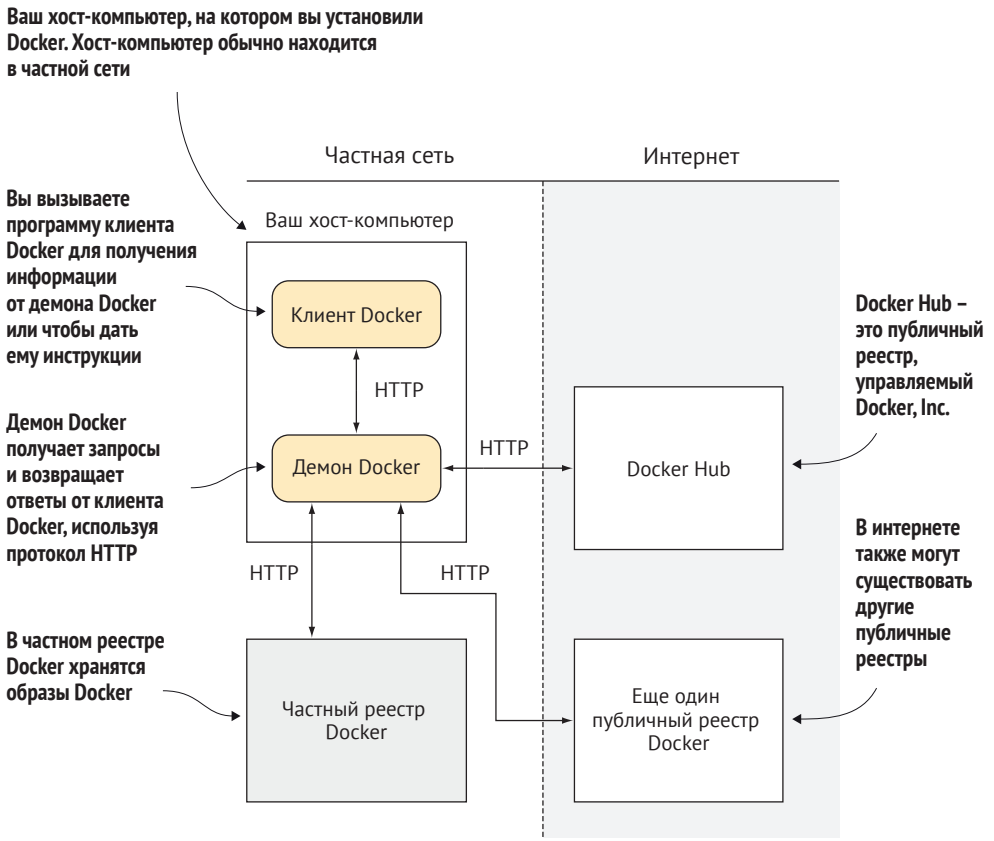


Рис. 2.1 ❖ Обзор архитектуры Docker

Docker на вашем хост-компьютере (на момент написания этих строк) разделен на две части: демон с прикладным программным интерфейсом RESTful и клиент, который общается с демоном. На рис. 2.1 показан ваш хост-компьютер, на котором запущены клиент и демон Docker.

> **ПОДСКАЗКА.** Интерфейс RESTful использует стандартные типы HTTP-запросов, такие как `GET`, `POST`, `DELETE` и др., для представления ресурсов и операций над ними. В этом случае образы, контейнеры, тома и т. п. являются представленными ресурсами.

Вы вызываете Docker-клиент, чтобы получить информацию или дать инструкции демону. Демон – это сервер, который получает запросы и возвращает ответы от клиента по протоколу HTTP. В свою очередь, он будет отправлять запросы в другие службы для отправки и получения образов, также используя протокол HTTP. Сервер будет принимать запросы от клиента командной строки или любого, кто авторизован для подключения.

Демон также отвечает за заботу о ваших образах и контейнерах за кулисами, тогда как клиент выступает в качестве посредника между вами и интерфейсом RESTful.

Частный реестр Docker – это сервис, который хранит образы Docker. Их можно запросить у любого демона Docker, у которого есть соответствующий доступ. Этот реестр находится во внутренней сети и не является общедоступным, поэтому считается закрытым.

Ваш хост-компьютер обычно будет находиться в частной сети. Демон Docker станет обращаться к интернету, чтобы получить образы по запросу.

Docker Hub – это общедоступный реестр, управляемый Docker Inc. В интернете также могут существовать другие публичные реестры, и ваш демон Docker способен взаимодействовать с ними.

В первой главе мы говорили, что контейнеры Docker могут быть отправлены в любое место, где можно запустить Docker, – это не совсем верно. Фактически контейнеры будут работать на компьютере, только если может быть установлен *демон*.

Ключевой момент на рис. 2.1 заключается в том, что, когда вы запускаете Docker на своем компьютере, вы можете взаимодействовать с другими процессами на компьютере или даже со службами, работающими в вашей сети или в интернете.

Теперь, когда у вас есть представление о том, как устроен Docker, мы представим различные методы, относящиеся к разным частям фигуры.

2.2. Демон Docker

Демон Docker (см. рис. 2.2) – это центр ваших взаимодействий с Docker, и поэтому он является лучшим местом, где вы можете начать понимать все соответствующие элементы. Он контролирует доступ к Docker на вашем компьютере, управляет состоянием контейнеров и образов, а также взаимодействует с внешним миром.

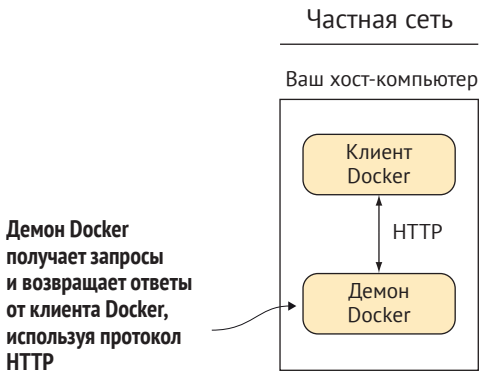


Рис. 2.2 ❖ Демон Docker

> **ПОДСКАЗКА.** *Демон* – это процесс, который выполняется в фоновом режиме, а не под непосредственным контролем пользователя. *Сервер* – процесс, который принимает запросы от клиента и осуществляет действия, необходимые для выполнения запросов. Демоны часто также являются серверами, принимающими запросы от клиентов для выполнения действий для них. Команда `docker` – это клиент, а демон Docker выступает в качестве сервера, выполняющего обработку ваших контейнеров и образов Docker.

Давайте рассмотрим пару методов, которые иллюстрируют, как Docker эффективно работает в качестве демона и как ваши взаимодействия с ним с помощью команды `docker` ограничиваются простыми запросами на выполнение действий, во многом как взаимодействия с веб-сервером. Первый метод позволяет другим подключаться к демону Docker и выполнять те же действия, что и на вашем хост-компьютере, а второй показывает, что контейнеры Docker управляются демоном, а не сеансом оболочки.

МЕТОД 1 Сделайте демон Docker доступным

Хотя по умолчанию ваш демон Docker доступен только на вашем хосте, могут быть веские причины, чтобы разрешить другим доступ к нему. У вас может быть проблема, и вы хотите, чтобы кто-то отлаживал ее удаленно, или у вас может возникнуть желание позволить другой части вашего рабочего процесса DevOps запускать процесс на хост-компьютере.

> **ВНИМАНИЕ!** Хотя этот метод может быть мощным и полезным, он считается небезопасным. Сокет Docker может быть использован любым, у кого есть доступ (включая контейнеры с подключенным сокетом Docker) для получения привилегий пользователя root.

ПРОБЛЕМА

Вы хотите открыть свой сервер Docker, чтобы у других был доступ.

РЕШЕНИЕ

Запустите демон Docker с открытым TCP-адресом.

На рис. 2.3 приводится обзор того, как работает этот метод.

Перед тем как открыть демон Docker, вы должны сначала закрыть работающий. То, как вы это сделаете, будет зависеть от вашей операционной системы (пользователям, не применяющим Linux, следует ознакомиться с приложением A). Если вы не уверены, как это сделать, начните с этой команды:

```
$ sudo service docker stop
```

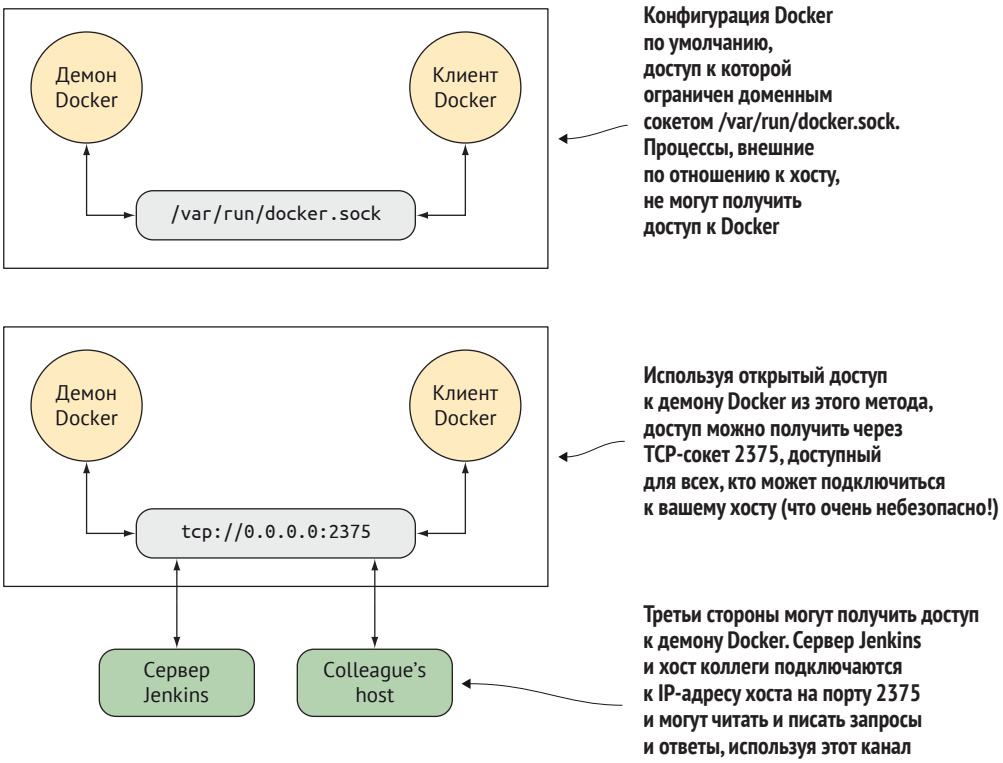


Рис. 2.3 ❖ Доступность Docker: нормальная и открытая

Если вы получите сообщение, которое выглядит так:

```
The service command supports only basic LSB actions (start, stop, restart,
try-restart, reload, force-reload, status). For other actions, please try
to use systemctl.
```

тогда у вас система запуска systemctl. Попробуйте эту команду:

```
$ systemctl stop docker
```

Если это сработает, вы не должны видеть никаких выводов этой команды:

```
$ ps -ef | grep -E 'docker(d| -d| daemon)\b' | grep -v grep
```

Как только демон Docker будет остановлен, можете перезапустить его вручную и открыть для внешних пользователей с помощью следующей команды:

```
$ sudo docker daemon -H tcp://0.0.0.0:2375
```

Эта команда запускается в качестве демона (`docker daemon`), определяет хост-сервер с помощью флага -H, использует протокол TCP, открывает все IP-интерфейсы (с 0.0.0.0) и стандартный порт сервера Docker (2375).

Вы можете подключиться снаружи с помощью следующей команды:

```
$ docker -H tcp://<your host's ip>:2375 <subcommand>
```

Или можете экспортировать переменную среды DOCKER_HOST (это не сработает, если вам нужно использовать команду sudo для запуска Docker, – см. метод 41, чтобы узнать, как устранить это требование):

```
$ export DOCKER_HOST=tcp://<your host's ip>:2375
$ docker <subcommand>
```

Обратите внимание, что вам также нужно будет выполнить одно из этих действий на локальном компьютере, поскольку Docker больше не прослушивает местоположение по умолчанию.

Если вы хотите, чтобы это изменение было постоянным на вашем хосте, необходимо будет настроить загрузочную систему (см. приложение B для получения информации о том, как это сделать).

> **ВНИМАНИЕ!** Если вы используете этот метод, чтобы ваш демон Docker слушал порт, имейте в виду, что, если указать IP как 0.0.0.0, это даст доступ пользователям на всех сетевых интерфейсах (как публичных, так и частных), что обычно считается небезопасным.

ОБСУЖДЕНИЕ

Это отличный метод, если у вас есть мощный компьютер, выделенный для Docker, в защищенной частной локальной сети, потому что каждый в сети может легко направить инструменты Docker в нужное место – DOCKER_HOST; это широко известная переменная среды, которая сообщит большинству программ, получающим доступ к Docker, где искать.

В качестве альтернативы несколько громоздкому процессу остановки службы Docker и ее запуска вручную вы можете объединить монтирование сокета Docker в качестве тома (из метода 45) с использованием утилиты socat для пересылки трафика из внешнего порта – просто выполните команду `docker run -p 2375: 2375 -v/var/run/docker.sock:/var/run/docker.sock sequenceid/socat`.

Вы увидите конкретный пример того, что позволяет этот метод, далее в этой главе, в методе 5.

МЕТОД 2 Запуск контейнеров в качестве демонов

Когда вы познакомитесь с Docker (и если вы чем-то похожи на нас), то начнете думать о других вариантах использования Docker, и одним из первых будет запуск контейнеров Docker в качестве фоновых служб.

Запуск контейнеров Docker в качестве служб с предсказуемым поведением через программную изоляцию является одним из основных вариантов использования Docker. Этот метод позволит вам управлять службами таким образом, чтобы это подходило для вашей работы.

ПРОБЛЕМА

Вы хотите запустить контейнер Docker в фоновом режиме как службу.

РЕШЕНИЕ

Используйте флаг `-d` в команде `docker run` и связанные флаги управления контейнерами для определения характеристик службы.

Контейнеры Docker, как и большинство процессов, будут запускаться по умолчанию на переднем плане. Наиболее очевидный способ запуска контейнера Docker в фоновом режиме – это использование стандартного оператора управления &. Хотя это и сработает, вы можете столкнуться с проблемами, если выйдете из сеанса терминала, что потребует использования флага `nohup`, создающего файл в вашем локальном каталоге с выводом, которым вы должны управлять … У вас есть идея: гораздо лучше применить для этого функциональность демона Docker.

Для этого используйте флаг `-d`.

```
$ docker run -d -i -p 1234:1234 --name daemon ubuntu:14.04 nc -l 1234
```

Флаг `-d`, когда используется с командой `docker run`, запускает контейнер в качестве демона. Флаг `-i` дает этому контейнеру возможность взаимодействовать с вашим сеансом Telnet. С помощью флага `-p` вы «публикуете» порт 1234 из контейнера на хост. Флаг `--name` позволяет присвоить контейнеру имя, чтобы вы могли обратиться к нему позже. В конце вы запускаете простой прослушивающий эхо-сервер на порту 1234 с помощью netcat (nc).

Теперь можно подключиться к нему и отправлять сообщения через Telnet. Вы увидите, что контейнер получил сообщение с помощью команды `docker logs`, как показано в следующем листинге.

Листинг 2.1. Подключение к серверу контейнера netcat с помощью Telnet

```
$ telnet localhost 1234
 Trying ::1...
Connected to localhost.
Escape character is '^]'.
hello daemon
^]

telnet> q
 Connection closed.
$ docker logs daemon
 hello daemon
```

Подключается к серверу контейнера netcat с помощью команды telnet

Вводит строку текста для отправки на сервер netcat

Нажимает Ctrl-], а затем клавишу Return, чтобы выйти из сеанса Telnet

Набирает q, а затем клавишу ввода, чтобы выйти из программы Telnet

Запускает команду docker logs для просмотра вывода контейнера

```
$ docker rm daemon
 daemon
$
```

Очищает контейнер с помощью команды rm

Видно, что запуск контейнера в качестве демона достаточно прост, но в оперативном отношении на некоторые вопросы еще предстоит ответить:

- Что происходит со службой, если она не работает?
- Что происходит со службой, когда она останавливается?
- Что происходит, если служба продолжает давать сбой снова и снова?

К счастью, Docker предоставляет флаги для каждого из этих вопросов!

> **ПРИМЕЧАНИЕ.** Хотя флаги перезапуска чаще всего используются с флагом демона (-d), запускать их с этим флагом не обязательно.

Флаг `docker run--restart` позволяет применять набор правил, которым необходимо следовать (так называемая стратегия повторного запуска), когда контейнер завершается (см. табл. 2.1).

Таблица 2.1 ❖ Параметры флагов повторного запуска Docker

Стратегия	Описание
`no`	Не перезапускать при выходе контейнера
`always`	Всегда перезапускать при выходе контейнера
`unless-stopped`	Всегда перезагружать, но помнить о явной остановке
`on-failure[:max-retry]`	Перезапускать только в случае сбоя

Стратегия `no` проста: при завершении работы (выходе) контейнера контейнер не перезапускается. Это по умолчанию.

Стратегия `always` также проста, но стоит кратко обсудить ее:

```
$ docker run -d --restart=always ubuntu echo done
```

Эта команда запускает контейнер в качестве демона (`-d`) и всегда перезапускает его по завершении (`--restart=always`). Она выдает простую команду `echo`, которая быстро завершается, выходя из контейнера.

Если вы выполните предыдущую команду, а затем команду `docker ps`, вы увидите вывод, подобный этому:

```
$ docker ps
CONTAINER ID      IMAGE             COMMAND          CREATED
➥     STATUS                     PORTS            NAMES
69828b118ec3      ubuntu:14.04      "echo done"      4 seconds ago
➥     Restarting (0) Less than a second ago       sick_brattain
```

Команда `docker ps` выводит список всех запущенных контейнеров и информацию о них, включая следующее:

- когда был создан контейнер (`CREATED`);
- текущее состояние контейнера – обычно это `Restarting`, потому что он будет работать только в течение короткого времени (`STATUS`);
- код выхода предыдущего запуска контейнера (также в разделе `STATUS`). 0 означает, что запуск прошел успешно;
- название контейнера. По умолчанию Docker присваивает имена контейнерам, объединяя два случайных слова. Иногда это приводит к странным результатам (именно поэтому мы обычно рекомендуем давать им осмысленные имена).

Обратите внимание, что столбец `STATUS` также сообщил нам, что контейнер вышел менее секунды назад и перезапускается. Это связано с тем, что команда `echo done` завершается немедленно, и Docker должен постоянно перезапускать контейнер.

Важно отметить, что Docker повторно использует идентификатор контейнера. Он не изменяется при перезапуске, и для этого вызова Docker всегда будет только одна запись в таблице `ps`.

Параметр `unless-stopped` почти то же самое, что и `always` – оба прекратят перезапуск, если вы выполните команду `docker stop` для контейнера, но `unless-stopped` удостоверится, что это остановленное состояние запоминается при повторном запуске демона (возможно, при перезагрузке компьютера), в то время как `always` снова вернет контейнер.

Наконец, стратегия `on-failure` перезапускается только тогда, когда контейнер возвращает ненулевой код завершения (что обычно означает сбой) из своего основного процесса:

```
$ docker run -d --restart=on-failure:10 ubuntu /bin/false
```

Эта команда запускает контейнер как демон (`-d`) и устанавливает ограничение на количество попыток перезапуска (`--restart = on-failure: 10`), выходя, если оно превышено. Она запускает простую команду (`/ bin / false`), которая быстро завершается и непременно потерпит неудачу.

Если вы выполните предыдущую команду и подождете минуту, а затем – команду `docker ps -a`, то увидите вывод, подобный этому:

```
$ docker ps -a
CONTAINER ID      IMAGE             COMMAND           CREATED
➥     STATUS                     PORTS               NAMES
b0f40c410fe3      ubuntu:14.04      "/bin/false"      2 minutes ago
➥     Exited (1) 25 seconds ago                      loving_rosalind
```

ОБСУЖДЕНИЕ

Создание служб для запуска в фоновом режиме часто сопряжено с трудностями обеспечения того, что фоновая служба не будет работать в необычных средах. Поскольку он не виден сразу, пользователи могут не заметить, что что-то работает неправильно.

Этот метод позволяет вам перестать думать о непредвиденной сложности вашего сервиса, вызванной перезапуском среды и обработкой. Можете сосредоточить свое внимание на базовой функциональности.

В качестве конкретного примера вы и ваша команда можете использовать этот метод для запуска нескольких баз данных на одном компьютере и избежать необходимости писать инструкции по их настройке или открывать терминалы, чтобы поддерживать их работу.

МЕТОД 3 Перемещение Docker в другой раздел

Docker хранит все данные, относящиеся к вашим контейнерам и образам в папке. Поскольку он способен хранить потенциально большое количество различных образов, эта папка может быстро увеличиться!

Если у вашего хост-компьютера есть разные разделы (как это часто бывает на рабочих станциях Linux для предприятий), вы гораздо быстрее столкнетесь с ограничением пространства. В этих случаях можете захотеть переместить каталог, из которого работает Docker.

ПРОБЛЕМА

Вы хотите перейти туда, где Docker хранит свои данные.

РЕШЕНИЕ

Остановите и запустите демон Docker, указав новое местоположение с помощью флага `-g`.

Представьте, что вы хотите запустить Docker из /home/dockeruser/mydocker. Сначала остановите ваш демон Docker (см. приложение B, где обсуждается как это сделать). Затем выполните следующую команду:

```
$ dockerd -g /home/dockeruser/mydocker
```

В каталоге будет создан новый набор папок и файлов. Эти папки являются внутренними для Docker, поэтому экспериментируйте с ними на свой (как выяснилось!) страх и риск.

Вы должны знать, что эта команда появится, чтобы стереть контейнеры и образы из вашего предыдущего демона Docker. Не отчаивайтесь. Если вы уничтожите только что запущенный процесс Docker и перезапустите службу Docker, клиент Docker будет направлен в исходное местоположение, а контейнеры и образы вернутся к вам. Если вы хотите сделать этот переход постоянным, нужно соответствующим образом настроить процесс запуска вашей хост-системы.

ОБСУЖДЕНИЕ

Помимо очевидного варианта использования этого способа (освобождение места на дисках с ограниченным дисковым пространством), вы также можете применить этот метод, если хотите строго разделить наборы образов и контейнеров. Например, если у вас есть доступ к нескольким частным реестрам Docker с разными владельцами, возможно, стоит приложить дополнительные усилия, чтобы убедиться, что вы случайно не передаете личные данные не тому человеку.

2.3. Клиент Docker

Клиент Docker (см. рис. 2.4) – самый простой компонент в архитектуре Docker. Это то, что вы запускаете, когда набираете такие команды, как `docker run` или `docker pull` на своем компьютере. Его задача – взаимодействовать с демоном Docker посредством HTTP-запросов.

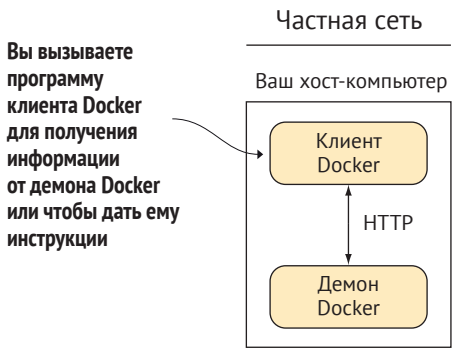


Рис. 2.4 ❖ Клиент Docker

В этом разделе вы узнаете, как можно отслеживать сообщения между клиентом и сервером Docker. Вы также увидите способ использования своего браузера в качестве клиента Docker и несколько основных приемов, связанных с пробросом портов, которые представляют собой маленькие шажки в сторону оркестровки, обсуждаемой в четвертой части этой книги.

МЕТОД 4 Использование socat для мониторинга трафика Docker API

Иногда команда `docker` может работать не так, как вы ожидаете. Чаще всего непонятны некоторые аспекты аргументов командной строки, но иногда возникают более серьезные проблемы с настройкой, такие как устаревший двоичный файл Docker.

Для диагностики проблемы может быть полезно просмотреть поток данных, поступающих к демону Docker и от него, с которым вы общаетесь.

ПРИМЕЧАНИЕ. Не паникуйте! Наличие этого метода не означает, что Docker нужно часто отлаживать или что он в любом случае нестабилен! Данный метод используется здесь в первую очередь как инструмент для понимания архитектуры Docker, а также для знакомства с мощной утилитой socat. Если, как и мы, вы используете Docker в разных местах, версии Docker будут различаться. Как в случае с любым программным обеспечением, в разных версиях будут разные функции и флаги, которые могут вас «подловить».

ПРОБЛЕМА

Вы хотите отладить проблему с помощью команды Docker.

РЕШЕНИЕ

Используйте программу для контроля сетевого трафика, чтобы проверять API-вызовы и создавать свои собственные.

В этом методе вы вставите доменный сокет Unix между вашим запросом и сокетом сервера, чтобы посмотреть, что проходит через него (как показано на рис. 2.5). Обратите внимание, для этого вам понадобятся привилегии пользователя root или sudo.

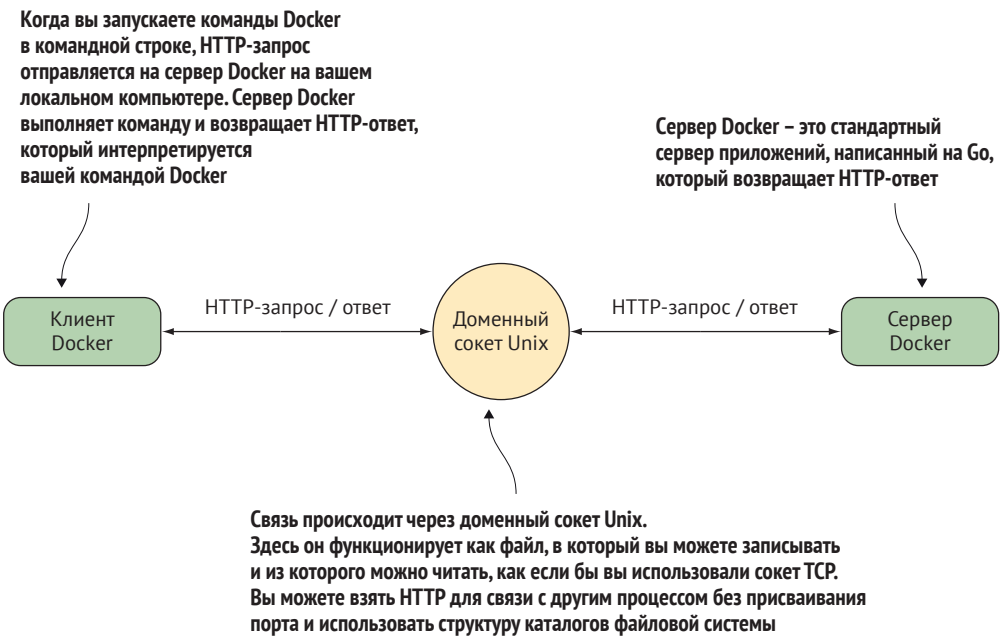


Рис. 2.5 ❖ Архитектура клиент/сервер Docker на вашем хосте

Для создания этого прокси вы будете использовать socat.

ПОДСКАЗКА. `socat` – это мощная команда, которая позволяет передавать данные практически любого типа между двумя каналами данных. Если вы знакомы с netcat, то можете рассматривать ее как netcat на стероидах. Чтобы установить ее, используйте стандартный менеджер пакетов для вашей системы.

```
$ socat -v UNIX-LISTEN:/tmp/dockerapi.sock,fork \
  UNIX-CONNECT:/var/run/docker.sock &
```

В этой команде `-v` делает вывод читаемым с указанием потока данных.

Часть `UNIX-LISTEN` говорит `socat` прослушивать сокет Unix, `fork` гарантирует, что `socat` не завершит работу после первого запроса, а `UNIX-CONNECT` сообщает `socat` подключиться к Unix-сокету Docker. `&` указывает, что команда выполняется в фоновом режиме. Если вы обычно запускаете клиент Docker с помощью sudo, вам нужно сделать то же самое и здесь.

Новый маршрут, по которому будут проходить ваши запросы к демону, показан на рис. 2.6.

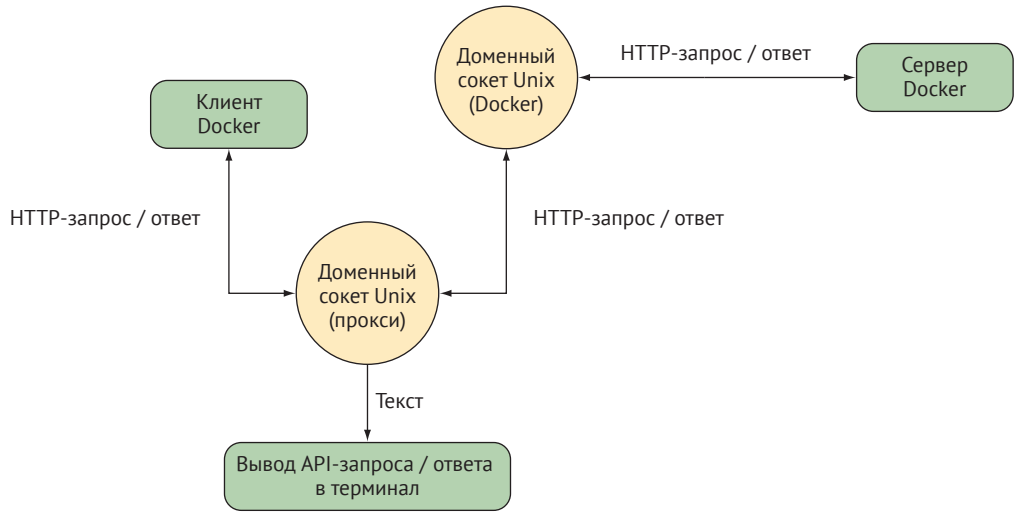


Рис. 2.6 ❖ Клиент и сервер Docker с socat, вставленным в качестве прокси

Весь трафик, проходящий в каждом направлении, `socat` будет видеть, и он будет зарегистрирован на вашем терминале в дополнение к любому выводу, который предоставляет клиент Docker.

Вывод простой команды `docker` теперь будет выглядеть примерно так:

```
$ docker -H unix:///tmp/dockerapi.sock ps -a
 > 2017/05/1516:01:51.163427 length=83 from=0 to=82
GET /_ping HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/17.04.0-ce (linux)\r
```

Команда, которую вы вводите, чтобы увидеть запрос и ответ

```
\r
< 2017/05/1516:01:51.164132 length=215 from=0 to=214
HTTP/1.1200 OK\r
Api-Version: 1.28\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/17.04.0-ce (linux)\r
Date: Mon, 15 May 201715:01:51 GMT\r
Content-Length: 2\r
Content-Type: text/plain; charset=utf-8\r
\r
OK> 2017/05/1516:01:51.165175 length=105 from=83 to=187
 GET /v1.28/containers/json?all=1 HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/17.04.0-ce (linux)\r
\r
< 2017/05/1516:01:51.165819 length=886 from=215 to=1100
 HTTP/1.1200 OK\r
Api-Version: 1.28\r
Content-Type: application/json\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/17.04.0-ce (linux)\r
Date: Mon, 15 May 201715:01:51 GMT\r
Content-Length: 680\r
\r
[{"Id":"1d0d5b5a7b506417949653a59deac030ccbcbb816842a63ba68401708d55383e",
➥ "Names":["/example1"],"Image":"todoapp","ImageID":
➥ "sha256:ccdda5b6b021f7d12bd2c16dbcd2f195ff20d10a660921db0ac5bff5ecd92bc2",
➥ "Command":"npm start","Created":1494857777,"Ports":[],"Labels":{},
➥ "State":"exited","Status":"Exited (0) 45 minutes ago","HostConfig":
➥ {"NetworkMode":"default"},"NetworkSettings":{"Networks":{"bridge":
➥ {"IPAMConfig":null,"Links":null,"Aliases":null,"NetworkID":
➥ "6f327d67a38b57379afa7525ea63829797fd31a948b316fdf2ae0365faeed632",
➥ "EndpointID":"","Gateway":"","IPAddress":"","IPPrefixLen":0,
➥ "IPv6Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0,
➥ "MacAddress":""}}},"Mounts":[]}]
CONTAINER ID        IMAGE         COMMAND           CREATED
➥     STATUS                     PORTS         NAMES
1d0d5b5a7b50        todoapp       "npm start"       45 minutes ago
➥     Exited (0) 45 minutes ago              example1
```

HTTP-запрос начинается здесь, со знака > слева

HTTP-ответ начинается здесь, со знака < слева

Содержимое ответа в формате JSON от сервера Docker

Вывод, каким его обычно видит пользователь, интерпретируется клиентом Docker из предыдущего вывода

Особенности предыдущего вывода будут меняться по мере роста и разработки Docker API. Выполнив предыдущую команду, вы увидите более поздние номера версий и другой вывод в формате JSON. Можете проверить свои клиентские и серверные версии API, выполнив команду `docker version`.

> **ВНИМАНИЕ!** Если в предыдущем примере вы запустили socat от имени пользователя root, вам потребуется использовать sudo для запуска команды `docker -H`. Это связано с тем, что файл dockerapi.sock принадлежит пользователю root.

Использование socat – это мощный способ отладки не только Docker, но и любых других сетевых сервисов, с которыми вы можете столкнуться в ходе своей работы.

ОБСУЖДЕНИЕ

Для этого метода можно придумать ряд других вариантов использования:

- Socat – своего рода швейцарский армейский нож. Она может обрабатывать множество различных протоколов. Предыдущий пример показывает, что она прослушивает сокет Unix, но вы также можете заставить ее прослушивать внешний порт с помощью `TCP-LISTEN: 2375, fork` вместо аргумента `UNIX-LISTEN:…`. Это действует как более простая версия метода 1. При таком подходе нет необходимости перезапускать демон Docker (что уничтожит все работающие контейнеры). Вы можете просто включать и отключать слушателя socat по своему желанию.
- Поскольку предыдущий пункт очень прост в настройке и является временным, вы можете использовать его в сочетании с методом 47, чтобы удаленно присоединиться к работающему контейнеру коллег и помочь им отладить проблему. Также можно применить малоиспользуемую команду `docker attach`, чтобы присоединиться к тому же терминалу, который они запустили с помощью команды `docker run`, что позволит вам сотрудничать напрямую.
- Если у вас есть общий сервер Docker (возможно, настроенный по методу 1), вы можете открыть доступ к внешним файлам и установить socat в качестве посредника между внешним миром и сокетом Docker, чтобы он действовал как примитивный журнал проверок, записывая, откуда приходят все запросы и что они делают.

МЕТОД 5 Использование Docker в вашем браузере

Продавать новые технологии может быть сложно, поэтому простые и эффективные демонстрации неоценимы. Создание демонстрационной версии еще лучше, поэтому мы обнаружили, что создание веб-страницы, которая позволяет пользователям взаимодействовать с контейнером в браузере, является отличным методом. Это позволяет новичкам впервые попробовать

Docker легкодоступным способом, позволяя им создавать контейнер, использовать контейнерный терминал в браузере, подсоединяться к чужому терминалу и осуществлять совместное управление. Значительный вау-фактор тоже не повредит!

ПРОБЛЕМА

Вы хотите продемонстрировать мощь Docker, не требуя, чтобы пользователи сами устанавливали его или запускали команды, которые они не понимают.

РЕШЕНИЕ

Запустите демон Docker с открытым портом и задействуйте совместное использование ресурсов между разными источниками, а затем обслуживайте репозиторий Docker-терминала на выбранном вами веб-сервере.

Наиболее распространенное использование REST API – это открыть его на сервере и использовать JavaScript на веб-странице для выполнения вызовов. Поскольку Docker выполняет все взаимодействия через REST API, вы должны иметь возможность контролировать Docker таким же образом. Хотя поначалу это может показаться удивительным, этот контроль распространяется на весь процесс взаимодействия с контейнером через терминал в вашем браузере.

Мы уже обсуждали, как запустить демон на порту 2375 в методе 1, поэтому не будем вдаваться в подробности. Кроме того, совместное использование ресурсов между разными источниками – довольно обширная тема, чтобы подробно рассказывать о ней здесь. Если вы не знакомы с ним, обратитесь к книге Монсура Хуссейна *CORS в действии* (Manning, 2014). Говоря кратко, совместное использование ресурсов между разными источниками – это механизм, который тщательно обходит обычное ограничение JavaScript, ограничивающее вас только доступом к текущему домену. В этом случае совместное использование ресурсов позволяет демону прослушивать другой порт, с которого вы обслуживаете страницу терминала Docker. Чтобы включить совместное использование ресурсов, нужно запустить демон Docker с помощью параметра `--api-enable-cors` вместе с параметром, чтобы он прослушивал порт.

Теперь, когда предварительные условия отсортированы, давайте начнем. Во-первых, вам нужно получить код:

```
git clone https://github.com/aidanhs/Docker-Terminal.git
cd Docker-Terminal
```

Затем необходимо обслужить файлы:

```
python2 -m SimpleHTTPServer 8000
```

Предыдущая команда использует модуль, встроенный в Python, для обслуживания статических файлов из каталога. Не стесняйтесь использовать любой эквивалент, который вы предпочитаете. Теперь перейдите по адресу http://localhost:8000 в своем браузере и запустите контейнер.

На рис. 2.7 показано, как подключается терминал Docker. Страница размещена на вашем локальном компьютере и подключается к демону Docker на локальном компьютере для выполнения любых операций.

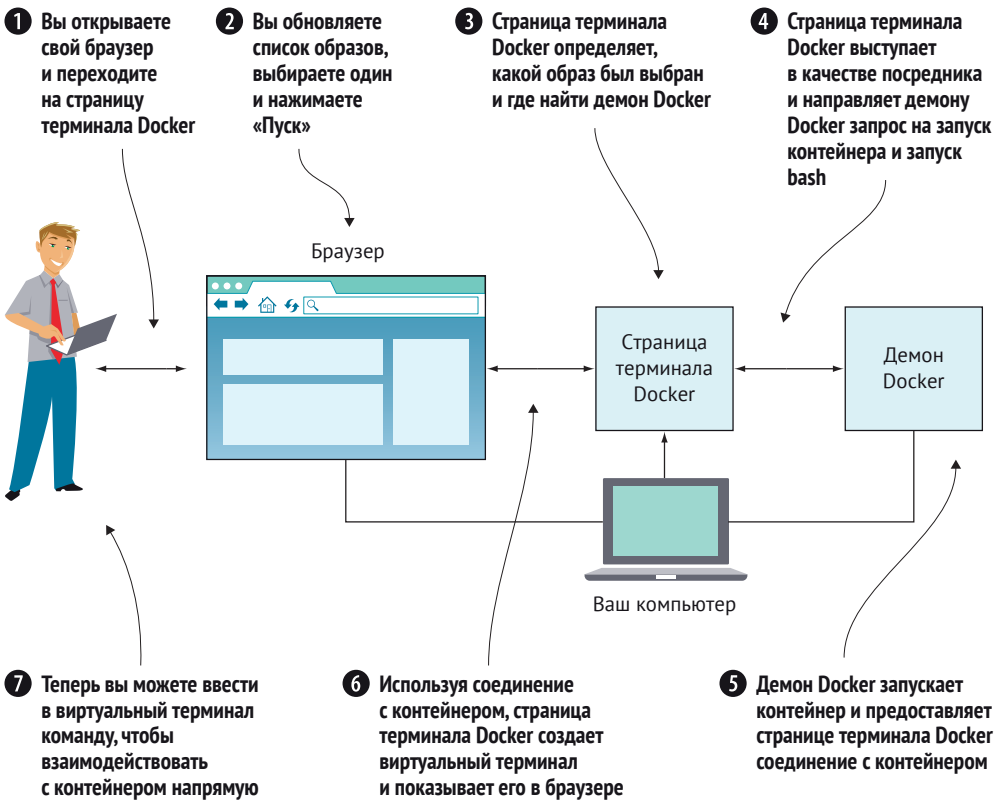


Рис. 2.7 ❖ Как работает терминал Docker

Стоит помнить о следующих моментах, если вы хотите дать эту ссылку другим людям:

- другой человек не должен использовать прокси-сервер какого-либо типа. Это наиболее распространенный источник ошибок, с которым мы встречались – терминал Docker использует вебсокеты, которые в настоящее время не работают через прокси;
- ссылка на `localhost`, очевидно, не сработает – вам нужно будет указать внешний IP-адрес;
- терминал Docker должен знать, где найти Docker API, – он должен делать это автоматически на основе адреса, по которому вы переходите в браузере, но об этом нужно помнить.

ПОДСКАЗКА. Если вы более опытный пользователь Docker, можете спросить, почему мы не использовали образ Docker в этом методе. Причина состоит в том, что мы все еще знакомим читателей с Docker и не хотим повышать сложность для новичков. Докеризация этого метода предоставляется читателю в качестве упражнения.

ОБСУЖДЕНИЕ

Хотя мы первоначально применяли этот метод в качестве захватывающей демонстрации для Docker (сделать так, чтобы несколько человек могли с легкостью совместно использовать терминал на одноразовых компьютерах, даже с терминальными мультиплексорами – задача непростая), мы нашли интересные приложения в несвязанных областях. Один из примеров -это его использование для мониторинга небольшой группы стажеров в какой-нибудь задаче в командной строке. Вам или им не нужно ничего устанавливать – просто откройте браузер, и можете подключиться к их терминалу, чтобы присоединиться и помочь в любое время!

Также следует отметить, что тут есть некоторые сильные стороны в плане сотрудничества. В прошлом, когда нужно было поделиться ошибкой с коллегой, мы воспроизводили эту ошибку в контейнере Docker, чтобы отследить ее вместе. С помощью этого метода не нужно заранее обсуждать возможные вопросы типа «а зачем мне Docker?».

МЕТОД 6 Использование портов для подключения к контейнерам

Контейнеры Docker изначально разрабатывались для запуска служб. В большинстве случаев это те или иные HTTP-службы. Значительную часть из них составляют веб-сервисы, доступные через браузер.

Это приводит к проблеме. Если у вас есть несколько Docker-контейнеров, работающих на порту 80 во внутренней среде, они не могут быть доступны через порт 80 на вашем хост-компьютере. Метод показывает, что вы можете управлять этим общим сценарием, предоставляя доступ к порту и настраивая его проброс.

ПРОБЛЕМА

Вы хотите сделать несколько служб контейнера Docker доступными для порта на вашем хост-компьютере.

РЕШЕНИЕ

Используйте флаг `-p` для отображения порта контейнера в свой хост-компьютер.

В этом примере мы будем использовать образ tutum-wordpress. Допустим, вы хотите запустить два из них на своем хост-компьютере для обслуживания разных блогов.

Поскольку есть люди, пожелавшие сделать это раньше, кто-то подготовил образ, который каждый может получить и запустить. Чтобы получить образы

из внешних источников, вы можете использовать команду `docker pull`. По умолчанию образы будут загружаться из Docker Hub:

```
$ docker pull tutum/wordpress
```

Образы также будут получены автоматически, когда вы попытаетесь запустить их, если их еще нет на вашем компьютере.

Чтобы запустить первый блог, используйте следующую команду:

```
$ docker rnu -d -p 10001:80 --name blog1 tutum/wordpress
```

Команда `docker run` запускает контейнер как демон (`-d`) с флагом (`-p`). Она идентифицирует порт хоста (`10001`) для отображения в порт контейнера (`80`) и присваивает контейнеру имя, чтобы его идентифицировать (`--name blog1 tutum/wordpress`).

Вы можете сделать то же самое для второго блога:

```
$ docker run -d -p 10002:80 --name blog2 tutum/wordpress
```

Если вы сейчас выполните эту команду:

```
$ docker ps | grep blog
```

то увидите два перечисленных контейнера блогов с пробросом портов, которые выглядят примерно так:

```
$ docker ps | grep blog
9afb95ad3617 tutum/wordpress:latest "/run.sh" 9 seconds ago
➥ Up 9 seconds  3306/tcp, 0.0.0.0:10001->80/tcp blog1
31ddc8a7a2fd tutum/wordpress:latest "/run.sh" 17 seconds ago
➥ Up 16 seconds 3306/tcp, 0.0.0.0:10002->80/tcp blog2
```

Теперь можно получить доступ к своим контейнерам, перейдя по адресам: http://localhost:10001 и http://localhost:10002.

Чтобы удалить контейнеры по окончании (при условии, что вы не хотите их сохранять, – мы будем использовать их в следующем методе), выполните следующую команду:

```
$ docker rm -f blog1 blog2
```

Теперь вы сможете запускать несколько идентичных образов и служб на своем хосте, самостоятельно управляя распределением портов, если это необходимо.

> **ПОДСКАЗКА.** При использовании флага `-p` можно легко забыть, какой порт является хостом, а какой – контейнером. Мы представляем себе это как читать предложение слева направо. Пользователь подключается к хосту (`-p`), и порт хоста передается на порт контейнера (`host_port: container_port`). Кроме того, это тот же формат, что и SSH-команды для переадресации портов, если вы с ними знакомы.

ОБСУЖДЕНИЕ

Открытие портов является невероятно важной частью множества вариантов использования Docker, и вы будете неоднократно встречаться с этим на протяжении книги, особенно в части 4, где контейнеры, общающиеся друг с другом, являются частью повседневной жизни.

В методе 80 мы познакомим вас с виртуальными сетями и объясним, что они делают за кулисами и как направляют порты хоста в нужный контейнер.

МЕТОД 7 Разрешение связи между контейнерами

Предыдущий метод показал, как вы можете развернуть свои контейнеры для хост-сети, открыв порты. Вам не всегда захочется предоставлять свои службы хост-компьютеру или внешнему миру, но вы захотите соединить контейнеры друг с другом.

Данный метод показывает, как можно достичь этого с помощью функции пользовательских сетей Docker, гарантирующей, что посторонние не смогут получить доступ к вашим внутренним службам.

ПРОБЛЕМА

Вы хотите разрешить связь между контейнерами для внутренних целей.

РЕШЕНИЕ

Используйте пользовательские сети, чтобы контейнеры могли взаимодействовать друг с другом.

Пользовательские сети простые и гибкие. У нас есть пара блогов на WordPress, работающих в контейнерах из предыдущего метода, поэтому давайте посмотрим, как можно получить к ним доступ из другого контейнера (а не из внешнего мира, который вы уже видели).

Для начала нужно создать пользовательскую сеть:

```
$ docker network create my_network
0c3386c9db5bb1d457c8af79a62808f78b42b3a8178e75cc8a252fac6fdc09e4
```

С помощью этой команды мы создаем новую виртуальную сеть на вашем компьютере, которую можно использовать для управления взаимодействием между контейнерами. По умолчанию все контейнеры, которые вы подключаете к этой сети, смогут видеть друг друга по их именам.

Затем, предположив, что у вас все еще есть контейнеры `blog1` и `blog2` из предыдущего метода, вы можете подключить один из них к вашей новой сети на лету.

```
$ docker network connect my_network blog1
```

Наконец, можно запустить новый контейнер, явно указав сеть, и посмотреть, сможете ли вы извлечь первые пять строк HTML-кода из целевой страницы блога.

```
$ docker run -it --network my_network ubuntu:16.04 bash
root@06d6282d32a5:/# apt update && apt install -y curl
[...]
root@06d6282d32a5:/# curl -sSL blog1 | head -n5
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US">
<head>
        <meta name="viewport" content="width=device-width" />
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
root@06d6282d32a5:/# curl -sSL blog2
curl: (6) Could not resolve host: blog2
```

> **ПОДСКАЗКА.** Давать имена контейнерам очень полезно для присваивания запоминаемых имен хостов, к которым вы можете обратиться позже, но это не является строго необходимым – если соединения только исходящие, то вам, вероятно, не нужно искать контейнер. Если вы обнаружите, что все же хотите найти хост и не присвоили имя, то можете использовать короткий идентификатор образа, указанный в приглашении терминала (если только он не был переопределен именем хоста) или в выводе `docker ps`.

Новый контейнер успешно смог получить доступ к блогу, что мы подключили к `my_network`, отображая часть HTML-кода страницы, которую мы увидели бы, если бы зашли на нее в браузере. С другой стороны, новый контейнер не смог увидеть `blog2`. Поскольку мы так и не подключили его к `my_network`, это имеет смысл.

ОБСУЖДЕНИЕ

Вы можете использовать этот метод для установки любого количества контейнеров в кластере в собственной частной сети, требуя только, чтобы у них был какой-то способ для обнаружения имен друг друга. В методе 80 вы увидите способ сделать это, который хорошо интегрируется с сетями Docker. Между тем в следующем методе мы начнем с малого, демонстрируя преимущества возможности явного соединения между отдельным контейнером и предоставляемым им сервисом.

Еще одно замечание касается интересного конечного состояния контейнера `blog1`. Все контейнеры по умолчанию подключены к сети Docker `bridge`, поэтому, когда мы попросили его присоединиться к `my_network`, он сделал это

в дополнение к сети, в которой он уже был. В методе 80 мы рассмотрим это более подробно, чтобы увидеть, как можно использовать *расслоение сетей* в качестве модели в реальных ситуациях.

МЕТОД 8 Установление соединений между контейнерами для изоляции портов

В предыдущем методе вы видели, как заставить контейнеры обмениваться данными с пользовательскими сетями. Но есть более старый метод объявления обмена данными между контейнерами – флаг соединений Docker. Он больше не рекомендуется к использованию, но данный метод был частью Docker, и о нем стоит знать, если вы столкнетесь с ним в естественных условиях.

ПРОБЛЕМА

Вы хотите разрешить обмен данными между контейнерами без использования пользовательских сетей.

РЕШЕНИЕ

Используйте функции соединений Docker, чтобы контейнеры могли общаться друг с другом.

Взяв за основу пример с WordPress, мы отделим слой базы данных MySQL от контейнера WordPress и свяжем их друг с другом без настройки порта или создания сети. На рис. 2.8 приводится обзор окончательного состояния.

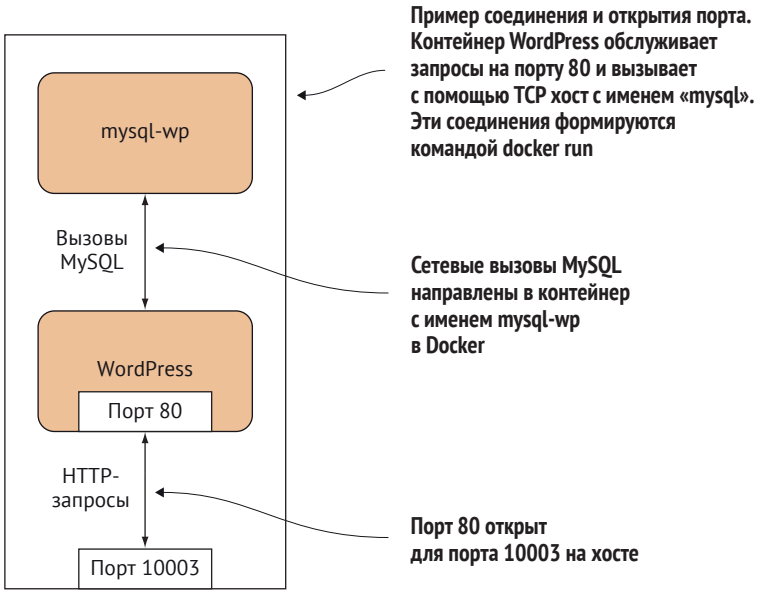


Рис. 2.8 ❖ Настройка WordPress с соединенными контейнерами

Зачем беспокоиться об установлении соединений, если вы уже можете предоставить порты хосту и использовать это? Установление соединений позволяет инкапсулировать и определять отношения между контейнерами, не подвергая сервисы сети хоста (и, вероятно, внешнему миру). Возможно, вы захотите сделать это, например, по соображениям безопасности.

Запустите ваши контейнеры в следующем порядке, делая паузу длиной примерно минуту между первой и второй командами:

```
$ docker run --name wp-mysql \
  -e MYSQL_ROOT_PASSWORD=yoursecretpassword -d mysql
$ docker run --name wordpress \
  --link wp-mysql:mysql -p 10003:80 -d wordpress
```

Сначала вы даете контейнеру MySQL имя `wp-mysql`, чтобы иметь возможность обратиться к нему позже. Также нужно указать переменную среды, чтобы этот контейнер мог инициализировать базу данных (`-e MYSQL_ROOT_PASSWORD=yoursecretpassword`). Вы запускаете оба контейнера в качестве демонов (`-d`) и используете ссылку Docker Hub для официального образа MySQL.

Во второй команде вы даете образу WordPress имя `wordpress` на случай, если захотите обратиться к нему позже, а также соединяете контейнер `wp-mysql` с контейнером WordPress (`--link wp-mysql: mysql`). Ссылки на сервер MySQL в контейнере Word-Press будут отправлены в контейнер `wp-mysql`. Вы также используете проброс локальных портов (`-p 10003:80`), как описано в методе 6, и добавляете ссылку Docker Hub для официального образа WordPress (`wordpress`). Имейте в виду, что соединения не будут ждать запуска служб в связанных контейнерах; отсюда и инструкция для паузы между командами. Более точный способ сделать это – поискать фразу «mysqid: ready for connection» в выводе `docker logs wp-mysql` перед запуском контейнера WordPress.

Если вы сейчас перейдете по адресу http: // localhost:10003, то увидите вводный экран Word-Press и сможете настроить свой экземпляр WordPress.

Основой этого примера является флаг `--link` во второй команде. Этот флаг устанавливает хост-файл контейнера, чтобы контейнер WordPress мог обращаться к серверу MySQL, и он будет перенаправлен на любой контейнер с именем `wp-mysql`, что дает значительное преимущество. Оно заключается в том, что различные контейнеры MySQL могут быть заменены без каких-либо изменений в контейнере WordPress, а это значительно упрощает управление конфигурацией различных служб.

> **ПРИМЕЧАНИЕ.** Контейнеры должны быть запущены в правильном порядке, чтобы имел место проброс для имен контейнеров, которые уже существуют. Динамическое разрешение соединений не является (на момент написания этих строк) особенностью Docker.

Чтобы контейнеры могли быть соединены таким способом, их порты должны быть указаны как открытые при сборке образов. Это делается с помощью команды `EXPOSE` в файле сборки образа Dockerfile. Порты, перечисленные

в директивах `EXPOSE` в файлах Dockerfile, также применяются при использовании флага `-P` («объявлять все порты открытыми» вместо `-P`, который объявляет открытым определенный порт) для команды `docker run`.

Запуская разные контейнеры в определенном порядке, вы познакомились с простым примером оркестровки Docker. Оркестровка Docker – это любой процесс, координирующий работу контейнеров Docker. Это большая и важная тема, которую мы подробно рассмотрим в четвертой части этой книги.

Разделив свою рабочую нагрузку на отдельные контейнеры, вы сделали шаг к созданию архитектуры микросервисов для своего приложения. В этом случае можете работать с контейнером MySQL, не трогая контейнер WordPress, или наоборот. Такой детальный контроль над запущенными сервисами является одним из ключевых эксплуатационных преимуществ архитектуры микросервисов.

ОБСУЖДЕНИЕ

Такой точный контроль над набором контейнеров требуется не часто, но может быть полезен в качестве очень простого и понятного способа замены контейнеров. Используя пример из этого метода, вы можете протестировать другую версию MySQL – образу WordPress не нужно ничего знать об этом, потому что он просто ищет ссылку `mysql`.

2.4. Реестры Docker

После того как вы создали свои образы, можете поделиться ими с другими пользователями. Вот тут-то и вступает в действие концепция *реестра Docker*.

Три реестра на рис. 2.9 различаются по своей доступности. Один находится в частной сети, один открыт во внешней сети, а другой общедоступен, но только для тех, кто зарегистрирован в Docker. Все они выполняют одну и ту же функцию с одним и тем же API, и вот откуда демон Docker знает, как обмениваться с ними информацией на взаимозаменяемой основе.

Реестр Docker позволяет нескольким пользователям размещать и извлекать образы из центрального хранилища, используя RESTful API.

Код реестра, как и сам Docker, – это ПО с открытым исходным кодом. Многие компании (как, например, наша) создают частные реестры для внутреннего хранения и совместного использования своих собственных образов. Это то, что мы обсудим, прежде чем более внимательно изучить реестр Docker Inc.

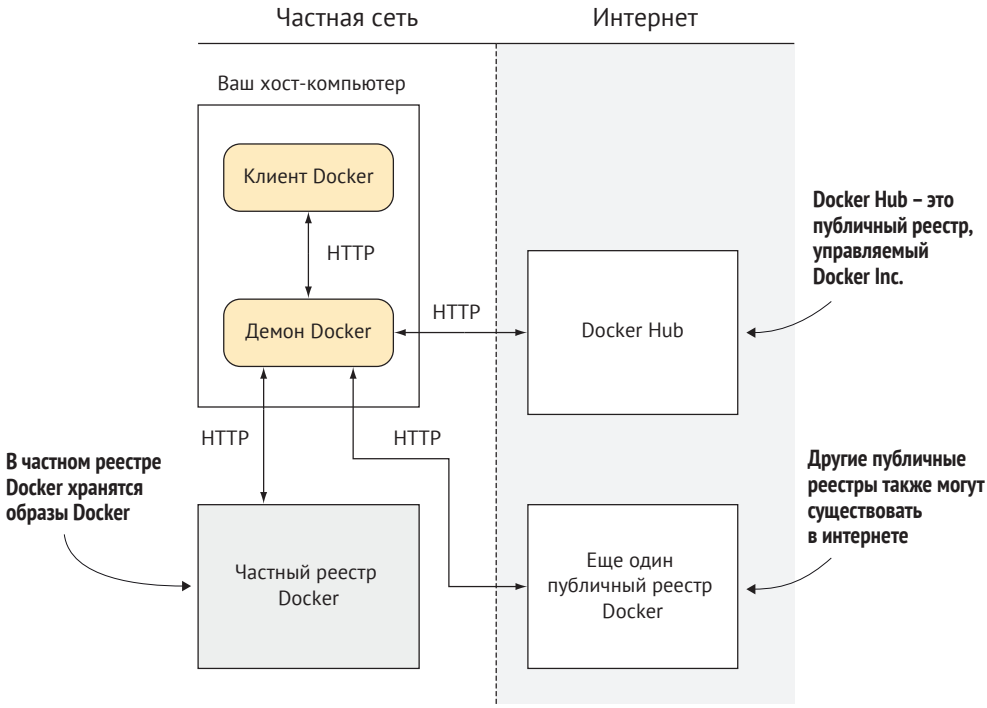


Рис. 2.9 ❖ Реестр Docker

МЕТОД 9 Настройка локального реестра Docker

Вы видели, что у Docker Inc. есть сервис, где люди могут в открытую совместно использовать образы (и вы можете заплатить, если желаете делать это в частном порядке). Но есть ряд причин, по которым вам, возможно, нужно совместно использовать образы, не обращаясь к хабу, – некоторые компании предпочитают хранить как можно больше внутренних данных, может быть, ваши образы большие и будут передаваться по интернету слишком медленно, или вы хотите сохранить конфиденциальность образов, пока экспериментируете, и не желаете платить. Независимо от причины, есть простое решение.

ПРОБЛЕМА

Вам нужен способ локального размещения ваших образов.

РЕШЕНИЕ

Настройте сервер реестра в своей локальной сети. Просто введите следующую команду на компьютере с большим количеством дискового пространства:

```
$ docker run -d -p 5000:5000 -v $HOME/registry:/var/lib/registry registry:2
```

Эта команда делает реестр доступным на порту 5000 хоста Docker (`-p 5000:5000`). С помощью флага -v он делает папку реестра на вашем хосте (`/var/lib/registry`) доступной в контейнере как `$ HOME/registry`. Поэтому файлы реестра будут храниться на хосте в папке `/var/lib/registry`.

На всех компьютерах, на которых вы хотите получить доступ к этому реестру, добавьте следующее в параметры демона (где `HOSTNAME` – это имя хоста или IP-адрес сервера вашего нового реестра): `--insecure-registry HOSTNAME` (подробную информацию о том, как это сделать, см. в приложении B). Теперь вы можете выполнить следующую команду: `docker push HOSTNAME:5000/ image:tag`.

Как видно, самый базовый уровень конфигурации локального реестра со всеми данными, хранящимися в каталоге `$HOME/registry`, прост. Если вы хотите увеличить масштаб или сделать его более надежным, репозиторий на GitHub (https://github.com/docker/distribution/blob/v2.2.1/docs/storagedrivers.md) описывает некоторые варианты, такие как хранение данных в Amazon S3.

Вам может быть интересно, что такое опция `--insecure-registry`. Чтобы пользователи могли оставаться в безопасности, Docker позволит вам получать данные из реестров только с подписанным сертификатом HTTPS. Мы переопределили это, потому что нам вполне комфортно, оттого что можем доверять нашей локальной сети. Само собой разумеется, вам следует быть гораздо более осторожными, когда вы делаете это через интернет.

ОБСУЖДЕНИЕ

Поскольку реестры очень просты в настройке, возникает ряд возможностей. Если в вашей компании несколько команд, можете предложить всем запустить и поддерживать реестр на запасном компьютере, чтобы обеспечить некоторую плавность хранения образов и их перемещения.

Это особенно хорошо работает, если у вас есть внутренний диапазон IP-адресов, – команда `--insecure-registry` примет нотацию CIDR, например `10.1.0.0/16`, для указания диапазона IP-адресов, которые могут быть небезопасными. Если вы не знакомы с этим, мы настоятельно рекомендуем связаться со своим сетевым администратором.

2.5. Docker Hub

Docker Hub (см. рис. 2.10) – это реестр, поддерживаемый Docker Inc. Он содержит десятки тысяч образов, готовых к загрузке и запуску. Любой пользователь Docker может создать бесплатную учетную запись и хранить там общедоступные образы Docker. В дополнение к предоставленным пользователями официальные образы поддерживаются для справочных целей.

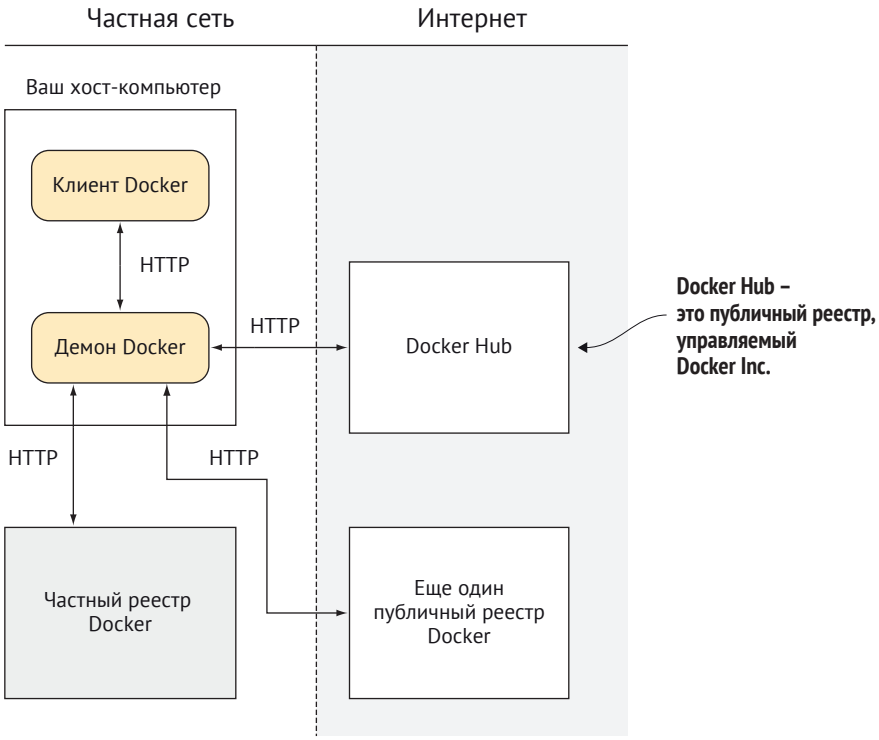


Рис. 2.10 ❖ Docker Hub

Ваши образы защищены аутентификацией пользователя, и существует звездная система популярности, похожая на ту, что есть в GitHub. Официальные образы могут быть представлениями дистрибутивов Linux, таких как Ubuntu или CentOS, предустановленными пакетами программного обеспечения, такими как Node.js, или целыми программными стеками, такими как WordPress.

МЕТОД 10 Поиск и запуск образа Docker

Реестры Docker позволяют использовать культуру социального кодирования, аналогичную GitHub. Если вас интересует испытание нового программного приложения или вы ищете новое приложение, которое предназначено для определенной цели, образы Docker могут быть простым способом поэкспериментировать без вмешательства в работу вашего хост-компьютера, подготовки виртуальной машины или необходимости волноваться об этапах установки.

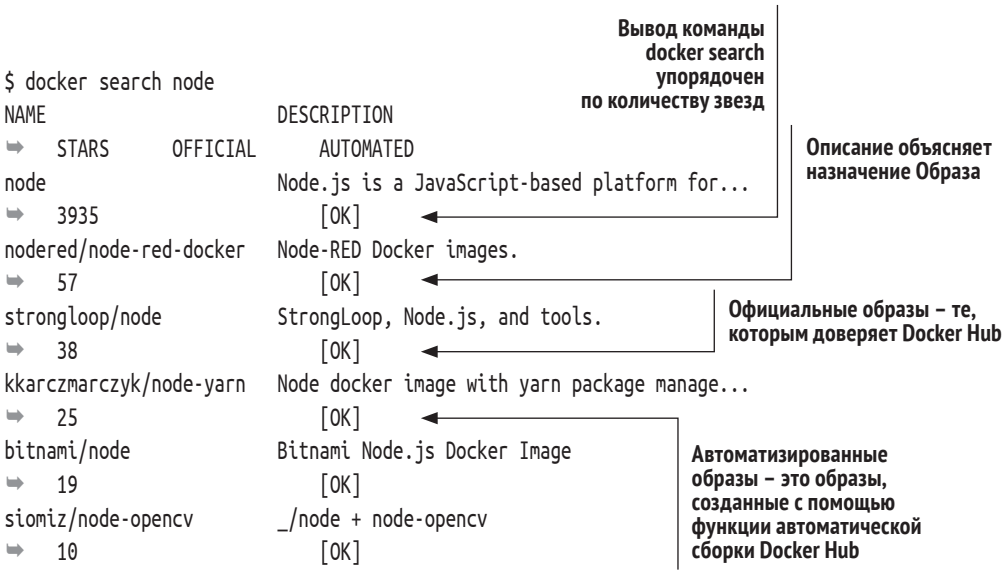
ПРОБЛЕМА

Вы хотите найти приложение или инструмент в качестве образа Docker и опробовать его.

РЕШЕНИЕ

Используйте команду поиска `docker search`, чтобы найти образ, который нужно извлечь, а затем запустите его.

Допустим, вы хотите поэкспериментировать с Node.js. В следующем примере мы искали образы, соответствующие «node» с помощью команды `docker search`:




```
$ docker search node
NAME                      DESCRIPTION
➥   STARS      OFFICIAL     AUTOMATED
node                      Node.js is a JavaScript-based platform for...
➥   3935                     [OK]
nodered/node-red-docker   Node-RED Docker images.
➥   57                       [OK]
strongloop/node           StrongLoop, Node.js, and tools.
➥   38                       [OK]
kkarczmarczyk/node-yarn   Node docker image with yarn package manage...
➥   25                       [OK]
bitnami/node              Bitnami Node.js Docker Image
➥   19                       [OK]
siomiz/node-opencv        _/node + node-opencv
➥   10                       [OK]
dahlb/alpine-node         small node for gitlab ci runner
➥   8                        [OK]
cusspvz/node              Super small Node.js container (~15MB) ba...
➥   7                        [OK]
anigeo/node-forever       Daily build node.js with forever
➥   4                        [OK]
seegno/node               A node docker base image.
➥   3                        [OK]
starefossen/ruby-node     Docker Image with Ruby and Node.js installed
➥   3                        [OK]
urbanmassage/node         Some handy (read, better) docker node images
➥   1                        [OK]
xataz/node                very light node image
➥   1                        [OK]
centralping/node          Bare bones CentOS 7 NodeJS container.
➥   1                        [OK]
joxit/node                Slim node docker with some utils for dev
➥   1                        [OK]
bigtruedata/node          Docker image providing Node.js & NPM
➥   1                        [OK]
```

```
1science/node             Node.js Docker images based on Alpine Linux
➥   1                        [OK]
domandtom/node            Docker image for Node.js including Yarn an...
➥   0                        [OK]
makeomatic/node           various alpine + node based containers
➥   0                        [OK]
c4tech/node               NodeJS images, aimed at generated single-p...
➥   0                        [OK]
instructure/node          Instructure node images
➥   0                        [OK]
octoblu/node              Docker images for node
➥   0                        [OK]
edvisor/node              Automated build of Node.js with commonly u...
➥   0                        [OK]
watsco/node               node:7
➥   0                        [OK]
codexsystems/node         Node.js for Development and Production
➥   0                        [OK]
```

Как только вы выбрали образ, можете загрузить его, выполнив команду `docker pull`:

```
$ docker pull node
 Using default tag: latest
latest: Pulling from library/node
5040bd298390: Already exists
fce5728aad85: Pull complete
76610ec20bf5: Pull complete
9c1bc3c30371: Pull complete
33d67d70af20: Pull complete
da053401c2b1: Pull complete
05b24114aa8d: Pull complete
Digest:
➥ sha256:ea65cf88ed7d97f0b43bcc5deed67cfd13c70e20a66f8b2b4fd4b7955de92297
Status: Downloaded newer image for node:latest
```

Извлекает образ с именем node из Docker Hub

Это сообщение отображается, если Docker извлек новый образ (в отличие от установленного факта, согласно которому не существует более нового образа, чем тот, который у вас уже есть). Вывод может быть другим

Затем вы можете запустить его в интерактивном режиме, используя флаги `-t` и `-i`. Флаг `-t` создает для вас TTY- устройство (терминал), а флаг `-i` указывает, что этот сеанс Docker является интерактивным:

```
$ docker run -t -i node /bin/bash
root@c267ae999646:/# node
> process.version
'v7.6.0'
>
```

> **ПОДСКАЗКА.** Вы можете сохранить нажатия клавиш, заменив `-t -i` на `-ti` или `-it` в предыдущем вызове команды `docker run`. Вы будете встречать это на протяжении всей книги.

Часто авторы образов будут давать конкретные советы по поводу того, как следует запускать образ. В результате поиска образа на сайте http://hub.docker.com вы перейдете на страницу образа. Вкладка «Описание» может дать вам больше информации.

> **ВНИМАНИЕ!** Если вы загружаете образ и запускаете его, вы запускаете код, который не сможете полностью проверить. Хотя использование доверенных образов относительно безопасно, ничто не может гарантировать 100 % безопасность при загрузке и запуске программного обеспечения через интернет.

Вооружившись этими знаниями и опытом, теперь вы можете использовать огромные ресурсы, доступные в Docker Hub. Здесь буквально десятки тысяч образов, которые можно опробовать, и есть чему поучиться. Наслаждайтесь!

ОБСУЖДЕНИЕ

Docker Hub – отличный ресурс, но иногда он может быть медленным – стоит остановиться, чтобы решить, как лучше создать команду поиска Docker для получения наилучших результатов. Возможность выполнять поиск, не открывая браузер, дает быстрое понимание возможных элементов, представляющих интерес в экосистеме, поэтому вы можете лучше ориентироваться в документации к образам, которые способны удовлетворить ваши потребности.

При выполнении повторной сборки образов также может быть полезно время от времени запускать поиск, чтобы убедиться, что подсчет звезд показывает, что сообщество Docker начало собираться вокруг образа, отличного от того, которой вы используете в настоящее время.

Резюме

- Вы можете открыть API демона Docker для посторонних, и все, что им нужно, – это способ сделать HTTP-запрос; достаточно веб-браузера.
- Контейнеры не должны захватывать ваш терминал. Можно запустить их в фоновом режиме и вернуться к ним позже.
- Вы можете установить связь между контейнерами либо с пользовательскими сетями (рекомендуемый подход), либо с помощью соединений, чтобы очень явно контролировать взаимодействие между контейнерами.
- Поскольку API демона Docker работает по протоколу HTTP, его легко отладить с помощью инструментов мониторинга сети, если у вас возникли проблемы.

- Одним из особенно полезных инструментов для отладки и отслеживания сетевых вызовов является socat.
- Настройка реестров – это поле деятельности не только Docker Inc. Вы можете настроить свой собственный реестр в локальной сети для бесплатного частного хранения образов.
- Docker Hub – это отличное место, где можно найти и загрузить готовые образы, в частности, те, что официально предоставлены Docker Inc.

Часть 2

Docker и разработка

В первой части вы изучили основные концепции и архитектуру Docker на примере. Во второй части вы узнаете, как использовать Docker в разработке.

Глава 3 посвящена использованию Docker в качестве легкой виртуальной машины. Это спорная область. Хотя между виртуальными машинами и контейнерами Docker существуют критические различия, во многих случаях с помощью Docker разработка может быть значительно ускорена. Это также эффективное средство знакомства с Docker, прежде чем перейти к более продвинутому его использованию.

Главы 4, 5 и 6 охватывают более 20 методов, позволяющих сделать сборку, запуск и управление контейнерами Docker более эффективными и действенными. Помимо создания и запуска контейнеров вы узнаете о сохранении данных с помощью томов и о том, как поддерживать порядок в хосте Docker.

Глава 7 охватывает важную область управления конфигурацией. Вы будете использовать файлы Dockerfile и традиционные инструменты управления конфигурацией, чтобы получить контроль над своими сборками Docker. Мы также расскажем о создании и обработке минимальных образов Docker для уменьшения размазывания образа. К концу этой части у вас будет множество полезных приемов для одноразового использования Docker, и вы будете готовы принять Docker в контекст DevOps.

Глава 3

Использование Docker в качестве легкой виртуальной машины

О чем рассказывается в этой главе:

- преобразование виртуальной машины в образ Docker;
- управление запуском служб вашего контейнера;
- сохранение работы по мере продвижения;
- управление образами Docker на вашем компьютере;
- совместное использование образов в Docker Hub;
- играйте в 2048 и выигрывайте вместе с Docker.

Виртуальные машины (VM) стали повсеместными в разработке и развертывании программного обеспечения с начала века. Абстракция машин к программному обеспечению сделала перемещение и управление программным обеспечением и службами в эпоху интернета проще и дешевле.

> **ПОДСКАЗКА.** Виртуальная машина – это приложение, которое эмулирует компьютер обычно для того, чтобы запустить операционную систему и приложения. Его можно разместить на любых (совместимых) доступных физических ресурсах. Конечный пользователь воспринимает программное обеспечение, как если бы оно было на физической машине, но те, кто управляет оборудованием, могут сосредоточиться на более широком распределении ресурсов.

Docker не является VM-технологией. Он не моделирует аппаратное обеспечение компьютера и не включает в себя операционную систему. Контейнер Docker по умолчанию не ограничен конкретными аппаратными ограничениями. Если Docker виртуализирует что-либо, он виртуализирует среду, в которой

запускаются службы, а не компьютер. Более того, Docker не может с легкостью запускать программное обеспечение Windows (или даже ПО, написанное для других операционных систем Unix).

Однако с некоторых точек зрения Docker можно использовать как виртуальную машину. Для разработчиков и тестировщиков в эпоху интернета тот факт отсутствия процесса инициализации или прямого взаимодействия с оборудованием обычно не имеет большого значения. И есть существенные общие черты, такие как его изоляция от окружающего оборудования и доступность для более тонких подходов к доставке программного обеспечения.

Эта глава проведет вас через сценарии, в которых вы могли бы применять Docker, как ранее могли использовать виртуальную машину. Применение Docker не даст вам никаких очевидных функциональных преимуществ по сравнению с виртуальной машиной, но скорость и удобство, которые Docker обеспечивает для перемещения и отслеживания сред, могут изменить правила игры для вашего конвейера развертывания.

3.1. От виртуальной машины к контейнеру

В идеальном мире переход от виртуальных машин к контейнерам был бы простым делом запуска сценариев управления конфигурацией для образа Docker из дистрибутива, похожего на дистрибутив виртуальной машины. Для тех из нас, у кого положение дел не такое радужное, этот раздел покажет, как можно преобразовать виртуальную машину в контейнер или контейнеры.

МЕТОД 11 Преобразование вашей виртуальной машины в контейнер

В Docker Hub нет всех возможных базовых образов, поэтому для некоторых нишевых дистрибутивов Linux и вариантов использования нужно создавать свои собственные. Например, если у вас есть состояние существующего приложения на виртуальной машине, вы можете захотеть поместить это состояние в образ Docker, чтобы иметь возможность выполнять дальнейшие итерации с ним или извлечь выгоду из экосистемы Docker, используя инструменты и связанные с этим технологии, которые там существуют.

В идеале вы хотите создать эквивалент своей виртуальной машины с нуля, используя стандартные методы Docker, такие как файлы Dockerfile в сочетании со стандартными инструментами управления конфигурацией (см. главу 7). Реальность, однако, заключается в том, что многие виртуальные машины не подвергаются тщательному управлению конфигурацией. Это может произойти из-за того, что виртуальная машина органично росла, по мере того как ее использовали люди, и необходимые инвестиции для воссоздания ее более структурированным способом не стоят того.

ПРОБЛЕМА

У вас есть виртуальная машина, которую вы хотите преобразовать в образ Docker.

РЕШЕНИЕ

Архивируйте и скопируйте файловую систему виртуальной машины и упакуйте ее в образ Docker.

Сначала мы разделим виртуальные машины на две большие группы:

- *локальная виртуальная машина* – образ диска виртуальной машины продолжает работать, а выполнение виртуальной машины происходит на вашем компьютере;
- *удаленная виртуальная машина* – хранение образа диска виртуальной машины и ее выполнение происходят где-то еще.

Принцип обеих групп виртуальных машин (и всего, что вы хотите создать из образа Docker) одинаков – вы получаете TAR-файл файловой системы и добавляете (`ADD`) его в/из образа `scratch`.

> **ПОДСКАЗКА.** Команда `ADD` (в отличие от команды одноуровневого копирования `COPY`) распаковывает TAR-файлы (а также файлы в формате gzip и другие подобные типы файлов), когда они размещены в образе, подобном этому.

> **ПОДСКАЗКА.** Образ `scratch` – это псевдообраз с нулевым байтом, который вы можете собрать поверх. Обычно он применяется в случаях, когда вы хотите скопировать (или добавить) полную файловую систему, используя файл Dockerfile.

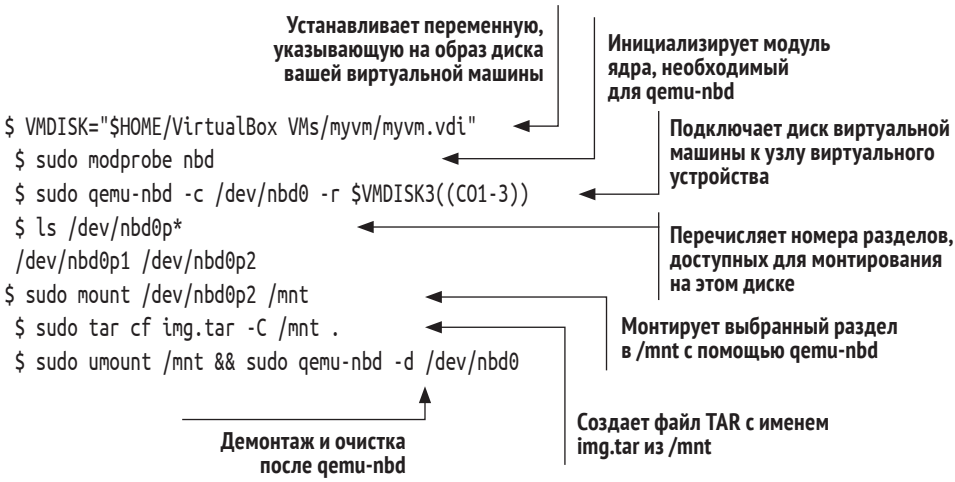
Сейчас мы рассмотрим случай, когда у вас есть локальная виртуальная машина VirtualBox.

Прежде чем начать, вам необходимо сделать следующее:

1. Установите утилиту `qemu-nbd` (доступна как часть пакета `qemu-utils` в Ubuntu).
2. Определите путь к образу диска вашей виртуальной машины.
3. Завершите работу своей виртуальной машины.

Если образ диска вашей виртуальной машины имеет формат .vdi или .vmdk, этот метод должен сработать. Другие форматы могут иметь смешанный успех. Приведенный ниже код демонстрирует, как можно превратить файл вашей виртуальной машины в виртуальный диск, что позволяет копировать с него все файлы.

Листинг 3.1. Извлечение файловой системы образа виртуальной машины



ПРИМЕЧАНИЕ. Для выбора раздела для монтирования выполните команду `sudo cfdisk/dev/nbd0`, чтобы увидеть, что доступно. Обратите внимание, что, если вы где-либо видите LVM, то у вашего диска нетривиальная схема разметки – вам нужно будет дополнительно выяснить, как монтировать разделы LVM.

Если виртуальная машина хранится удаленно, у вас есть выбор: либо выключить ее и попросить операционную группу выполнить дамп нужного вам раздела, либо создать TAR-файл своей виртуальной машины, пока она еще работает.

Если вы получили дамп раздела, можно легко его смонтировать, а затем превратить в TAR-файл:

Листинг 3.2. Извлечение раздела

```
$ sudo mount -o loop partition.dump /mnt
$ sudo tar cf $(pwd)/img.tar -C /mnt .
$ sudo umount /mnt
```

Кроме того, вы можете создать TAR-файл из работающей системы. Это довольно просто после входа в систему:

Листинг 3.3. Извлечение файловой системы работающей виртуальной машины

```
$ cd /
$ sudo tar cf /img.tar --exclude=/img.tar --one-file-system /
```

Теперь у вас есть TAR-файл образа файловой системы, который вы можете перенести на другой компьютер с помощью `scp`.

> **ПРЕДУПРЕЖДЕНИЕ.** Создание TAR-файла из работающей системы может показаться самым простым вариантом (без выключений, установки программного обеспечения или отправки запросов другим командам), но у него есть серьезный недостаток – вы можете скопировать файл в несогласованном состоянии и столкнуться со странными проблемами при попытке использовать свой новый образ Docker. Если вы должны идти по этому пути, сначала остановите как можно больше приложений и служб.

Как только вы получите TAR-файл своей файловой системы, можете добавить его к своему образу. Это самый простой шаг процесса, который состоит из двухстрочного файла Dockerfile.

Листинг 3.4. Добавление архива в образ Docker

```
FROM scratch
ADD img.tar /
```

Теперь вы можете выполнить команду `docker build .` – и ваш образ готов!

> **ПРИМЕЧАНИЕ.** Docker предоставляет альтернативу команде `ADD` в виде команды `docker import`, которую вы можете использовать в `cat img.tar | docker import - new_image_name`. Но сборка поверх образа с дополнительными инструкциями потребует от вас создания файла Dockerfile в любом случае, поэтому может быть проще пойти по пути использования `ADD`, чтобы иметь возможность без труда увидеть историю своего образа.

Теперь у вас есть образ в Docker, и можно приступить к экспериментам с ним. В этом случае вы можете начать с создания нового файла Dockerfile на основе своего нового образа, чтобы поэкспериментировать с удалением файлов и пакетов.

После того как вы это сделали и довольны результатами, можете использовать Docker Export для работающего контейнера, чтобы экспортировать

новый, более тонкий TAR-файл, который можно взять в качестве основы для нового образа, и повторять этот процесс, пока не получите образ, которым вы останетесь довольны.

Блок-схема на рис. 3.1 демонстрирует этот процесс.

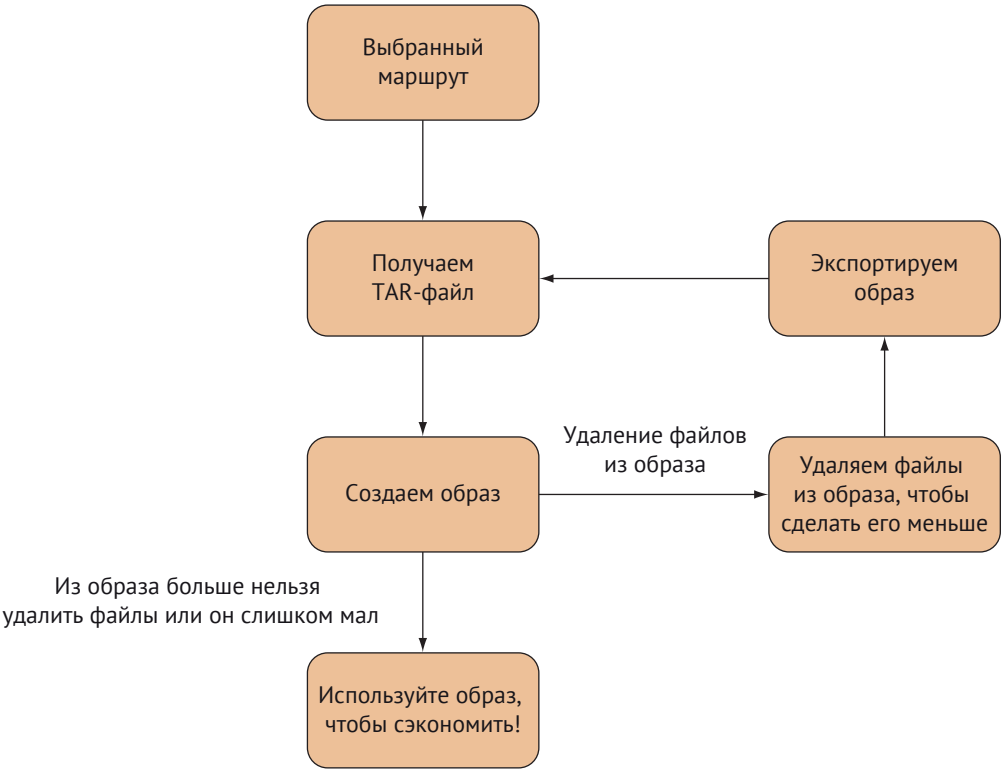


Рис. 3.1 ❖ Схема удаления файлов из образа с целью уменьшения его размера

ОБСУЖДЕНИЕ

Этот способ демонстрирует несколько фундаментальных принципов и методов, которые полезны в других контекстах, кроме преобразования виртуальной машины в образ Docker.

В более широком смысле он показывает, что образ Docker – это, по сути, набор файлов и метаданные: образ `scratch` – это пустая файловая система, поверх которой может быть наложен TAR-файл. Мы вернемся к этой теме, когда будем рассматривать тонкие образы Docker.

В частности, вы увидели, как можно добавить TAR-файл в образ Docker и использовать утилиту `qemu-nbd`.

Когда у вас будет образ, вам, возможно, потребуется знать, как запустить его в качестве более традиционного хоста. Поскольку контейнеры Docker обычно запускают только один процесс приложения, это несколько противоречит принципу и рассматривается в следующем методе.

МЕТОД 12 Хост-подобный контейнер

Теперь мы переходим к одной из наиболее спорных областей обсуждения в сообществе Docker – запуск хост-подобного образа с использованием нескольких процессов, запущенных с самого начала.

Этот способ считается плохим среди некоторых членов сообщества Docker. Контейнеры не являются виртуальными машинами – есть значительные различия – и, если мы будем делать вид, что это не так, это может вызвать путаницу и привести к проблемам в будущем.

Хорошо это или плохо, данный метод покажет вам, как запустить хост-подобный образ, и мы обсудим проблемы, связанные с этим.

> **ПРИМЕЧАНИЕ.** Запуск хост-подобного образа может быть хорошим способом убедить отказавшихся использовать Docker в том, что он полезен. Чем больше они будут его использовать, тем лучше поймут эту парадигму, и подход к микросервисам будет иметь для них больше смысла. В компании, где внедрили Docker, мы обнаружили, что этот монолитный подход является отличным способом перевести людей с разработки на серверах и ноутбуках в более замкнутую и управляемую среду. Оттуда перевод Docker в тестирование, непрерывную интеграцию, условное депонирование и рабочие процессы DevOps был тривиален.

Различия между виртуальными машинами и контейнерами Docker

Вот некоторые различия между виртуальными машинами и контейнерами Docker:

- Docker ориентирован на приложения, в то время как виртуальные машины - на операционные системы;
- контейнеры Docker совместно используют операционную систему вместе с другими контейнерами Docker. Напротив, у каждой виртуальной машины есть собственная операционная система, управляемая гипервизором;
- контейнеры Docker предназначены для запуска одного основного процесса, а не для управления несколькими наборами процессов.

ПРОБЛЕМА

Вам нужна нормальная хост-среда для своего контейнера с несколькими настроенными процессами и службами.

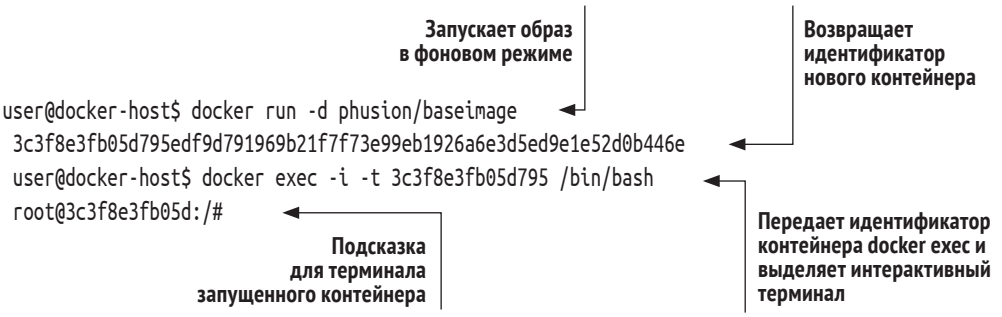
РЕШЕНИЕ

Используйте базовый контейнер, предназначенный для запуска нескольких процессов.

В этом методе вы будете использовать образ, предназначенный для имитации хоста, снабдив его необходимыми приложениями. Базовым будет образ `phusion/baseimage`, предназначенный для запуска нескольких процессов.

Первые шаги – запустить образ и перейти к нему с помощью команды `docker exec`.

Листинг 3.5. Запуск базового образа phusion

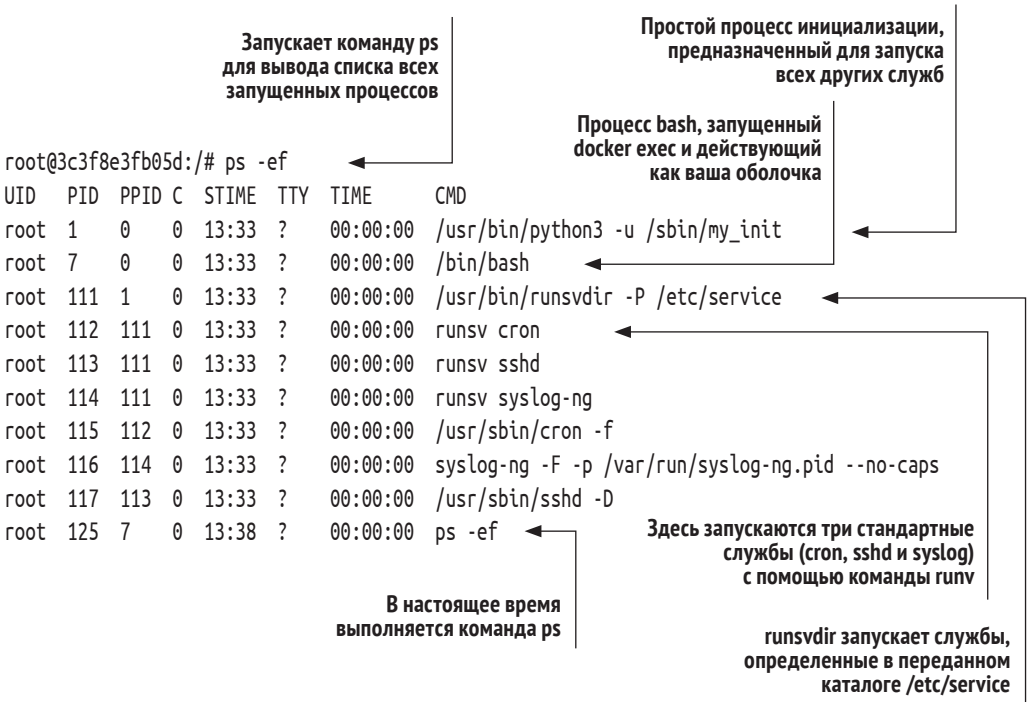


В этом коде команда `docker run` запустит образ в фоновом режиме, выполнив команду по умолчанию для образа и вернув идентификатор вновь созданного контейнера.

Затем вы передаете этот идентификатор контейнера команде `docker exec`, которая запускает новый процесс внутри уже запущенного контейнера. Флаг `-i` позволяет вам взаимодействовать с новым процессом, а флаг `-t` указывает, что вы хотите установить TTY, чтобы дать себе возможность запустить терминал (`/bin/bash`) внутри контейнера.

Если вы подождете минуту, а затем посмотрите на таблицу процессов, ваш вывод будет выглядеть примерно так:

Листинг 3.6. Процессы, идущие в хост-подобном контейнере



Видно, что контейнер запускается во многом как хост, инициализируя такие службы, как cron и sshd, которые делают его похожим на стандартный хост Linux.

ОБСУЖДЕНИЕ

Хотя это может быть полезно в начальных демонстрациях для инженеров, не знакомых с Docker, или действительно полезно в ваших конкретных обстоятельствах, стоит помнить, что это несколько спорная идея.

История использования контейнеров имеет тенденцию применять их для изоляции рабочих нагрузок от «одного сервиса на контейнер». Сторонники подхода хост-подобного образа, утверждают, что это не нарушает данный принцип, поскольку контейнер все еще может выполнять одну дискретную функцию для системы, в которой он работает.

В последнее время растущая популярность концепций модуля Kubernetes и docker-compose сделала хост-подобный контейнер относительно избыточным – отдельные контейнеры могут быть объединены в один объект на более высоком уровне, вместо того чтобы управлять несколькими процессами, используя традиционную службу init.

Следующий метод рассматривает, как можно разбить такое монолитное приложение на контейнеры в стиле микросервисов.

МЕТОД 13 Разделение системы на микросервисные контейнеры

Мы изучили, как использовать контейнер в качестве монолитного объекта (например, классического сервера), и объяснили, что это может быть отличным способом быстрого переноса архитектуры системы на Docker. Однако в мире Docker, как правило, рекомендуется разделять систему как можно больше, пока у вас не будет запущена одна служба для каждого контейнера, а все контейнеры не будут связаны сетями.

Основной причиной использования одного сервиса на контейнер является более легкое разделение интересов по принципу единственной ответственности. Если у вас есть контейнер, выполняющий одну работу, легче провести его через жизненный цикл разработки программного обеспечения, включая разработку, тестирование и производство, не беспокоясь о его взаимодействии с другими компонентами. Это обеспечивает более гибкие поставки и более масштабируемые программные проекты, однако и создает дополнительные расходы на управление, поэтому неплохо подумать, стоит ли оно того при вашем варианте использования.

Если оставить в стороне обсуждение того, какой подход лучше для вас сейчас, у подхода best-practice, как вы увидите, есть одно явное преимущество – экспериментирование и повторная сборка намного быстрее при использовании файлов Dockerfile.

ПРОБЛЕМА

Вы хотите разбить свое приложение на отдельные и более управляемые сервисы.

РЕШЕНИЕ

Создайте контейнер для каждого отдельного служебного процесса.

Как мы уже упоминали, в сообществе Docker идут споры о том, как строго следует соблюдать правило «один сервис на контейнер», причем часть этого вытекает из разногласий по поводу определений – это отдельный процесс или коллекция процессов, которые объединяются для удовлетворения потребностей? Часто это сводится к утверждению, что, учитывая возможность перепроектирования системы с нуля, микросервисы – это путь, который выберет большинство. Но иногда практичность побеждает идеализм – оценивая Docker для нашей организации, мы оказались в положении необходимости пойти по монолитному маршруту, для того чтобы Docker работал максимально быстро и легко.

Давайте рассмотрим один из конкретных недостатков использования монолитов внутри Docker. Во-первых, в следующем листинге показано, как создать монолит с базой данных, приложением и веб-сервером.

> **ПРИМЕЧАНИЕ.** Эти примеры приведены для пояснения и были соответственно упрощены. Попытка запустить их напрямую не обязательно сработает.

Листинг 3.7. Настройка простого приложения с PostgreSQL, NodeJS и Nginx

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql nodejs npm nginx
WORKDIR /opt
COPY . /opt/ # {*}
RUN service postgresql start && \
    cat db/schema.sql | psql && \
    service postgresql stop
RUN cd app && npm install
RUN cp conf/mysite /etc/nginx/sites-available/ && \
    cd /etc/nginx/sites-enabled && \
    ln -s ../sites-available/mysite
```

Каждая команда файла Dockerfile создает один новый слой поверх предыдущего, но использование `&&` в операторах `RUN` эффективно гарантирует, что несколько команд будут запущены как одна. Это полезно, потому что может сохранять ваши образы маленькими. Если вы запустите команду обновления пакета, такую как `apt-get update`, с помощью команды `install`, таким образом, вы убедитесь, что при установке пакетов они будут из обновленного кеша пакетов.

Предыдущий пример представляет собой концептуально простой файл Dockerfile, который устанавливает все необходимое внутри контейнера, а затем настраивает базу данных, приложение и веб-сервер. К сожалению, если вы хотите быстро выполнить повторную сборку своего контейнера, возникает проблема – при любом изменении любого файла в вашем репозитории будет выполнена повторная сборка всего, начиная с {*}, потому что кеш не может быть использован снова. Если у вас есть несколько медленных шагов (создание базы данных или `npm install`), можете какое-то время подождать, пока контейнер будет собран снова.

Решение этой проблемы состоит в том, чтобы разделить инструкцию `COPY . /opt/` на отдельные аспекты приложения (база данных, приложение и веб-настройка).

Листинг 3.8. Файл Dockerfile для монолитного приложения

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql nodejs npm nginx
WORKDIR /opt
COPY db /opt/db                                    -+
RUN service postgresql start && \                   |- db setup
    cat db/schema.sql | psql && \                   |
    service postgresql stop                        -+
COPY app /opt/app                                  -+
```

```
RUN cd app && npm install                                  |- app setup
RUN cd app && ./minify_static.sh                          -+
COPY conf /opt/conf                                       -+
RUN cp conf/mysite /etc/nginx/sites-available/ && \        +
    cd /etc/nginx/sites-enabled && \                       |- web setup
    ln -s ../sites-available/mysite                       -+
```

В предыдущем коде команда COPY разделена на две отдельные инструкции. Это означает, что база данных не будет перестраиваться при каждом изменении кода, поскольку кеш можно повторно использовать для неизмененных файлов, доставленных до кода. К сожалению, поскольку функции кеширования довольно просты, контейнер все равно необходимо полностью собирать снова каждый раз, когда вносятся изменения в сценарии схемы. Единственный способ решить эту проблему – отойти от последовательных шагов настройки и создать несколько файлов Dockerfile, как показано в листингах с 3.9 по 3.11.

Листинг 3.9. Файл Dockerfile для службы postgres

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install postgresql
WORKDIR /opt
COPY db /opt/db
RUN service postgresql start && \
    cat db/schema.sql | psql && \
    service postgresql stop
```

Листинг 3.10. Файл Dockerfile для службы nodejs

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nodejs npm
WORKDIR /opt
COPY app /opt/app
RUN cd app && npm install
RUN cd app && ./minify_static.sh
```

Листинг 3.11. Файл Dockerfile для службы nginx

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nginx
WORKDIR /opt
COPY conf /opt/conf
RUN cp conf/mysite /etc/nginx/sites-available/ && \
    cd /etc/nginx/sites-enabled && \
    ln -s ../sites-available/mysite
```

Всякий раз, когда изменяется одна из папок `db`, `app` или `conf`, нужно выполнять повторную сборку только одного контейнера. Это особенно полезно, когда у вас более трех контейнеров или требуются длительные этапы установки. Вы можете осторожно добавить минимум файлов, необходимых для каждого шага, в результате получите более полезное кеширование файла Dockerfile.

В приложении Dockerfile (листинг 3.10) операция `npm install` определяется одним файлом `package.json`, поэтому вы можете изменить свой файл Dockerfile, чтобы использовать преимущества кеширования слоя Dockerfile и повторно создавать медленный шаг `npm install` только при необходимости:

Листинг 3.12. Более быстрый файл Dockerfile для службы nginx

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install nodejs npm
WORKDIR /opt
COPY app/package.json /opt/app/package.json
RUN cd app && npm install
COPY app /opt/app
RUN cd app && ./minify_static.sh
```

Теперь у вас есть три отдельных файла Dockerfile там, где раньше был один.

ОБСУЖДЕНИЕ

К сожалению, такого понятия, как бесплатный обед, не существует – вы обменяли один простой Dockerfile на несколько файлов Dockerfile с помощью дублирования. Можно частично решить эту проблему, добавив еще один файл Dockerfile, который будет использоваться в качестве базового образа, но дублирование не является редкостью. Кроме того, теперь существует некоторая сложность в запуске вашего образа – в дополнение к шагам с использованием `EXPOSE`, которые делают соответствующие порты доступными для соединения и изменения конфигурации Postgres, вам обязательно нужно соединять контейнеры при каждом запуске. К счастью, для этого есть инструмент под названием *Docker Compose*, который мы рассмотрим в методе 76.

Пока в этом разделе вы взяли виртуальную машину, превратили ее в образ Docker, запустили контейнер, похожий на хост, и разбили монолит на отдельные образы Docker.

Если после прочтения этой книги вы по-прежнему хотите запускать несколько процессов внутри контейнера, существуют специальные инструменты, которые могут помочь в этом. Один из них – Supervisord – рассматривается в следующем методе.

МЕТОД 14 Управление запуском служб вашего контейнера

Как следует из книг, посвященных Docker, контейнер Docker *не является* виртуальной машиной.

Одно из ключевых отличий между контейнером Docker и виртуальной машиной заключается в том, что контейнер предназначен для запуска одного процесса. Когда этот процесс заканчивается, контейнер завершает работу. Это отличается от виртуальной машины Linux (или любой операционной системы Linux) тем, что в ней нет процесса инициализации.

Процесс инициализации выполняется в ОС Linux с помощью идентификатора процесса 1 и идентификатора родительского процесса 0. Этот процесс может называться «init» или «systemd». Как бы он ни назывался, его задача состоит в управлении служебной деятельностью для всех остальных процессов, запущенных в этой операционной системе.

Если вы начнете экспериментировать с Docker, то сможете запустить несколько процессов. Например, запустить задания cron, чтобы привести в порядок файлы журналов своих локальных приложений или настроить внутренний сервер Memcached в контейнере. Если вы выберете этот путь, можете написать сценарии оболочки для управления запуском этих подпроцессов. По сути, вы будете эмулировать работу процесса init. Не делайте этого! Многие проблемы, возникающие из-за управления процессами, встречались ранее и были решены в предварительно упакованных системах.

Какой бы ни была ваша причина для запуска нескольких процессов внутри контейнера, важно не изобретать велосипед заново.

ПРОБЛЕМА

Вы хотите управлять несколькими процессами внутри контейнера.

РЕШЕНИЕ

Используйте Supervisor для управления процессами в вашем контейнере.

Мы покажем, как подготовить контейнер для запуска на Tomcat и Apache, и как он будет запускаться и работать управляемым образом, а приложение Supervisor (http://supervisord.org/) будет управлять запуском процесса за вас.

Сначала создайте свой файл Dockerfile в новом и пустом каталоге, как показано в приведенном ниже листинге.

Листинг 3.13. Пример файла Dockerfile для Supervisor

```
FROM ubuntu:14.04
 ENV DEBIAN_FRONTEND noninteractive
 RUN apt-get update && apt-get install -y python-pip apache2 tomcat7
 RUN pip install supervisor
 RUN mkdir -p /var/lock/apache2
 RUN mkdir -p /var/run/apache2
 RUN mkdir -p /var/log/tomcat
```

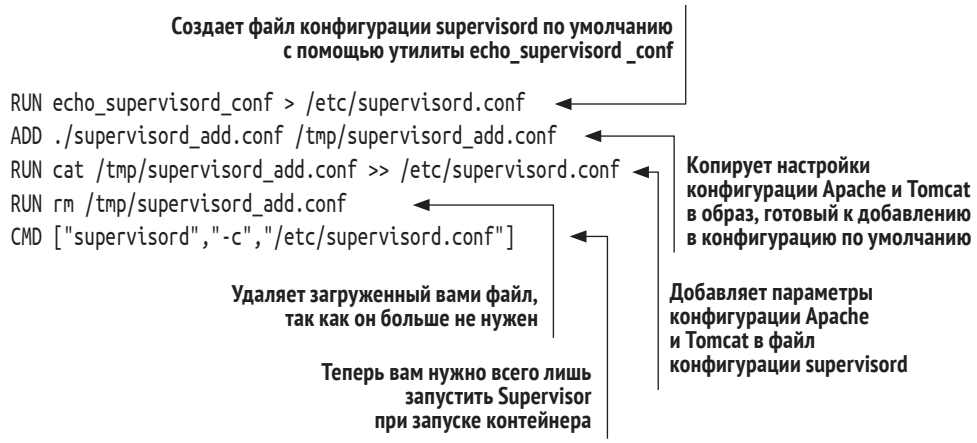
Начинается с Ubuntu: 14.04

Устанавливает переменную среды, чтобы указать, что этот сеанс неинтерактивный

Устанавливает python-pip (для установки Supervisor), apache2 и tomcat7

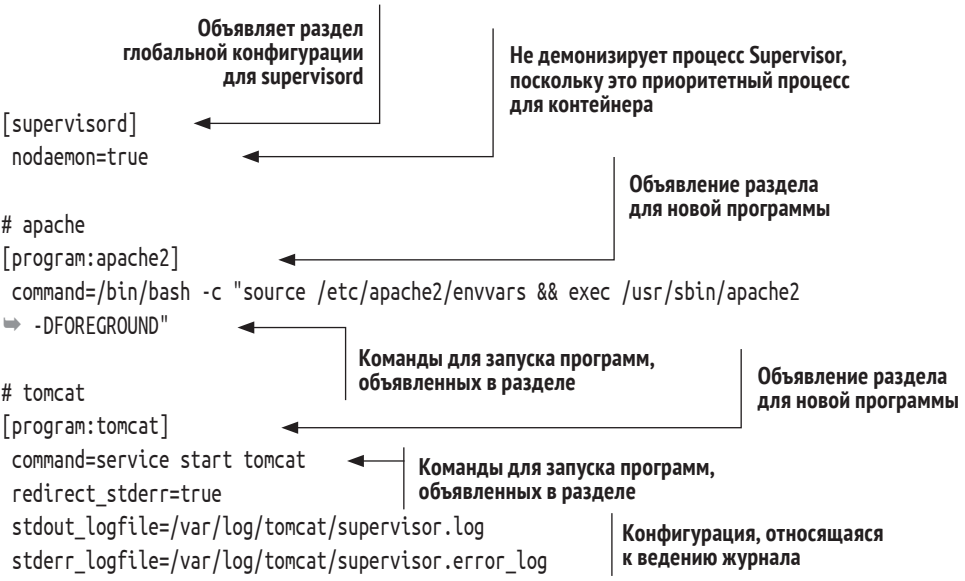
Устанавливает Supervisor с помощью pip

Создает вспомогательные каталоги, необходимые для запуска приложений



Вам также понадобится настроить Supervisor, чтобы указать, какие приложения необходимо запустить, как показано в следующем листинге.

Листинг 3.14. supervisord_add.conf

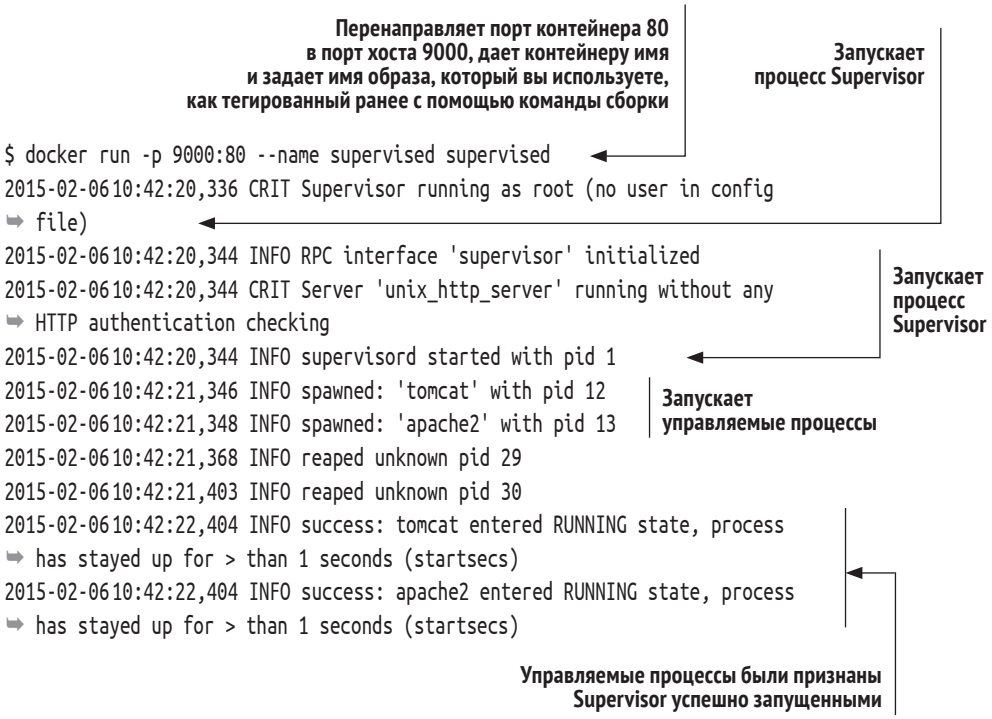


Вы собираете образ, применяя стандартный процесс Docker с одной командой, потому что используете файл Dockerfile. Запустите эту команду для выполнения сборки:

```
docker build -t supervised .
```

Теперь можете запустить свой образ!

Листинг 3.15. Запуск контейнера



Перейдя по адресу http://localhost:9000, вы должны увидеть страницу по умолчанию сервера Apache, который вы запустили.

Чтобы очистить контейнер, выполните следующую команду:

```
docker rm -f supervised
```

ОБСУЖДЕНИЕ

В этом методе использовался Supervisor для управления несколькими процессами в вашем контейнере Docker.

Если вас интересуют альтернативы Supervisor, есть также `runit`, который использовался базовым образом phusion, описанным в методе 12.

3.2. Сохранение и восстановление работы

Говорят, что код не написан до тех пор, пока не будут зафиксированы изменения в системе управления версиями – не всегда вредно иметь одинаковое отношение к контейнерам. Можно сохранить состояние с помощью виртуальных машин, используя снапшоты, но у Docker имеется гораздо более активный подход к поощрению сохранения и повторного использования существующей работы.

Мы расскажем о подходе к разработке под названием «сохранить игру», о тонкостях присвоения тегов, использовании Docker Hub и обращении к конкретным образам в ваших сборках. Поскольку эти операции считаются настолько фундаментальными, Docker делает их относительно простыми и быстрыми. Тем не менее эта тема все равно может вызывать вопросы у новичков, поэтому в следующем разделе мы пошагово рассмотрим ее для более полного понимания этого предмета.

МЕТОД 15 Подход «сохранить игру»: дешевое управление исходным кодом

Если вы когда-нибудь разрабатывали какое-либо программное обеспечение, вы, вероятно, хотя бы раз воскликнули: «Я уверен, что оно работало раньше!» Неспособность восстановить систему до известного хорошего (или, может быть, только «лучшего») состояния, когда вы второпях взламываете код, чтобы уложиться в срок, или исправляете ошибку, является причиной множества сломанных клавиатур.

Управление исходным кодом значительно помогло в этом, но в данном конкретном случае есть две проблемы:

- исходный код может не отражать состояние файловой системы вашей «рабочей» среды;
- возможно, вы еще не готовы зафиксировать изменения.

Первая проблема более значима. Хотя современные инструменты управления исходным кодом, такие как Git, могут легко создавать локальные разовые ветки, регистрация состояния всей вашей файловой системы разработки не является целью управления исходным кодом.

Docker предоставляет дешевый и быстрый способ хранения состояния файловой системы разработки вашего контейнера с помощью функции фиксации изменений, это и есть предмет изучения.

ПРОБЛЕМА

Вы хотите сохранить состояние вашей среды разработки.

РЕШЕНИЕ

Регулярно фиксируйте свой контейнер, чтобы можно было восстановить состояние в этой точке.

Давайте представим, что вы хотите внести изменения в свое приложение из главы 1.

Генеральный директор ToDoCorp недоволен и хочет, чтобы в заголовке браузера отображалось «ToDoCorp's ToDo App» вместо «Swarm + React – TodoMVC».

Вы не знаете, как этого добиться, поэтому, возможно, захотите запустить свое приложение и поэкспериментировать, изменяя файлы, чтобы посмотреть, что произойдет.

Листинг 3.16. Отладка приложения в терминале

```
$ docker run -d -p 8000:8000 --name todobug1 dockerinpractice/todoapp
 3c3d5d3ffd70d17e7e47e90801af7d12d6fc0b8b14a8b33131fc708423ee4372
$ docker exec -i -t todobug1 /bin/bash 2((CO7-2))
```

Команда `docker run` запускает приложение в контейнере в фоновом режиме (`-d`), перенаправляя порт контейнера 8000 в порт 8000 на хосте (`-p 8000: 8000`) и для удобства обозначая его как todobug1 (`--name todobug1`), возвращая идентификатор контейнера. Команда, запущенная в контейнере, будет командой по умолчанию, указанной при сборке образа `dockerinpractice/todoapp`. Мы собрали его за вас и сделали доступным в Docker Hub.

Вторая команда запустит `/bin/bash` в работающем контейнере. Используется имя `todobug1`, но вы также можете взять идентификатор контейнера. `-i` делает команду `docker exec` интерактивной, а `-t` гарантирует, что она будет работать так же, как терминал.

Теперь вы находитесь в контейнере, а значит, первым шагом в эксперименте является установка редактора.

Мы предпочитаем vim, поэтому использовали эти команды:

```
apt-get update
apt-get install vim
```

После небольшого усилия вы понимаете, что файл, который нужно преобразовать, – это local.html. Поэтому вы измените строку 5 в этом файле следующим образом:

```
<title>ToDoCorp's ToDo App</title>
```

Затем доходят слухи, что генеральный директор, возможно, хочет, чтобы название было написано строчными буквами, поскольку она слышала, что это выглядит более современным. Вы хотите быть готовым в любом случае, поэтому фиксируете те изменения, которые есть в данный момент. В другом терминале вы запускаете следующую команду.

Листинг 3.17. Фиксация состояния контейнера

Превращает созданный ранее контейнер в образ

```
$ docker commit todobug1
 ca76b45144f2cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970
```

Идентификатор нового образа контейнера, который вы зафиксировали

Теперь вы зафиксировали свой контейнер в образе, из которого сможете запустить его позже.

> **ПРИМЕЧАНИЕ.** При фиксации контейнера сохраняется только состояние файловой системы во время фиксации, а не процессы. Помните, что контейнеры Docker не являются виртуальными машинами!

Если состояние вашей среды зависит от состояния запущенных процессов, которые не восстанавливаются с помощью стандартных файлов, этот метод не будет сохранять состояние так, как вам нужно. В этом случае вы, вероятно, захотите познакомиться с тем, как сделать процессы разработки восстанавливаемыми.

Затем меняете local.html на другое возможное требуемое значение:

```
<title>todocorp's todo app</title>
```

Снова фиксируем изменения:

```
$ docker commit todobug1
071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036
```

Теперь есть два идентификатора образа, которые представляют два варианта (ca76b45144f2 cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970 и 071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036 в нашем примере, но у вас они будут другими). Когда генеральный директор придет, чтобы оценить и выбрать один из них, вы можете запустить любой образ и позволить ей решить, какой из них зафиксировать.

Это делается путем открытия новых терминалов и выполнения следующих команд.

Листинг 3.18. Запуск обоих зафиксированных образов в виде контейнеров

```
$ docker run -p 8001:8000 \
 ca76b45144f2cb31fda6a31e55f784c93df8c9d4c96bbeacd73cad9cd55d2970
 $ docker run -p 8002:8000 \
 071f6a36c23a19801285b82eafc99333c76f63ea0aa0b44902c6bae482a6e036
```

Перенаправляет порт контейнера 8000 в порт хоста 8001 и указывает идентификатор образа в нижнем регистре

Перенаправляет порт контейнера 8000 в порт хоста 8002 и указывает идентификатор образа в верхнем регистре

Таким образом, вы можете представить вариант с прописными буквами доступным по адресу http://localhost:8001, а вариант со строчными буквами по адресу http://localhost:8002.

ПРИМЕЧАНИЕ. Любые зависимости, внешние по отношению к контейнеру (такие как базы данных, тома Docker или другие вызываемые службы), не сохраняются при фиксации изменений. Этот метод не имеет никаких внешних зависимостей, поэтому вам не нужно беспокоиться об этом.

ОБСУЖДЕНИЕ

Этот метод продемонстрировал функциональность команды `docker commit` и то, как ее можно использовать в процессе разработки. Пользователи Docker склоняются к тому, чтобы применять эту команду только как часть формального рабочего процесса `commit-tag-push`, поэтому неплохо помнить, что у нее есть и другие применения.

Мы посчитали, что это полезный метод, когда согласовали сложную последовательность команд для настройки приложения. При успешной фиксации контейнера также записывается история вашего сеанса bash, а это означает, что есть набор шагов для восстановления состояния вашей системы. Вы можете сэкономить много времени! Это также полезно, когда вы экспериментируете с новой функцией и не уверены, закончили ли ее, или когда вы воссоздали ошибку и хотите быть как можно более уверенными в том, что можете вернуться в неисправное состояние.

Вам, наверное, интересно, есть ли лучший способ для обращения к образам, чем использование длинных случайных строк символов. Следующий метод будет заключаться в том, чтобы дать этим контейнерам имена, к которым вам будет проще обращаться.

МЕТОД 16 Присвоение тегов

Теперь вы сохранили состояние вашего контейнера, зафиксировав изменения, и у вас есть случайная строка в качестве идентификатора вашего образа. Очевидно, что запоминать и управлять большим количеством этих идентификаторов трудно. Было бы полезно использовать функцию тегирования Docker, чтобы дать вашим образам читаемые имена (и теги) и напомнить, для чего они были созданы.

Овладение этой техникой позволит сразу увидеть, для чего нужны ваши образы, что сделает управление ими на компьютере гораздо проще.

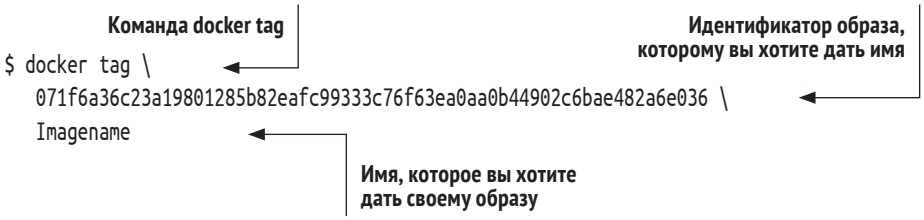
ПРОБЛЕМА

Вы хотите удобным образом обращаться к коммиту Docker и хранить его.

РЕШЕНИЕ

Используйте команду `docker tag` для именования своих коммитов.
В своей основной форме тегировать образ Docker просто.

Листинг 3.19. Простая команда docker tag



Вашему образу дается имя, к которому можно обращаться, например:

```
docker run imagename
```

Это гораздо проще, чем запоминать случайные строки букв и цифр! Если вы хотите поделиться своими образами с другими, добавление тегов этим не ограничивается. К сожалению, терминология вокруг тегов может быть довольно запутанной. Такие термины, как *имя образа* и *репозиторий*, используются взаимозаменяемо. В табл. 3.1 приведено несколько определений.

Таблица 3.1 ❖ Термины, относящиеся к присвоению тегов

Термин	Значение
Образ	Слой только для чтения
Имя	Имя вашего образа, например «todoapp»
Тег	Будучи глаголом, относится к присвоению имени образу. Как существительное является модификатором имени вашего образа
Репозиторий	Размещенная коллекция тегированных образов, которые вместе создают файловую систему контейнера.

Возможно, наиболее запутанными терминами в этой таблице являются «образ» и «репозиторий». Мы использовали термин «образ» в общих чертах для обозначения набора слоев, из которого создаем контейнер, но технически образ представляет собой один слой, который рекурсивно относится к его родительскому слою. *Репозиторий* размещен, что означает, что он где-то хранится (либо в вашем демоне Docker, либо в реестре). Кроме того, репозиторий представляет собой набор образов с тегами, которые составляют файловую систему контейнера.

Здесь может быть полезна аналогия с Git. При клонировании репозитория Git вы проверяете состояние файлов в запрашиваемой вами точке. Это аналог образа. Репозиторий – это полная история файлов при каждом коммите, начиная с первоначального. Поэтому вы извлекаете версию на «уровне» текущей ветки. Другие «уровни» (или коммиты) находятся в репозитории, который вы клонировали.

На практике термины «образ» и «репозиторий» используются более или менее взаимозаменяемо, поэтому не стоит слишком волноваться по этому поводу. Но имейте в виду, что эти термины существуют и используются аналогично.

Пока вы увидели, как присвоить имя идентификатору образа. По непонятной причине это имя не является тегом образа, хотя часто его таковым считают. Мы различаем действие «тегировать» (глагол) и «тег» (существительное), который вы можете присвоить имени образа. Этот тег позволяет именовать конкретную версию образа. Вы можете добавить тег для управления ссылками на разные версии одного и того же образа. Например, можно тегировать образ, используя название версии или дату фиксации изменений.

Хорошим примером репозитория с несколькими тегами является образ Ubuntu. Если вы извлечете образ Ubuntu, а затем выполните команду `docker images`, то получите вывод, подобный тому, что приведен в следующем листинге.

Листинг 3.20. Образ с несколькими тегами

```
$ docker images
REPOSITORY      TAG          IMAGE ID          CREATED          VIRTUAL SIZE
ubuntu          trusty       8eaa4ff06b53      4 weeks ago      192.7 MB
ubuntu          14.04        8eaa4ff06b53      4 weeks ago      192.7 MB
ubuntu          14.04.1      8eaa4ff06b53      4 weeks ago      192.7 MB
ubuntu          latest       8eaa4ff06b53      4 weeks ago      192.7 MB
```

В столбце Repository содержится размещенная коллекция слоев, которая называется «ubuntu». Она часто упоминается как «образ». В столбце Tag перечислены четыре разных имени (trusty, 14.04, 14.04.1 и latest). Столбец Image ID содержит идентичные идентификаторы образов, потому что эти образы с разными метками идентичны.

Это показывает, что у вас может быть репозиторий с несколькими тегами с одним и тем же идентификатором образа. Однако теоретически эти теги могут позже указывать на разные идентификаторы образов. Например, если «trusty» получает обновление безопасности, автор может изменить идентификатор образа новым коммитом, и тот будет помечен как «trusty», «14.04.2» и «latest».

По умолчанию вашему образу присваивается тег «latest», если тег не указан.

> **ПРИМЕЧАНИЕ.** Тег «latest» не имеет особого значения в Docker – он используется по умолчанию для тегирования и извлечения данных. Это *не* обязательно означает, что это был последний набор тегов для данного образа. Тег вашего образа «latest» может быть старой версией образа, так как созданные позже могут быть помечены специальным тегом, таким как «v1.2.3».

ОБСУЖДЕНИЕ

В этом разделе мы рассмотрели присвоение тегов образам Docker. Сама по себе данная методика относительно проста. Реальная проблема, с которой мы столкнулись – и сосредоточились на ней – заключается в том, чтобы ориентироваться в свободном использовании терминологии среди пользователей Docker. Стоит еще раз подчеркнуть, что, когда речь идет об образе, может иметься в виду образ с тегом или даже репозиторий. Еще одна распространенная ошибка – обращение к образу как к контейнеру: «Просто скачайте контейнер и запустите его». Коллеги на работе, которые использовали Docker в течение некоторого времени, все еще часто спрашивают нас: «В чем разница между контейнером и образом?»

Из следующего метода вы узнаете, как делиться своим теперь тегированным образом с другими, используя Docker Hub.

МЕТОД 17 Совместное использование образов в Docker Hub

Присвоение тегов образам с использованием описательных имен было бы еще более полезным, если бы вы могли делиться этими именами (и образами) с другими. Чтобы удовлетворить эту потребность, у Docker есть возможность легко перемещать образы в другие места, а Docker Inc. создала Docker Hub в качестве бесплатного сервиса для поощрения такого использования.

> **ПРИМЕЧАНИЕ.** Чтобы следовать этой методике, вам понадобится учетная запись в Docker Hub, куда вы вошли ранее, запустив команду `docker login` на своем хост-компьютере.

Если у вас нет учетной записи, можете завести ее по адресу http://hub.docker.com. Просто следуйте инструкциям, чтобы зарегистрироваться.

ПРОБЛЕМА

Вы хотите сделать образ Docker общедоступным.

РЕШЕНИЕ

Используйте реестр Docker Hub.

Как и в случае с присвоением тегов, терминология касательно реестров может сбивать с толку. Таблица 3.2 должна помочь вам понять, как используются эти термины.

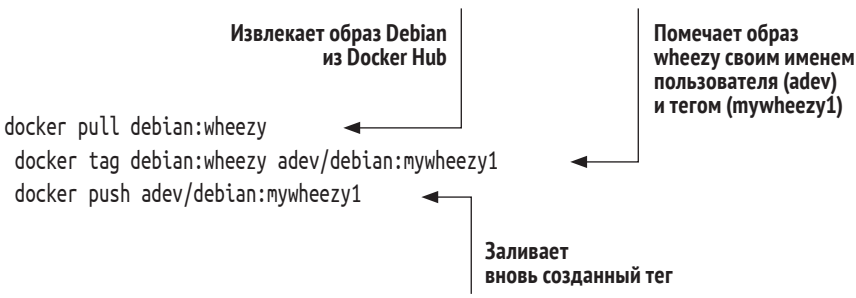
Таблица 3.2 ❖ Термины, относящиеся к реестру Docker

Термин	Значение
Имя пользователя	Ваше имя пользователя
Реестр	В реестрах хранятся образы. Реестр – это хранилище, куда вы можете загружать образы или скачать их. Реестры могут быть общедоступными или частными
Хост реестра	Хост, на котором работает реестр Docker
Docker Hub	Общедоступный реестр по умолчанию, размещенный по адресу https://hub.docker.com
Индекс	То же, что и хост реестра. Похоже, это устаревший термин

Как вы уже видели ранее, присвоить тег образу можно столько раз, сколько вы захотите. Это полезно для «копирования» образа, чтобы вы могли им управлять.

Допустим, ваше имя пользователя в Docker Hub – «adev». Следующие три команды показывают, как скопировать образ «debian: wheezy» из Docker Hub, чтобы он принадлежал вашей учетной записи.

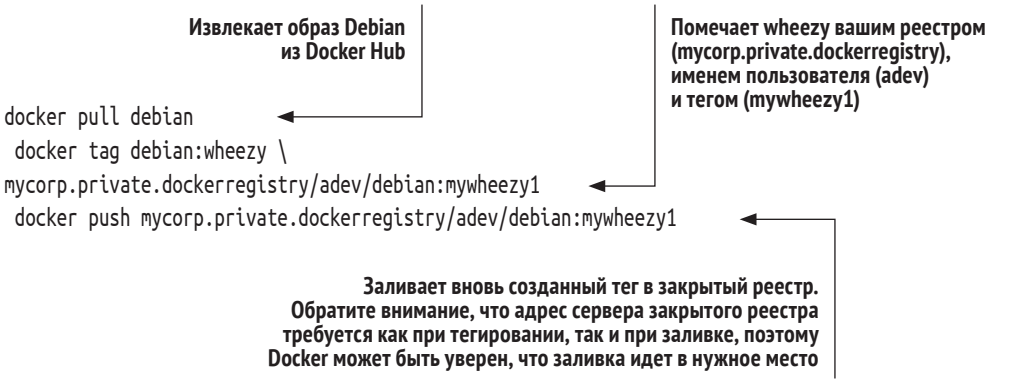
Листинг 3.21. Копирование общедоступного образа и помещение его в учетную запись Docker Hub



Теперь у вас есть ссылка на загруженный образ Debian wheezy, который можно поддерживать, обращаться к нему и использовать.

Если имеется частный репозиторий, процесс идентичен, за исключением того, что вы должны указать адрес реестра перед тегом. Допустим, у вас есть репозиторий, который обслуживается с http://mycorp.private.dockerregistry. В приведенном ниже листинге мы присваиваем образу тег и размещаем его.

Листинг 3.22. Копирование общедоступного образа и помещение его в частный реестр adev



Предыдущие команды не будут помещать образ в общедоступный Docker Hub, а поместят его в закрытый репозиторий, чтобы любой, у кого есть доступ к ресурсам этой службы, мог его извлечь.

ОБСУЖДЕНИЕ

Теперь у вас есть возможность поделиться своими образами с другими. Это отличный способ рассказать о работе, идеях или даже проблемах, с которыми вы сталкиваетесь, с другими инженерами.

Так же, как GitHub – не единственный общедоступный сервер Git, Docker Hub – это не единственный общедоступный реестр Docker. Но, как и GitHub, он самый популярный. Например, у RedHat есть хаб по адресу https://access.redhat.com/containers.

Опять же, подобно серверам Git, общедоступные и частные реестры Docker могут иметь различные функции и характеристики, которые, возможно, вас заинтересуют. Если вы оцениваете их, то можете подумать о таких вещах, как стоимость (купить, подписаться или поддерживать), соблюдение API, функции безопасности и производительность.

В следующем методе мы рассмотрим, как можно обращаться к конкретным образам, чтобы избежать проблем, возникающих, когда ссылка на образ, которую вы используете, не является конкретной.

МЕТОД 18 Обращение к конкретному образу в сборках

Большую часть времени вы будете обращаться к обобщенным именам образов в ваших сборках, таким как «node» или «ubuntu», и будете работать без проблем.

Если вы обращаетесь к имени образа, возможно, что образ способен измениться, в то время как тег остается таким же, как это ни парадоксально звучит. Имя репозитория – всего лишь ссылка, и его можно изменить, чтобы

указывать на другой базовый образ. Указание тега с нотацией двоеточия (например, `ubuntu: trusty`) также не устраняет эту опасность, так как обновления безопасности могут использовать тот же тег для повторной сборки уязвимых образов в автоматическом режиме.

В большинстве случаев вам этого захочется – авторы образа могли бы найти улучшение, и исправление дыр в безопасности – как правило, хорошая вещь. Однако иногда это может вас огорчить. И здесь имеет место не просто теоретический риск: с нами такое случалось неоднократно, нарушались сборки непрерывной доставки таким образом, что это сложно было отладить. На заре появления Docker пакеты добавлялись и удалялись из самых популярных образов регулярно (включая, в одном памятном случае, исчезновение команды `passwd`!), что приводило к внезапной поломке сборок, которые раньше работали.

ПРОБЛЕМА

Вы хотите быть уверены, что ваша сборка создана из конкретного и неизменного образа.

РЕШЕНИЕ

Чтобы быть абсолютно уверенным в том, что вы выполняете сборку для определенного набора файлов, укажите идентификатор конкретного образа в своем файле Dockerfile.

Вот пример (который, вероятно, вам не подойдет):

Листинг 3.23. Файл Dockerfile с идентификатором конкретного образа

```
FROM 8eaa4ff06b53
 RUN echo "Built from image id:" > /etc/buildinfo
 RUN echo "8eaa4ff06b53" >> /etc/buildinfo
 RUN echo "an ubuntu 14.4.01 image" >> /etc/buildinfo
 CMD ["echo","/etc/buildinfo"]
```

Выполняет сборку из идентификатора конкретного образа (или слоя)

Запускает команды в этом образе, чтобы записать образ, который вы собрали из файла в новый образ

Собранный образ по умолчанию выведет информацию, которую вы записали в файл /etc/buildinfo

Чтобы выполнить сборку из идентификатора конкретного образа (или слоя), подобного этому, идентификатор образа и его данные должны храниться локально на вашем демоне Docker. Реестр Docker не будет осуществлять какой-либо поиск, чтобы найти идентификатор в слоях образов, доступных вам в Docker Hub, или в любом другом реестре, который вы можете использовать.

Обратите внимание, что образ, к которому вы обращаетесь, не должен быть помечен – это может быть любой слой, который у вас есть локально. Вы можете начать сборку из любого слоя, какого пожелаете. Это может быть полезно

для определенных оперативных или экспериментальных процедур, которые вы хотите выполнить для анализа сборки файла Dockerfile.

Если вы хотите хранить образ удаленно, лучше присвоить ему метку и поместить в репозиторий, которое находится под вашим контролем в удаленном реестре.

> **ВНИМАНИЕ!** Стоит отметить, что может возникнуть почти противоположная проблема, когда образ Docker, который ранее работал, внезапно не работает. Обычно это происходит из-за того, что в сети что-то изменилось. Одним из запоминающихся примеров был случай, когда однажды утром у наших сборок возникла проблема с командой `apt-get update`. Мы предположили, что это связано с нашим локальным deb-кешем, и безуспешно пытались провести отладку, пока дружелюбный системный администратор не обратил внимание на то, что конкретная версия Ubuntu, из которой мы выполняли сборку, больше не поддерживается. Это означало, что сетевые вызовы `apt-get update` возвращали HTTP-ошибку.

ОБСУЖДЕНИЕ

Важно понимать преимущества и недостатки уточненного описания образа, который вы хотите собрать или запустить.

Более конкретная информация делает результат ваших действий еще больше предсказуемым и отлаживаемым, поскольку существует меньше сомнений относительно того, какой образ Docker скачивается или был скачан. Недостаток состоит в том, что ваш образ может быть не самым последним из доступных, и поэтому вы можете пропустить критические обновления. Какое состояние дел вы предпочитаете, будет зависеть от вашего конкретного варианта использования и назначения приоритетов в среде Docker.

В следующем разделе вы примените полученные знания к отчасти игровому сценарию реального мира: победить в игре 2048.

3.3. Среда как процесс

Один из способов изучения Docker – увидеть, как среда превращается в процесс. Виртуальные машины можно рассматривать таким же образом, но Docker делает это намного более удобным и эффективным.

В качестве иллюстрации мы покажем вам, как быстрый запуск, хранение и воссоздание состояния контейнера может позволить вам сделать нечто иное, (почти) невозможное – победить в игре 2048!

МЕТОД 19 Подход «сохранить игру»: победа в игре 2048

Этот метод разработан, для того чтобы дать вам немного отдохнуть и показать, как можно использовать Docker для легкого возврата состояния. Если

вы не знакомы с 2048, то это захватывающая игра, в которой вы нажимаете цифры на доске. Оригинальная версия доступна в режиме онлайн на странице http://gabrielecirulli.github.io/2048, если вы хотите сначала ознакомиться с ней.

ПРОБЛЕМА

Вы хотите регулярно сохранять состояние контейнера, чтобы при необходимости вернуться к известному состоянию.

РЕШЕНИЕ

Используйте команду `docker commit`, чтобы «сохранить игру» всякий раз, когда вы не уверены, победите ли вы в 2048.

Мы создали монолитный образ, на котором вы можете играть в 2048 в контейнере Docker, содержащем VNC-сервер и Firefox.

Чтобы использовать этот образ, вам нужно установить VNC-клиент. Популярные реализации включают в себя TigerVNC и VNC Viewer. Если у вас его нет, быстрый поиск по запросу «vnc client» в менеджере пакетов на вашем хосте должен дать полезные результаты.

Чтобы запустить контейнер, выполните следующие команды.

Листинг 3.24. Запуск контейнера 2048

```
$ docker run -d -p 5901:5901 -p 6080:6080 --name win2048 imiell/win2048
 $ vncviewer localhost:1
```

Запускаем образ imiell/win2048 в качестве демона

Используйте VNC, чтобы получить доступ к контейнеру через графический интерфейс пользователя

Сначала запускаете контейнер из образа `imiell/win2048`, который мы для вас подготовили.

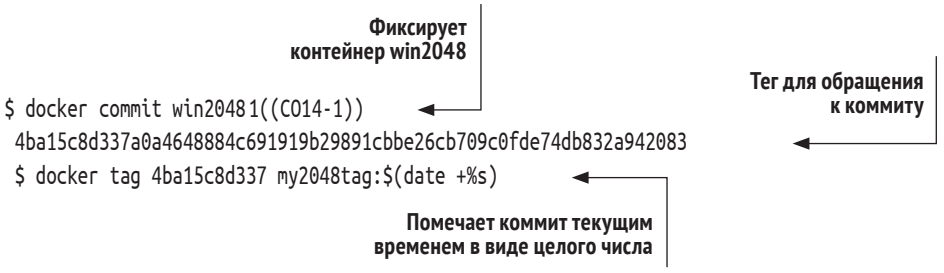
Вы запускаете его в фоновом режиме и указываете, что он должен открыть два порта (5901 и 6080) для хоста. Эти порты будут использоваться VNC-сервером, автоматически запускаемым внутри контейнера. Позже вы также дадите для удобства контейнеру имя: `win2048`.

Теперь запускаете VNC-клиента (исполняемый файл может отличаться в зависимости от того, что вы установили) и даете ему команду подключиться к локальному компьютеру. Поскольку соответствующие порты были открыты из контейнера, подключение к localhost будет на самом деле означать подключение к контейнеру. Значение «1» после `localhost` подходит, если на вашем хосте нет дисплеев X, кроме стандартного рабочего стола – если они есть, вам может потребоваться выбрать другое число и посмотреть документацию для своего VNC-клиента, чтобы вручную указать VNC-порт как `5901`.

Когда подключитесь к VNC-серверу, вам будет предложено ввести пароль. Пароль для VNC в этом образе – «vncpass». Вы увидите окно с вкладкой Firefox и предварительно загруженной таблицей 2048. Нажмите на нее, чтобы выделить, и играйте, пока не будете готовы сохранить игру.

Чтобы сохранить игру, пометьте именованный контейнер после фиксации изменений:

Листинг 3.25. Фиксируем и помечаем состояние игры



Идентификатор образа был сгенерирован путем фиксации контейнера `win2048`, и теперь вы хотите дать ему уникальное имя (потому что, возможно, создаете несколько таких образов). Для этого нужно использовать вывод `date +%s` как часть имени образа. Будет выведено количество секунд, начиная с первого дня 1970 года, что обеспечит уникальное (для наших целей), постоянно увеличивающееся значение. Синтаксис `$(command)` просто заменяет вывод `command` в данной позиции. Вместо этого можете запустить `date +%s` вручную и вставить вывод как часть имени образа, если вам так больше нравится.

Продолжайте играть, пока не проиграете. Теперь наступает волшебство! Вы можете вернуться к точке сохранения с помощью следующих команд.

Листинг 3.26. Возврат к сохраненной игре

```
$ docker rm -f win2048
$ docker run -d -p 5901:5901 -p 6080:6080 --name win2048 my2048tag:$mytag
```

`$mytag` – это тег, выбранный из команды `docker images`. Повторите шаги `tag`, `rm` и `run`, пока не закончите 2048.

ОБСУЖДЕНИЕ

Мы надеемся, что было весело. Этот пример скорее забавен, чем практичен, но мы использовали – и видели, как другие разработчики используют – этот метод для большого эффекта, особенно когда их среда сложна, а работа, которую они выполняют, в какой-то степени не лишена разных каверз.

Резюме

- Вы можете создать контейнер Docker, который выглядит как «обычный» хост. Некоторые считают, что это плохая практика, но это может принести пользу вашему бизнесу или соответствовать вашему варианту использования.
- Относительно легко преобразовать виртуальную машину в образ Docker, чтобы осуществить первоначальный переход в Docker.
- Вы можете контролировать службы в своих контейнерах, чтобы имитировать их предыдущую работу в стиле BM.
- Фиксация изменений является верным способом сохранить вашу работу по мере продвижения.
- Вы можете указать конкретный образ Docker для сборки, используя его идентификатор сборки.
- Вы можете дать имена своим образам и поделиться ими со всем миром в Docker Hub бесплатно.
- Вы даже можете использовать возможность фиксации Docker, чтобы одержать победу в таких играх, как 2048!

Глава 4

Сборка образов

О чем рассказывается в этой главе:

- основы создания образов;
- управление кешем сборки Docker для быстрой и надежной сборки;
- настройка часовых поясов как части сборки образа;
- запуск команд непосредственно в ваших контейнерах из хоста;
- подробное изучение слоев, созданных сборкой образа;
- применение более продвинутой функции ONBUILD при сборке и использовании образов.

Чтобы выйти за рамки основ использования Docker, вы захотите начать создавать свои собственные строительные блоки (образы), которые будут извлекаться совместно интересными способами. В этой главе будут рассмотрены некоторые важные части создания образов, а также практические аспекты, с которыми вы, возможно, столкнетесь.

4.1. Сборка образов

Хотя простота файлов Dockerfile делает их мощным инструментом, который экономит время, есть некоторые тонкости, которые могут вызвать путаницу. Мы расскажем вам о некоторых функциях, экономящих время, и их подробностях, начиная с инструкции `ADD`; затем о кеше сборки Docker, о том, как он может вас подвести и как можно управлять им в своих интересах.

Не забудьте обратиться к официальной документации Docker для получения полных инструкций по файлам Dockerfile на странице https://docs.docker.com.

МЕТОД 20 Внедрение файлов в образ с помощью ADD

Хотя можно добавлять файлы в Dockerfile с помощью команды RUN и базовых примитивов оболочки, это может быстро стать неуправляемым. Команда ADD была добавлена в список команд Dockerfile, чтобы решить вопрос с необходимостью помещать большое количество файлов в образ без суеты.

ПРОБЛЕМА

Вы хотите скачать и распаковать tar-архив в свой образ в сжатой форме.

РЕШЕНИЕ

Распакуйте и сожмите ваши файлы, используя директиву ADD в файле Dockerfile.

Создайте новую среду для этой сборки Docker с помощью команды `mkdir add_example && cd add_example`. Затем найдите tar-архив и дайте ему имя, к которому вы сможете обращаться в дальнейшем.

Листинг 4.1. Загрузка TAR-файла

```
$ curl \
https://www.flamingspork.com/projects/libeatmydata/
➥ libeatmydata-105.tar.gz > my.tar.gz
```

В этом случае мы использовали TAR-файл из другого метода, но это может быть любой архив, который вам нравится.

Листинг 4.2. Добавление TAR-файла к образу

```
FROM debian
RUN mkdir -p /opt/libeatmydata
ADD my.tar.gz /opt/libeatmydata/
RUN ls -lRt /opt/libeatmydata
```

Соберите этот файл Dockerfile с помощью команды `docker build --no-cache`. Вывод должен выглядеть так:

Листинг 4.3. Сборка образа с помощью TAR-файла

```
$ docker build --no-cache .
Sending build context to Docker daemon 422.9 kB
Sending build context to Docker daemon
Step 0 : FROM debian
 ---> c90d655b99b2
Step 1 : RUN mkdir -p /opt/libeatmydata
 ---> Running in fe04bac7df74
```

```
 ---> c0ab8c88bb46
Removing intermediate container fe04bac7df74
Step 2 : ADD my.tar.gz /opt/libeatmydata/
 ---> 06dcd7a88eb7
Removing intermediate container 3f093a1f9e33
Step 3 : RUN ls -lRt /opt/libeatmydata
 ---> Running in e3283848ad65
/opt/libeatmydata:
total 4
drwxr-xr-x 7 1000 1000 4096 Oct 29 23:02 libeatmydata-105

/opt/libeatmydata/libeatmydata-105:
total 880
drwxr-xr-x 2 1000 1000 4096 Oct 29 23:02 config
drwxr-xr-x 3 1000 1000 4096 Oct 29 23:02 debian
drwxr-xr-x 2 1000 1000 4096 Oct 29 23:02 docs
drwxr-xr-x 3 1000 1000 4096 Oct 29 23:02 libeatmydata
drwxr-xr-x 2 1000 1000 4096 Oct 29 23:02 m4
-rw-r--r-- 1 1000 1000 9803 Oct 29 23:01 config.h.in
[...edited...]
-rw-r--r-- 1 1000 1000 1824 Jun 18 2012 pandora_have_better_malloc.m4
-rw-r--r-- 1 1000 1000  742 Jun 18 2012 pandora_header_assert.m4
-rw-r--r-- 1 1000 1000  431 Jun 18 2012 pandora_version.m4
 ---> 2ee9b4c8059f
Removing intermediate container e3283848ad65
Successfully built 2ee9b4c8059f
```

Из этого вывода видно, что tar-архив распакован в целевой каталог демоном Docker (расширенный вывод всех файлов был отредактирован).

Docker будет распаковывать tar-файлы большинства стандартных типов (.gz, .bz2, .xz, .tar).

Стоит отметить, что, хотя вы можете скачивать tar-архивы с URL-адресов, они будут распакованы автоматически, только если хранятся в локальной файловой системе. Это может привести к путанице.

Если вы повторите предыдущий процесс, используя приведенный ниже файл Dockerfile, то заметите, что файл загружен, но не распакован.

Листинг 4.4. Непосредственное добавление TAR-файла из URL-адреса

```
FROM debian
RUN mkdir -p /opt/libeatmydata
ADD \
 https://www.flamingspork.com/projects/libeatmydata/libeatmydata-105.tar.gz \
```

Файл извлекается из интернета с помощью URL-адреса

```
/opt/libeatmydata/
 RUN ls -lRt /opt/libeatmydata
```

Целевой каталог обозначается именем каталога и косой чертой. Без этой черты аргумент рассматривается как имя загруженного файла

Вот итоговый вывод сборки:

```
Sending build context to Docker daemon 422.9 kB
Sending build context to Docker daemon
Step 0 : FROM debian
 ---> c90d655b99b2
Step 1 : RUN mkdir -p /opt/libeatmydata
 ---> Running in 6ac454c52962
 ---> bdd948e413c1
Removing intermediate container 6ac454c52962
Step 2 : ADD \
https://www.flamingspork.com/projects/libeatmydata/libeatmydata-105.tar.gz
➥ /opt/libeatmydata/
Downloading [==================================================>] \
419.4 kB/419.4 kB
 ---> 9d8758e90b64
Removing intermediate container 02545663f13f
Step 3 : RUN ls -lRt /opt/libeatmydata
 ---> Running in a947eaa04b8e
/opt/libeatmydata:
total 412
-rw------- 1 root root 419427 Jan 11970 \
libeatmydata-105.tar.gz
 ---> f18886c2418a
Removing intermediate container a947eaa04b8e
Successfully built f18886c2418a
```

Файл libeatmydata-105.tar.gz был скачан и помещен в каталог /opt/libeatmydata без распаковки

Обратите внимание, что без косой черты в строке с ADD в предыдущем Dockerfile файл будет скачан и сохранен с этим именем. Косая черта указывает на то, что файл должен быть загружен и помещен в указанный каталог.

Все новые файлы и каталоги принадлежат пользователю root (или тому, у кого есть идентификаторы группы или пользователя 0 внутри контейнера).

Пробелы в именах файлов

Если в именах ваших файлов есть пробелы, вам нужно использовать вариант ADD с кавычками (или COPY):

```
ADD "space file.txt" "/tmp/space file.txt"
```

ОБСУЖДЕНИЕ

Инструкция ADD – это рабочая лошадка с множеством различных функциональных возможностей, которыми вы можете воспользоваться. Если вы собираетесь написать более пары файлов Dockerfile (что, скорее всего, вы и будете делать, читая эту книгу), стоит прочитать официальную документацию с инструкциями по Dockerfile – их не так много (на момент написания этих строк на странице https://docs.docker.com/engine/reference/builder в документации перечислены 18 инструкций), и вы будете регулярно использовать только некоторые из них.

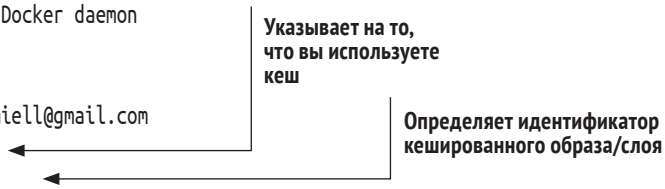
Люди часто спрашивают о добавлении сжатых файлов без их извлечения. Для этого следует использовать команду COPY, которая выглядит точно так же, как команда ADD, но не распаковывает файлы и не будет ничего скачивать через интернет.

МЕТОД 21 Повторная сборка без кеша

Сборка с помощью файлов Dockerfile использует полезную функцию кеширования: повторная сборка уже созданных шагов выполняется только в случае изменения команд. В следующем листинге показан вывод повторной сборки приложения из главы 1.

Листинг 4.5. Повторная сборка с кешем

```
$ docker build .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM node
 ---> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
 ---> Using cache
 ---> 8f5a8a3d9240
Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.git
 ---> Using cache
 ---> 48db97331aa2
Step 3 : WORKDIR todo
 ---> Using cache
 ---> c5c85db751d6
Step 4 : RUN npm install > /dev/null
 ---> Using cache
 ---> be943c45c55b
Step 5 : EXPOSE 8000
 ---> Using cache
 ---> 805b18d28a65
Step 6 : CMD npm start
```



```
 ---> Using cache
 ---> 19525d4ec794
Successfully built 19525d4ec794
```

Выполнена повторная сборка окончательного образа, но на самом деле ничего не изменилось

Как бы это ни было полезно и экономило время, это не всегда то поведение, которое вам нужно.

Взяв предыдущий файл Dockerfile в качестве примера, представьте, что вы изменили свой исходный код и поместили его в репозиторий Git. Новый код не будет извлечен, потому что команда `git clone` не изменилась. Что касается сборки Docker, она та же, поэтому кешированный образ можно использовать повторно.

В этих случаях вы захотите восстановить свой образ без использования кеша.

ПРОБЛЕМА

Вы хотите выполнить повторную сборку своего файла Dockerfile без использования кеша.

РЕШЕНИЕ

Чтобы принудительно выполнить повторную сборку без использования кеша образов, выполните команду `docker build` с флагом `--no-cache`. Следующий листинг запускает предыдущую сборку с флагом `--no-cache`.

Листинг 4.6. Принудительная повторная сборка без использования кеша

```
$ docker build --no-cache .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM node
 ---> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
 ---> Running in ca243b77f6a1
 ---> 602f1294d7f1
Removing intermediate container ca243b77f6a1
Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.git
 ---> Running in f2c0ac021247
 ---> 04ee24faaf18
Removing intermediate container f2c0ac021247
Step 3 : WORKDIR todo
 ---> Running in c2d9cd32c182
 ---> 4e0029de9074
Removing intermediate container c2d9cd32c182
Step 4 : RUN npm install > /dev/null
 ---> Running in 79122dbf9e52
npm WARN package.json todomvc-swarm@0.0.1 No repository field.
```

Выполняет повторную сборку образа Docker, игнорируя кешированные слои, используя флаг --no-cache

На этот раз нет упоминания о кешировании

У промежуточных образов другой идентификатор, в отличие от предыдущего листинга

```
 ---> 9b6531f2036a
Removing intermediate container 79122dbf9e52
Step 5 : EXPOSE 8000
 ---> Running in d1d58e1c4b15
 ---> f7c1b9151108
Removing intermediate container d1d58e1c4b15
Step 6 : CMD npm start
 ---> Running in 697713ebb185
 ---> 74f9ad384859
Removing intermediate container 697713ebb185
Successfully built 74f9ad384859
```

Новый образ собран

Выходные данные не содержат упоминания о кешировании, и каждый идентификатор промежуточного слоя отличается от вывода в листинге 4.5.

Подобные проблемы могут возникнуть в других ситуациях. Мы были в замешательстве на раннем этапе использования файлов Dockerfile, когда сбой в сети означал, что команда не получала что-то из сети должным образом, но она не выдавала ошибку. Мы продолжали вызывать команду `docker build`, но полученная ошибка не исчезла! Это было потому, что «плохой» образ нашёл дорогу к кешу, и мы не знали, как работает кеширование Docker. В конце концов удалось понять это.

ОБСУЖДЕНИЕ

Запрет кеширования может быть полезной проверкой работоспособности после получения окончательного файла Dockerfile, чтобы убедиться, что он полностью рабочий, особенно когда вы используете внутренние веб-ресурсы в своей компании, которые, возможно, изменили во время итерации файла Dockerfile.

Эта ситуация не возникает, если вы используете `ADD`, потому что Docker будет скачивать файл каждый раз, чтобы проверить, изменился ли он; но такое поведение может быть утомительным, если вы вполне уверены, что он останется прежним, и просто хотите приступить к написанию оставшейся части файла Dockerfile.

МЕТОД 22 Запрет кеширования

Использование флага `--no-cache` часто достаточно, чтобы обойти любые проблемы с кешем, но иногда требуется более детальное решение. Например, если у вас есть сборка, которая отнимает много времени, вы можете использовать кеш до определенной точки, а затем аннулировать его, чтобы повторно запустить команду и создать новый образ.

ПРОБЛЕМА

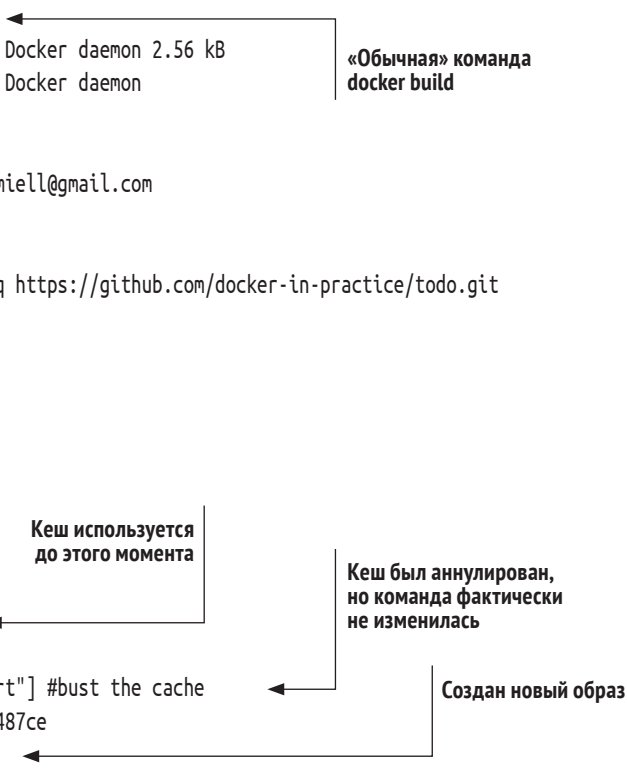
Вы хотите аннулировать кеш сборки Docker из определенной точки в сборке файла Dockerfile.

РЕШЕНИЕ

Добавьте безобидный комментарий после команды для аннулирования кеша.

Начиная с файла Dockerfile на странице https://github.com/docker-in-practice/todo (что соответствует строкам с `Step` в следующем выводе), мы выполнили сборку, а затем добавили комментарий в Dockerfile в строке с `CMD`. Вы снова можете увидеть результат выполнения команды `docker build` здесь:

```
$ docker build .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM node
 ---> 91cbcf796c2c
Step 1 : MAINTAINER ian.miell@gmail.com
 ---> Using cache
 ---> 8f5a8a3d9240
Step 2 : RUN git clone -q https://github.com/docker-in-practice/todo.git
 ---> Using cache
 ---> 48db97331aa2
Step 3 : WORKDIR todo
 ---> Using cache
 ---> c5c85db751d6
Step 4 : RUN npm install
 ---> Using cache
 ---> be943c45c55b
Step 5 : EXPOSE 8000
 ---> Using cache
 ---> 805b18d28a65
Step 6 : CMD ["npm","start"] #bust the cache
 ---> Running in fc6c4cd487ce
 ---> d66d9572115e
Removing intermediate container fc6c4cd487ce
Successfully built d66d9572115e
```



Причина, по которой этот трюк работает, заключается в том, что Docker рассматривает изменение, не являющееся пробелом, в строке, как если бы это была новая команда, поэтому кешированный слой не используется повторно.

Вы можете задаться вопросом (как мы это делали, когда впервые увидели Docker), можно ли перемещать слои Docker из образа в образ, объединяя их по своему желанию, как если бы они были наборами изменений в Git. В настоящее время это невозможно. Слой определяется как набор изменений только для данного образа. Из-за этого, как только кеширование было запрещено, его нельзя повторно применять для команд, использованных повторно позже в сборке. По этой причине вам рекомендуется размещать команды, которые

с меньшей вероятностью изменятся, ближе к верхней части файла Dockerfile, если это возможно.

ОБСУЖДЕНИЕ

Для начальной итерации в файле Dockerfile разделение каждой отдельной команды на отдельный слой отлично подходит для ускорения итерации, поскольку вы можете выборочно перезапускать части процесса, как показано в предыдущем листинге, но это не так хорошо для создания небольшого итогового образа. Это не является беспрецедентным для сборок с разумной степенью сложности, чтобы приблизиться к жесткому пределу в 42 слоя. Чтобы смягчить эту ситуацию, как только у вас будет рабочая сборка, которой вы довольны, вы должны взглянуть на шаги, приведенные в методе 56 для создания готового образа.

МЕТОД 23 Умный запрет кеширования с помощью build-args

В предыдущем методе вы увидели, что кеширование можно запретить в середине сборки путем изменения соответствующей строки.

В этом методе мы сделаем еще один шаг вперед, контролируя, отключено ли кеширование или нет из команды сборки.

ПРОБЛЕМА

Вы хотите запретить кеширование по требованию при выполнении сборки, не редактируя Dockerfile.

РЕШЕНИЕ

Используйте директиву `ARG` в вашем файле Dockerfile, чтобы привести в действие оперативный запрет кеширования.

Чтобы продемонстрировать это, вы снова будете использовать Dockerfile по адресу https://github.com/docker-in-practice/todo, но внесите в него небольшие изменения.

Вы хотите контролировать запрет кеширования перед `npm install`.

Зачем это делать? Как известно, по умолчанию Docker запретит кеширование, только если команда в Dockerfile изменится. Но давайте представим, что есть обновленные пакеты npm, и вы хотите убедиться, что получаете их. Одним из вариантов является ручное изменение строки (как видно из предыдущего метода), но, чтобы добиться того же более элегантным способом, нужно использовать директиву `ARGS` и трюк с bash.

Добавьте строку `ARG` в Dockerfile следующим образом.

Листинг 4.7. Простой файл Dockerfile с кешем, который можно «обойти»

```
WORKDIR todo
ARG CACHEBUST=no
RUN npm install
```

Директива ARG устанавливает переменную среды для сборки

В этом примере вы используете директиву ARG, чтобы установить переменную среды CACHEBUST, и ее значение по умолчанию равным no, если оно не установлено командой docker build.

Теперь соберем этот файл Dockerfile «нормально»:

```
$ docker build .
Sending build context to Docker daemon 2.56kB
Step 1/7 : FROM node
latest: Pulling from library/node
aa18ad1a0d33: Pull complete
15a33158a136: Pull complete
f67323742a64: Pull complete
c4b45e832c38: Pull complete
f83e14495c19: Pull complete
41fea39113bf: Pull complete
f617216d7379: Pull complete
cbb91377826f: Pull complete
Digest: sha256:
➥ a8918e06476bef51ab83991aea7c199bb50bfb131668c9739e6aa7984da1c1f6
Status: Downloaded newer image for node:latest
 ---> 9ea1c3e33a0b
Step 2/7 : MAINTAINER ian.miell@gmail.com
 ---> Running in 03dba6770157
 ---> a5b55873d2d8
Removing intermediate container 03dba6770157
Step 3/7 : RUN git clone https://github.com/docker-in-practice/todo.git
 ---> Running in 23336fd5991f
Cloning into 'todo'...
 ---> 8ba06824d184
Removing intermediate container 23336fd5991f
Step 4/7 : WORKDIR todo
 ---> f322e2dbeb85
Removing intermediate container 2aa5ae19fa63
Step 5/7 : ARG CACHEBUST=no
 ---> Running in 9b4917f2e38b
 ---> f7e86497dd72
Removing intermediate container 9b4917f2e38b
Step 6/7 : RUN npm install
 ---> Running in a48e38987b04
npm info it worked if it ends with ok
[...]
added 249 packages in 49.418s
npm info ok
 ---> 324ba92563fd
Removing intermediate container a48e38987b04
```

```
Step 7/7 : CMD npm start
 ---> Running in ae76fa693697
 ---> b84dbc4bf5f1
Removing intermediate container ae76fa693697
Successfully built b84dbc4bf5f1
```

Если вы соберете его снова с помощью точно такой же команды `docker build`, то увидите, что используется кеш сборки Docker, и в результирующий образ не вносится никаких изменений.

```
$ docker build .
Sending build context to Docker daemon 2.56kB
Step 1/7 : FROM node
 ---> 9ea1c3e33a0b
Step 2/7 : MAINTAINER ian.miell@gmail.com
 ---> Using cache
 ---> a5b55873d2d8
Step 3/7 : RUN git clone https://github.com/docker-in-practice/todo.git
 ---> Using cache
 ---> 8ba06824d184
Step 4/7 : WORKDIR todo
 ---> Using cache
 ---> f322e2dbeb85
Step 5/7 : ARG CACHEBUST=no
 ---> Using cache
 ---> f7e86497dd72
Step 6/7 : RUN npm install
 ---> Using cache
 ---> 324ba92563fd
Step 7/7 : CMD npm start
 ---> Using cache
 ---> b84dbc4bf5f1
Successfully built b84dbc4bf5f1
```

На этом этапе вы решаете, что хотите принудительно выполнить повторную сборку пакетов npm. Возможно, ошибка была исправлена или вы хотите быть в курсе последних изменений. Вот тут-то и появляется переменная `ARG`, которую вы добавили в Dockerfile в листинге 4.7. Если для этой переменной задано значение, которое ранее никогда не использовалось на вашем хосте, кеширование будет отключено из этой точки.

Здесь используется флаг `build-arg` для команды `docker build` вместе с трюком bash, чтобы получить новое значение:

```
$ docker build --build-arg CACHEBUST=${RANDOM} .
Sending build context to Docker daemon 4.096 kB
Step 1/9 : FROM node
 ---> 53d4d5f3b46e
```

Выполняем команду docker build с флагом build-arg, установив аргумент CACHEBUST в псевдослучайное значение, сгенерированное bash

```
Step 2/9 : MAINTAINER ian.miell@gmail.com
 ---> Using cache
 ---> 3a252318543d
Step 3/9 : RUN git clone https://github.com/docker-in-practice/todo.git
 ---> Using cache
 ---> c0f682653a4a
Step 4/9 : WORKDIR todo
 ---> Using cache
 ---> bd54f5d70700
Step 5/9 : ARG CACHEBUST=no
 ---> Using cache
 ---> 3229d52b7c33
Step 6/9 : RUN npm install
 ---> Running in 42f9b1f37a50
npm info it worked if it ends with ok
npm info using npm@4.1.2
npm info using node@v7.7.2
npm info attempt registry request try #1 at 11:25:55 AM
npm http request GET https://registry.npmjs.org/compression
npm info attempt registry request try #1 at 11:25:55 AM
[...]
Step 9/9 : CMD npm start
 ---> Running in 19219fe5307b
 ---> 129bab5e908a
Removing intermediate container 19219fe5307b
Successfully built 129bab5e908a
```

Поскольку строка ARG CACHEBUST = no сама по себе не изменилась, здесь используется кеш

Поскольку аргумент CACHEBUST был установлен в ранее неустановленное значение, кеширование отключено, и команда npm install запускается снова

Обратите внимание, что запрет кеширования осуществляется в строке, *следующей* за строкой ARG, а не в ней самой.

Это может быть немного запутанным. Главное, на что нужно обратить внимание, – это фраза «Running in», которая означает, что был создан новый контейнер для запуска строки сборки.

Использование аргумента ${RANDOM} заслуживает объяснения. Bash предоставляет вам это зарезервированное имя переменной, чтобы вы могли легко получить значение длиной от одной до пяти цифр:

```
$ echo ${RANDOM}
19856
$ echo ${RANDOM}
26429
$ echo ${RANDOM}
2856
```

Это может пригодиться, например, когда вы хотите, чтобы вероятно уникальное значение создавало файлы только для определенного запуска сценария.

Можно даже создать гораздо более длинное случайное число, если вас беспокоит конфликт:

```
$ echo ${RANDOM}${RANDOM}
434320509
$ echo ${RANDOM}${RANDOM}
1327340
```

Обратите внимание, что, если вы не используете bash (или оболочку, в которой доступна переменная `RANDOM`), этот метод не сработает. В таком случае можно задать вместо этого команду `date` для получения нового значения:

```
$ docker build --build-arg CACHEBUST=$(date +%s) .
```

Данный метод продемонстрировал ряд моментов, которые пригодятся при использовании Docker. Вы узнали о применении флага `--build-args` для передачи значения в Dockerfile и запрета кеширования по требованию, создавая новую сборку без изменения Dockerfile.

Если вы используете bash, вы также узнали о переменной `RANDOM` и о том, как она может быть полезна в других контекстах, нежели просто сборки Docker.

МЕТОД 24 Умный запрет кеширования с помощью директивы ADD

В предыдущем методе вы увидели, как можно запретить кеширование в середине сборки по вашему выбору, что само по себе на уровень выше использования флага `--no-cache` для полного игнорирования кеша.

Теперь перейдем на следующий уровень, чтобы иметь возможность автоматически запрещать кеширование только тогда, когда необходимо. Это может сэкономить вам уйму времени, а следовательно, и денег!

ПРОБЛЕМА

Вы хотите запретить кеширование, когда удаленный ресурс изменился.

РЕШЕНИЕ

Используйте директиву `ADD`, чтобы запрещать кеширование только при изменении ответа от URL.

Одно из ранних критических замечаний по поводу файлов Dockerfile заключалось в том, что их утверждение о получении надежных результатов сборки вводило в заблуждение. Действительно, мы обсуждали эту тему с создателем Docker еще в 2013 году (http://mng.bz/B8E4).

В частности, если вы совершаете вызов сети, используя такую директиву в своем Dockerfile:

```
RUN git clone https://github.com/nodejs/node
```

то по умолчанию сборка Docker будет выполнять его по одному разу для каждого демона Docker. Код в GitHub может существенно измениться, но что

касается вашего демона Docker, сборка обновлена. Могут пройти годы, и тот же демон Docker все еще будет использовать кеш.

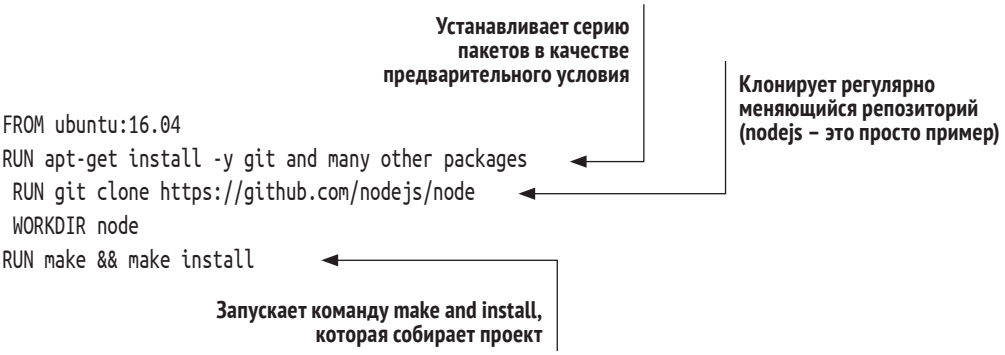
Это может звучать как теоретическая проблема, но она очень реальна для многих пользователей.

Мы видели, как это случалось много раз во время работы, вызывая путаницу. Вы уже встречались с решениями такой ситуации, но в случае с множеством сложных или больших сборок эти решения недостаточно детализированы.

Шаблон умного запрета кеширования

Представьте, что у вас есть файл Dockerfile, который выглядит следующим образом (обратите внимание, что он не будет работать! Это просто шаблон Dockerfile, демонстрирующий принцип).

Листинг 4.8. Образец файла Dockerfile



Этот файл представляет некоторые проблемы для создания эффективного процесса сборки. Если вы хотите каждый раз собирать все с нуля, решение простое: используйте аргумент `--no-cache` для команды `docker build`. Проблема состоит в том, что каждый раз, когда вы запускаете сборку, то повторяете установку пакета во второй строке, что (в основном) ненужно.

Эту проблему можно решить, запретив кеширование непосредственно перед командой `git clone` (продемонстрировано в последнем методе). Однако возникает еще одна проблема: что, если репозиторий Git не изменился? Тогда вы выполняете потенциально дорогостоящий перенос по сети, а затем потенциально дорогостоящую команду `make`. Сеть, вычисления и ресурсы диска – все они используются без необходимости.

Один из способов обойти это – использовать метод 23, где вы передаете аргумент сборки с новым значением каждый раз, когда узнаете, что удаленный репозиторий изменился.

Но это все еще требует ручного расследования, чтобы определить, были ли изменения и вмешательство.

Вам нужна команда, которая может определить, изменился ли ресурс с момента последней сборки, и только затем запретить кеширование.

Директива ADD – неожиданные преимущества

Введите директиву ADD!

Вы уже знакомы с ADD, так как это основная директива Dockerfile. Обычно она используется для добавления файла к результирующему образу, но у нее есть два полезных свойства, которые вы можете применять в своих целях в этом контексте: она кеширует содержимое файла, к которому обращается, и может использовать сетевой ресурс в качестве аргумента. Это означает, что вы можете запрещать кеширование всякий раз, когда меняется вывод веб-запроса.

Как можно воспользоваться этим при клонировании репозитория? Ну, это зависит от характера ресурса, к которому вы обращаетесь по сети. У многих ресурсов будет страница, которая меняется при изменении самого хранилища, но они будут различаться в зависимости от типа ресурса. Здесь мы сосредоточимся на репозиториях GitHub, поскольку это общераспространенный вариант использования.

GitHub API предоставляет полезный ресурс, который может помочь в данном случае. Он содержит URL-адреса для каждого репозитория, которые возвращают JSON для самых последних коммитов. Когда делается новый коммит, содержание ответа изменяется.

Листинг 4.9. Использование ADD для запуска запрета кеширования

```
FROM ubuntu:16.04
ADD https://api.github.com/repos/nodejs/node/commits
➥ /dev/null
 RUN git clone https://github.com/nodejs/node
 [...]
```

URL-адрес, который изменяется при создании нового коммита

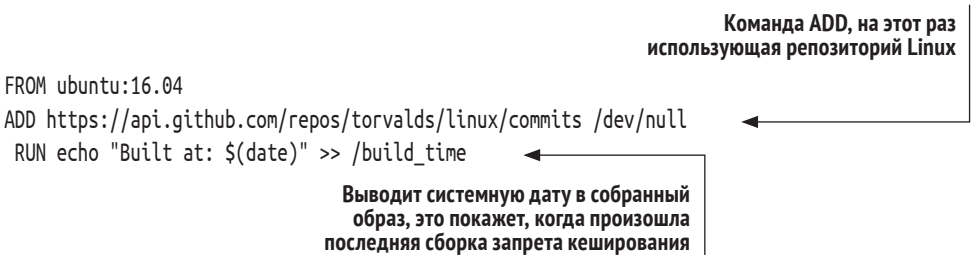
Неважно, куда выводится файл, поэтому мы отправляем его в /dev/null

Команда git clone будет иметь место только после внесения изменения

Результатом предыдущего листинга является то, что кеширование отключается только тогда, когда была сделана фиксация в репозитории с момента последней сборки. Никакого вмешательства человека не требуется, никакой ручной проверки не нужно.

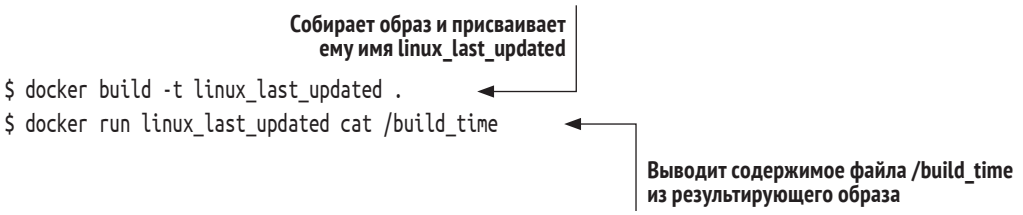
Если вы хотите протестировать этот механизм с часто меняющимся хранилищем, попробуйте использовать ядро Linux.

Листинг 4.10. Добавление кода ядра Linux к образу



Если вы создадите папку и поместите предыдущий код в Dockerfile, а затем будете регулярно выполнять следующую команду (например, каждый час), дата вывода будет меняться только при изменении репозитория Linux Git.

Листинг 4.11. Сборка образа Linux



ОБСУЖДЕНИЕ

Этот метод продемонстрировал ценную автоматическую технику, обеспечивая выполнение сборок только при необходимости, а также детали того, как работает команда ADD. Вы увидели, что «файл» может быть сетевым ресурсом, и что если содержимое файла (или сетевого ресурса) отличается от предыдущей сборки, то произойдет запрет кеширования.

Кроме того, вы также видели, что у сетевых ресурсов есть связанные ресурсы, которые могут указывать на то, изменился ли ресурс, к которому вы обращаетесь. Хотя вы бы могли, например, обратиться к главной странице GitHub, чтобы увидеть, есть ли там какие-либо изменения, вполне вероятно, что страница меняется чаще, чем при последнем коммите (например, если время веб-ответа скрыто в источнике страницы или в каждом ответе содержится уникальная ссылочная строка).

В случае с GitHub вы можете обращаться к API. Другие сервисы, такие как BitBucket, предлагают аналогичные ресурсы. Например, проект Kubernetes предлагает этот URL-адрес, где можно найти стабильную версию: https://storage.googleapis.com/kubernetes-release/release/stable.txt. Если бы вы собирали проект на основе Kubernetes, вы могли бы добавить строку с ADD в свой Dockerfile, чтобы запретить кеширование при каждом изменении этого ответа.

МЕТОД 25 Установка правильного часового пояса в контейнерах

Если вы когда-либо устанавливали полную операционную систему, вы знаете, что установка часового пояса – это часть процесса ее настройки. Даже если контейнер не является операционной системой (или виртуальной машиной), он содержит файлы, которые сообщают программам, как интерпретировать время для настроенного часового пояса.

ПРОБЛЕМА

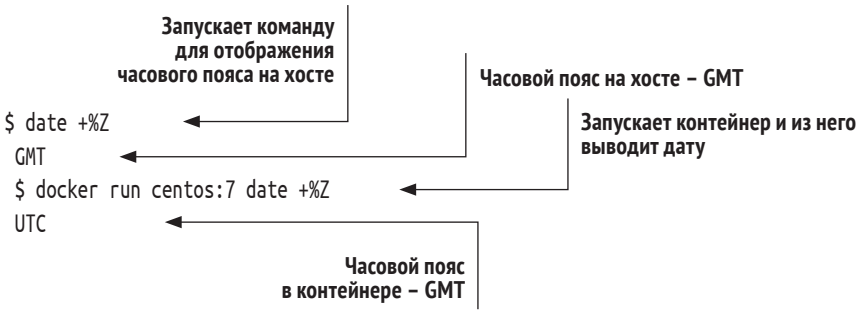
Вы хотите правильно установить часовой пояс для своих контейнеров.

РЕШЕНИЕ

Замените файл локального времени контейнера ссылкой на нужный часовой пояс.

Следующий листинг демонстрирует эту проблему. Неважно, где вы запускаете его, контейнер будет показывать один и тот же часовой пояс.

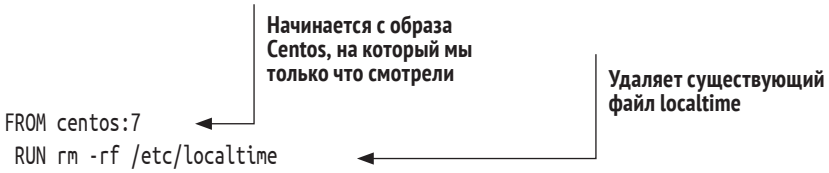
Листинг 4.12. Контейнер запускается с неправильным часовым поясом



Контейнер содержит файлы, определяющие, какой часовой пояс используется контейнером для интерпретации временного значения, которое он получает. Фактическое используемое время, конечно, отслеживается операционной системой хоста.

Следующий листинг показывает, как установить часовой пояс на тот, который вам нужен.

Листинг 4.13. Dockerfile для замены часового пояса по умолчанию centos: 7



```
RUN ln -s /usr/share/zoneinfo/GMT /etc/localtime
CMD date +%Z
```

Заменяет ссылку /etc/localtime ссылкой на нужный вам часовой пояс

Показывает часовой пояс вашего контейнера как команду по умолчанию для запуска

В листинге 4.13 ключевым файлом является /etc/localtime. Он указывает на файл, который сообщает контейнеру, какой часовой пояс использовать, когда он запрашивает время. Время по умолчанию указано в стандарте UTC, который применяется, если файл не существует (в минимальном образе BusyBox, например, его нет).

В следующем листинге показан вывод сборки предыдущего файла Dockerfile.

Листинг 4.14. Сборка файла Dockerfile для замены часового пояса

```
$ docker build -t timezone_change .
 Sending build context to Docker daemon 62.98 kB
Step 1 : FROM centos:7
7: Pulling from library/centos
45a2e645736c: Pull complete
Digest: sha256:
➥ c577af3197aacedf79c5a204cd7f493c8e07ffbce7f88f7600bf19c688c38799
Status: Downloaded newer image for centos:7
 ---> 67591570dd29
Step 2 : RUN rm -rf /etc/localtime
 ---> Running in fb52293849db
 ---> 0deda41be8e3
Removing intermediate container fb52293849db
Step 3 : RUN ln -s /usr/share/zoneinfo/GMT /etc/localtime
 ---> Running in 47bf21053b53
 ---> 5b5cb1197183
Removing intermediate container 47bf21053b53
Step 4 : CMD date +%Z
 ---> Running in 1e481eda8579
 ---> 9477cdaa73ac
Removing intermediate container 1e481eda8579
Successfully built 9477cdaa73ac
$ docker run timezone_change
 GMT
```

Собирает контейнер

Запускает контейнер

Выводит указанный часовой пояс

Таким образом, вы можете указать часовой пояс для использования внутри и только внутри вашего контейнера. От этого параметра зависят многие приложения, поэтому он нередко возникает, если вы запускаете службу Docker.

Есть еще одна проблема, которую может решить это временное разбиение на уровне контейнера. Если вы работаете в многонациональной организации и запускаете множество различных приложений на серверах, расположенных

в центрах обработки данных по всему миру, то возможность изменить часовой пояс в вашем образе и полагать, что он сообщит правильное время, где бы ни находился, – это прием, который полезно иметь под рукой.

ОБСУЖДЕНИЕ

Поскольку смысл образов Docker явно заключается в том, чтобы обеспечить единообразное взаимодействие независимо от того, где вы запускаете свой контейнер, есть ряд вещей, с которыми вы можете столкнуться, если все же нужны разные результаты в зависимости от того, где развернут образ.

Например, если вы автоматически создаете таблицы данных CSV для пользователей в разных местах, у них могут быть определенные ожидания в отношении формата данных.

Американские пользователи ожидают даты в формате мм/дд, тогда как европейцы – в формате дд/мм, а пользователи из Китая – в своем собственном наборе символов.

В следующем методе мы рассмотрим настройки локали, которые, помимо прочего, влияют на то, как даты и время выводятся в локальном формате.

МЕТОД 26 Управление локалями

В дополнение к часовым поясам локали – это еще один аспект образов Docker, который может иметь значение при сборке образов или запуске контейнеров.

> **ПРИМЕЧАНИЕ.** Локаль определяет, какие настройки языка и страны должны использоваться вашими программами. Обычно региональный стандарт задается в среде с помощью переменных `LANG`, `LANGUAGE` и `locale-gen`, а также с помощью переменных, начинающихся с `LC_`, таких как `LC_TIME`, настройка которых определяет способ отображения времени для пользователя.

> **ПРИМЕЧАНИЕ.** Кодировка (в этом контексте) – это средство, с помощью которого текст хранится в виде байтов на компьютере. Хорошее введение в эту тему доступно от W3C на странице https://www.w3.org/International/questions/qa-what-is-encoding. Стоит потратить время на то, чтобы разобрать эту тему, так как она подходит для всех видов контекстов.

ПРОБЛЕМА

Вы видите ошибки кодировки в сборках или развертываниях ваших приложений.

РЕШЕНИЕ

Убедитесь, что в вашем Dockerfile правильно заданы переменные среды для языка.

Проблемы с кодировкой не всегда очевидны для всех пользователей, но могут быть фатальными при создании приложений.

Вот пара примеров типичных ошибок кодировки при сборке приложений в Docker.

Листинг 4.15. Типичные ошибки кодировки

```
MyFileDialog:66: error: unmappable character for encoding ASCII

UnicodeEncodeError: 'ascii' codec can't encode character u'\xa0' in
➥ position 20: ordinal not in range(128)
```

Эти ошибки могут напрочь убить сборку или приложение.

> **СОВЕТ.** Неисчерпывающий список ключевых слов, на которые следует обращать внимание при появлении ошибки: «кодировка», «ascii», «юникод», «UTF-8», «символ» и «кодек». Если вы видите эти слова, то, скорее всего, вы имеете дело с проблемой кодировки.

Какое это имеет отношение к Docker?

Когда вы настраиваете полнофункциональную операционную систему, вы, как правило, проходите через процесс установки, который запрашивает у вас предпочтительный часовой пояс, язык, раскладку клавиатуры и т. д.

Контейнеры Docker, как вы уже знаете, не являются полноценными операционными системами, настроенными для общего использования. Скорее, это (все более) минимальные среды для запуска приложений. Поэтому по умолчанию они могут идти не со всеми настройками, к которым вы привыкли в операционной системе.

В частности, Debian удалил свою зависимость от пакета locales в 2011 году, что означает, что по умолчанию в контейнере, основанном на образе Debian, нет настройки регионального стандарта. Например, в приведенном ниже листинге показана среда по умолчанию для образа Ubuntu, полученного из Debian.

Листинг 4.16. Среда по умолчанию в контейнере Ubuntu

```
$ docker run -ti ubuntu bash
root@d17673300830:/# env
HOSTNAME=d17673300830
TERM=xterm
LS_COLORS=rs=0 [...]
HIST_FILE=/root/.bash_history
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
```

```
HOME=/root
_=/usr/bin/envj
```

По умолчанию в образе нет доступных настроек LANG или аналогичных настроек LC_. Наш хост Docker показан в следующем листинге.

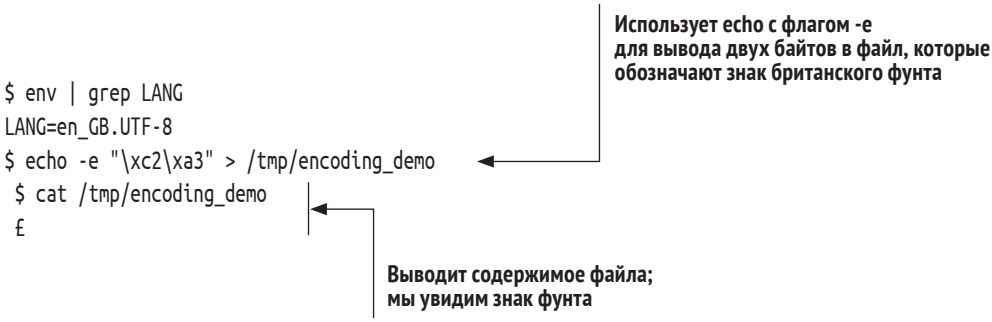
Листинг 4.17. Настройка LANG в хост-системе Docker

```
$ env | grep LANG
LANG=en_GB.UTF-8
```

В нашей оболочке есть настройка LANG, которая сообщает приложениям, что предпочтительная кодировка в нашем терминале – en_GB с текстом, закодированным в UTF.

Чтобы продемонстрировать проблему кодировки, мы локально создадим файл, который содержит символ валюты Великобритании в кодировке UTF-8 (знак британского фунта), а затем покажем, как меняется интерпретация этого файла в зависимости от кодировки терминала.

Листинг 4.18. Создание и отображение символа британской валюты в кодировке UTF-8



В UTF-8 знак фунта представлен двумя байтами. Мы выводим эти два байта, используя echo -e и нотацию \x, и перенаправляем вывод в файл. Когда выводим содержимое файла с помощью команды cat, терминал читает два байта и знает, как интерпретировать вывод в виде знака фунта.

Теперь, если мы изменим кодировку нашего терминала, чтобы использовать западную кодировку (ISO Latin 1) (которая также устанавливает нашу переменную LANG), и выведем файл, он будет выглядеть совсем иначе:

Листинг 4.19. Демонстрация проблемы кодировки с использованием символа валюты Великобритании

```
$ env | grep LANG
LANG=en_GB.ISO8859-1
```

Переменная среды LANG теперь установлена на Western (ISO Latin 1), что устанавливается терминалом

```
$ cat /tmp/encoding_demo
Â£
```

Два байта интерпретируются по-разному, как два отдельных символа, которые мы видим

Байт \xc2 интерпретируется как заглавная буква A с циркумфлексом сверху, а байт \xa3 – как знак британского фунта!

> **ПРИМЕЧАНИЕ.** Мы сознательно говорим «мы», а не «вы»! Отладка и управление кодировками – сложная задача, которая может зависеть от комбинации состояния запущенного приложения, установленных вами переменных среды, запущенного приложения и всех предшествующих факторов, которые создают исследуемые вами данные!

Как вы видели, кодировки могут зависеть от кодировки, установленной в терминале. Возвращаясь к Docker, мы заметили, что в нашем контейнере Ubuntu по умолчанию не заданы переменные среды кодировки. Из-за этого вы можете получить разные результаты при запуске одних и тех же команд на вашем хосте или в контейнере. Если вы видите ошибки, которые, по-видимому, связаны с кодировками, вам может потребоваться установить их в своем файле Dockerfile.

Установка кодировки в файле Dockerfile

Теперь рассмотрим, как управлять кодировкой образа на основе Debian. Мы выбрали этот образ, потому что он может быть одним из наиболее распространенных контекстов. В этом примере настроим простой образ, который выводит переменные среды по умолчанию.

Листинг 4.20. Настройка файла Dockerfile

Использует базовый образ, полученный из Debian

Обновляет индекс пакета и устанавливает пакет locales

Создает локаль для английского языка США в кодировке UTF-8

```
FROM ubuntu:16.04
 RUN apt-get update && apt-get install -y locales
 RUN locale-gen en_US.UTF-8
 ENV LANG en_US.UTF-8
 ENV LANGUAGE en_US:en
 CMD env
```

Устанавливает переменную среды LANG

Устанавливает переменную среды LANGUAGE

Команда env по умолчанию отобразит настройку среды для контейнера

Вам, наверное, интересно, в чем различия между переменными LANG и LANGUAGE. Вкратце LANG – это настройка по умолчанию для настроек предпочтительного языка и кодировки.

Она также обеспечивает значение по умолчанию, когда приложения ищут более конкретные настройки `LC_ *.. LANGUAGE` используется для предоставления упорядоченного списка языков, предпочитаемых приложениями, если основной язык недоступен. Дополнительную информацию можно найти, выполнив команду `man locale`.

Теперь вы можете собрать образ и запустить его, чтобы увидеть изменения.

Листинг 4.21. Сборка и запуск образа encoding

```
$ docker build -t encoding .
 [...]
$ docker run encoding
 no_proxy=*.local, 169.254/16
 LANGUAGE=en_US:en
HOSTNAME=aa9d6c8a3ff5
HOME=/root
HIST_FILE=/root/.bash_history
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
LANG=en_US.UTF-8
 PWD=/
```

Собирает образ encoding

Запускает собранный образ Docker

Переменная LANGUAGE устанавливается в среде

Переменная LANG устанавливается в среде

ОБСУЖДЕНИЕ

Как и предыдущий метод с часовыми поясами, этот метод иллюстрирует проблему, с которой сталкиваются регулярно. Как и многие из более раздражающих проблем, с которыми мы имеем дело, они не всегда дают понять, когда выполняется сборка образа, в результате чего на отладку проблем тратится время, что очень удручает. По этой причине стоит помнить об этих настройках, поддерживая тех, кто использует образы Docker.

МЕТОД 27 Шагаем по слоям с помощью image-stepper

Если вы собрали образ, содержащий несколько шагов, зачастую у вас может возникнуть желание узнать, где был впервые использован конкретный файл или в каком состоянии он находился в определенный момент сборки. Прочесывание каждого слоя образа может быть трудоемким, потому что вам придется определить порядок слоев, получить идентификатор и запускать каждый слой с использованием этого идентификатора.

Этот метод демонстрирует однострочный сценарий, который помечает каждый слой сборки по порядку, а это означает, что вам нужно только увеличить число, чтобы работать с образами и выяснить то, что нужно.

ПРОБЛЕМА

Вы хотите без труда обращаться к каждому шагу вашей сборки.

РЕШЕНИЕ

Используйте образ docker-in-practice/image-stepper, чтобы упорядочить теги для своего образа.

Для иллюстрации этого метода мы сначала покажем вам сценарий, который достигает этого результата, чтобы вы поняли, как это работает. Потом дадим вам сконструированный образ, чтобы облегчить достижение результата.

Вот простой сценарий, который помечает каждый слой данного образа (myimage) в порядке создания.

Далее идет файл Dockerfile для myimage.

Листинг 4.22. Dockerfile образа с несколькими слоями

```
FROM debian
 RUN touch /file1
 RUN touch /file2
 RUN touch /file3
 RUN touch /file4
 RUN touch /file5
 RUN touch /file6
 RUN touch /file7
 RUN touch /file8
 RUN touch /file9
 RUN touch /file10
 CMD ["cat","/file1"]
```

Использует debian в качестве базового образа

Создает 10 файлов в отдельных слоях

Выполняет специальную команду, которая выводит содержимое первого файла

Это достаточно простой Dockerfile, но с ним будет ясно, на каком этапе сборки вы находитесь.

Соберите этот образ с помощью следующей команды.

Листинг 4.23. Сборка образа myimage

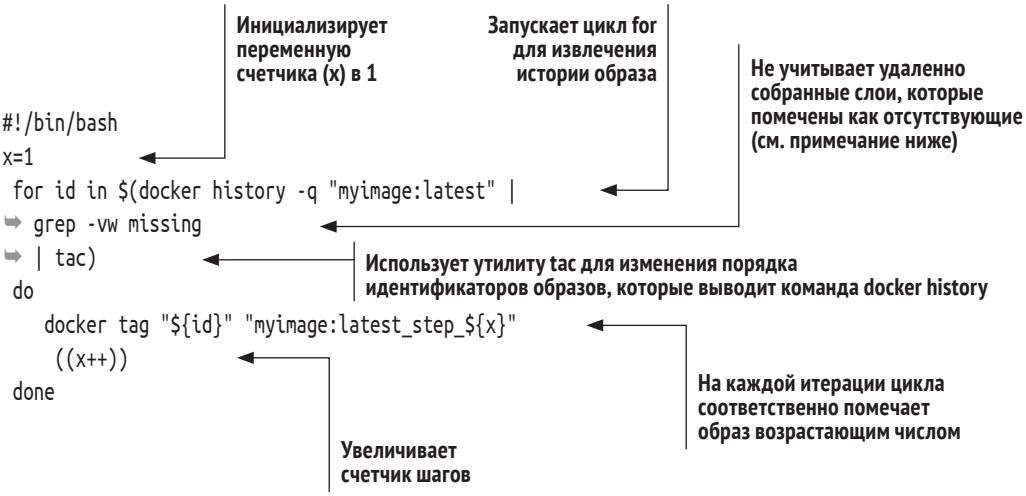
Собирает образ с использованием флага quiet (-q), помечая его как myimage

```
$ docker build -t myimage -q .
 sha256:b21d1e1da994952d8e309281d6a3e3d14c376f9a02b0dd2ecbe6cabffea95288
```

Идентификатор образа – единственное, что есть в выводе

Как только образ собран, вы можете запустить следующий сценарий.

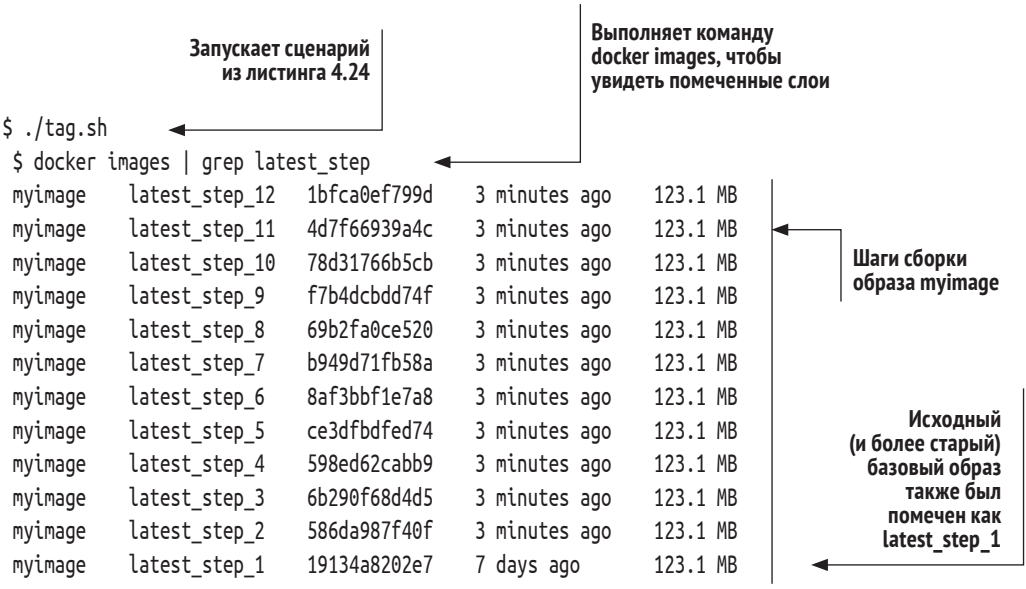
Листинг 4.24. Присваивание тега каждому слою образа в числовом порядке



Если вы сохраните предыдущий файл как tag.sh и запустите его, образ будет помечен в порядке слоя.

> **ПРИМЕЧАНИЕ.** Этот метод тегирования будет работать только для образов, собранных локально. Смотрите примечание в методе 16 для получения дополнительной информации.

Листинг 4.25. Присваивание тегов и отображение слоев



Теперь, когда вы ознакомились с этим принципом, мы продемонстрируем, как Docker'изировать этот одноразовый сценарий и заставить его работать в общем случае.

> **ПРИМЕЧАНИЕ.** Код для этого метода доступен по адресу https://github.com/dockerin-practice/image-stepper.

Для начала превратим предыдущий сценарий в сценарий, который может принимать аргументы.

Листинг 4.26. Обобщенный сценарий тегирования для образа image-stepper

```
#!/bin/bash
IMAGE_NAME=$1
IMAGE_TAG=$2
if [[ $IMAGE_NAME = '' ]]
then
    echo "Usage: $0 IMAGE_NAME [ TAG ]"
    exit 1
fi
if [[ $IMAGE_TAG = '' ]]
then
    IMAGE_TAG=latest
fi
x=1
for id in $(docker history -q "${IMAGE_NAME}:${IMAGE_TAG}" |
➥ grep -vw missing | tac)
do
    docker tag "${id}" "${IMAGE_NAME}:${IMAGE_TAG}_step_$x"
    ((x++))
done
```

Определяет сценарий bash, который может принимать два аргумента: имя образа для обработки и тег, к которому вы хотите перейти

Сценарий из листинга 4.24 с замененными аргументами

Затем вы можете встроить сценарий в листинге 4.26 в образ Docker, который помещаете в Dockerfile и запускаете в качестве ENTRYPOINT по умолчанию.

Листинг 4.27. Dockerfile образа image-stepper

```
FROM ubuntu:16.04
RUN apt-get update -y && apt-get install -y docker.io
```

Использует Ubuntu в качестве базового слоя

Устанавливает docker.io для получения двоичного файла клиента Docker

```
ADD image_stepper /usr/local/bin/image_stepper
ENTRYPOINT ["/usr/local/bin/image_stepper"]
```

Добавляет сценарий из листинга 4.26 к образу

Запускает сценарий image_stepper по умолчанию

Dockerfile в листинге 4.27 создает образ, который запускает сценарий в листинге 4.26. Команда в листинге 4.28 запускает этот образ, предоставляя `myimage` в качестве аргумента.

Этот образ при запуске с другим образом Docker, собранным на вашем хосте, будет затем создавать теги для каждого шага, что позволит вам легко просматривать слои по порядку.

Версия двоичного файла клиента, установленного пакетом docker.io, должна быть совместима с версией демона Docker на вашем хост-компьютере, что обычно означает, что клиент не должен быть моложе.

Листинг 4.28. Запуск image-stepper с другим образом

Запускает образ image-stepper как контейнер и удаляет контейнер по окончании

Монтирует сокет хоста, чтобы вы могли использовать клиент Docker, установленный в листинге 4.27

Скачивает образ image-stepper из Docker Hub

Присваивает тег myimage, созданному ранее

```
$ docker run --rm
➥ -v /var/run/docker.sock:/var/run/docker.sock
➥ dockerinpractice/image-stepper
➥ myimage
Unable to find image 'dockerinpractice/image-stepper:latest' locally
latest: Pulling from dockerinpractice/image-stepper
b3e1c725a85f: Pull complete
4daad8bdde31: Pull complete
63fe8c0068a8: Pull complete
4a70713c436f: Pull complete
bd842a2105a8: Pull complete
1a3a96204b4b: Pull complete
d3959cd7b55e: Pull complete
Digest: sha256:
➥ 65e22f8a82f2221c846c92f72923927402766b3c1f7d0ca851ad418fb998a753
Status: Downloaded newer image for dockerinpractice/image-stepper:latest
$ docker images | grep myimage
```

Вывод команды docker run

Запускает команду docker images и выполняет поиск по образам, которые вы только что пометили

```
myimage    latest           2c182dabe85c    24 minutes ago    123 MB
myimage    latest_step_12   2c182dabe85c    24 minutes ago    123 MB
myimage    latest_step_11   e0ff97533768    24 minutes ago    123 MB
myimage    latest_step_10   f46947065166    24 minutes ago    123 MB
myimage    latest_step_9    8a9805a19984    24 minutes ago    123 MB
myimage    latest_step_8    88e42bed92ce    24 minutes ago    123 MB
myimage    latest_step_7    5e638f955e4a    24 minutes ago    123 MB
myimage    latest_step_6    f66b1d9e9cbd    24 minutes ago    123 MB
myimage    latest_step_5    bd07d425bd0d    24 minutes ago    123 MB
myimage    latest_step_4    ba913e75a0b1    24 minutes ago    123 MB
myimage    latest_step_3    2ebcda8cd503    24 minutes ago    123 MB
myimage    latest_step_2    58f4ed4fe9dd    24 minutes ago    123 MB
myimage    latest_step_1    19134a8202e7    2 weeks ago       123 MB
$ docker run myimage:latest_step_8 ls / | grep file
file1
file2
file3
file4
file5
file6
file7
```

Образы помечены

Выбирает случайный шаг и выводит список файлов в корневом каталоге, выполняя поиск по файлам, созданным в Dockerfile, из листинга 4.27

Показанные файлы – это файлы, которые были созданы до этого шага

ПРИМЕЧАНИЕ. В операционных системах, отличных от Linux (таких как Mac и Windows), вам может потребоваться указать папку, в которой Docker запускается в настройках Docker в качестве параметра совместного использования файлов.

Этот метод полезен, чтобы увидеть, куда в сборке был добавлен определенный файл или в каком состоянии находился файл в определенный момент сборки. При отладке сборки это может быть неоценимо!

ОБСУЖДЕНИЕ

Этот метод используется в методе 52, чтобы продемонстрировать, что удаленный секрет доступен в пределах слоя в образе.

МЕТОД 28 Onbuild и golang

Директива `ONBUILD` может вызвать путаницу у новых пользователей Docker. Этот метод демонстрирует применение данной директивы в реальных условиях путем сборки и запуска приложения Go с использованием двухстрочного файла Dockerfile.

ПРОБЛЕМА

Вы хотите уменьшить количество шагов при сборке образа, необходимого для приложения.

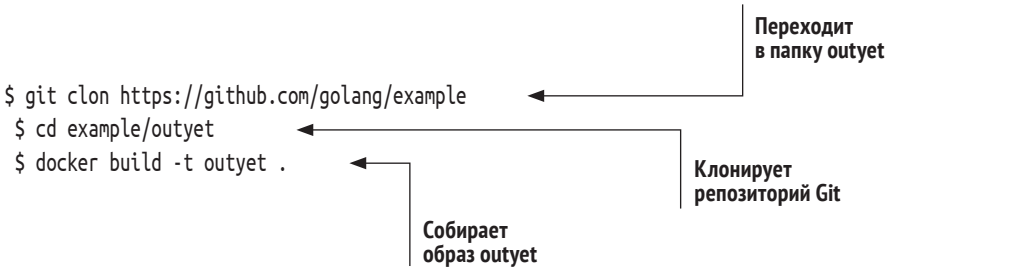
РЕШЕНИЕ

Используйте команду ONBUILD, чтобы автоматизировать и инкапсулировать сборку образа.

Сначала вы запустите процесс, а затем мы объясним, что происходит. В качестве примера будем использовать проект outyet, который является примером golang в репозитории GitHub. Все, что он делает, – это настраивает веб-сервис, который возвращает страницу, сообщающую, доступен ли Go 1.4.

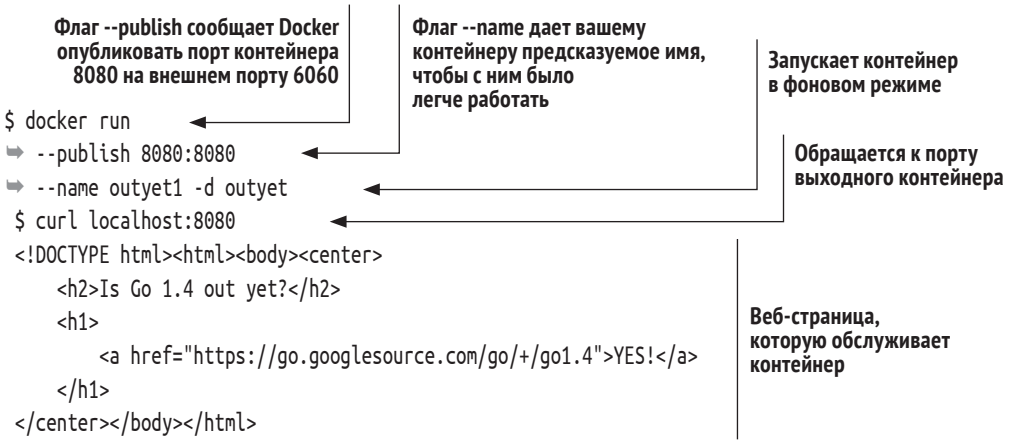
Соберите образ.

Листинг 4.29. Сборка образа outyet



Запустите контейнер из результирующего образа и получите обслуживаемую веб-страницу.

Листинг 4.30. Запуск и проверка образа outyet



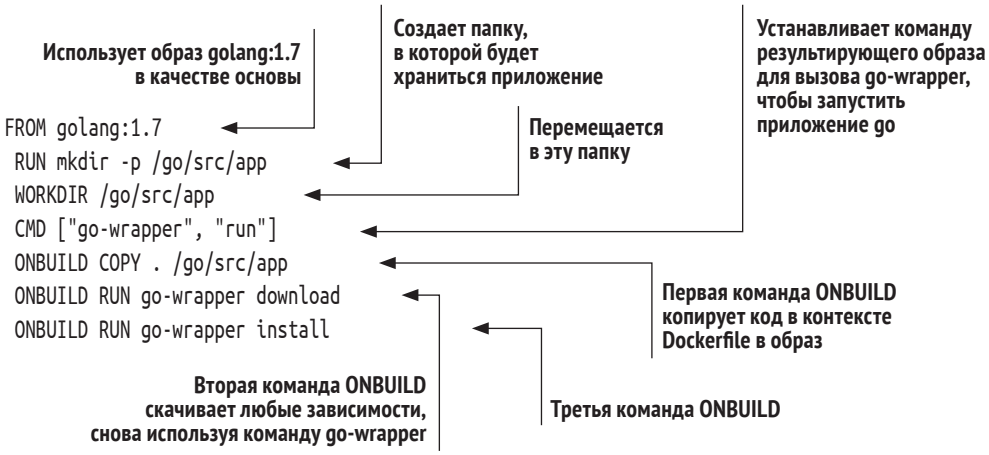
Вот и все – простое приложение, которое возвращает веб-страницу, сообщающую вам, вышел Go 1.4 или нет.

Если вы осмотрите клонированный репозиторий, то увидите, что Dockerfile состоит всего из двух строк!

Листинг 4.31. Dockerfile onyet

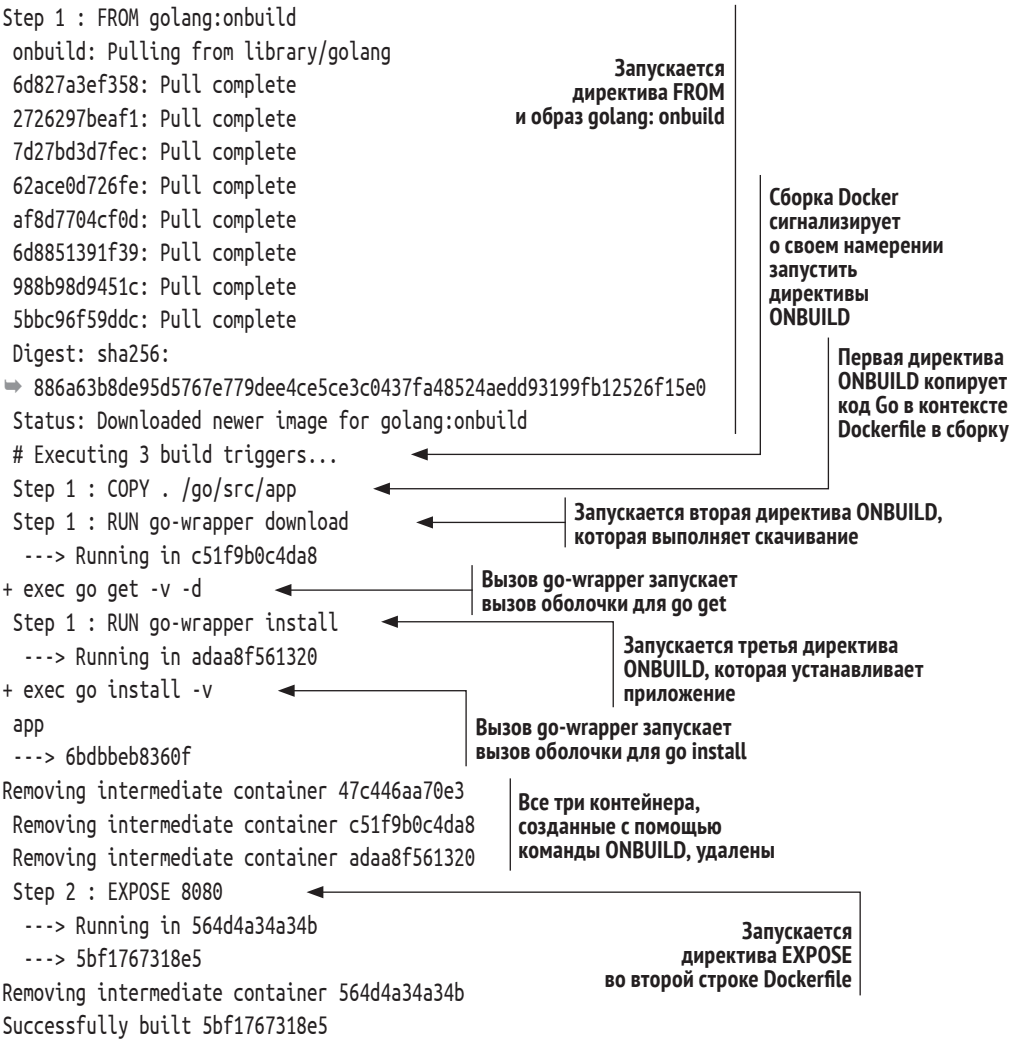
Все еще непонятно? Хорошо, это может иметь больше смысла, когда вы посмотрите на файл Dockerfile образа golang: onbuild.

Листинг 4.32. Файл Dockerfile образа golang: onbuild



Образ golang: onbuild определяет, что происходит, когда он используется в директиве `FROM` в любом другом файле Dockerfile. В результате, когда Dockerfile использует этот образ в качестве основы, команды `ONBUILD` будут запускаться сразу после скачивания образа `FROM`, а (если не переопределено) `CMD` будет запускаться, когда результирующий образ будет запущен в качестве контейнера.

Теперь вывод команды `docker build` в следующем листинге может иметь больше смысла.



Результатом этого метода является то, что у вас есть простой способ собрать образ, который содержит только код, необходимый для его запуска, и не более того. Оставляя инструменты сборки вокруг образа, вы не только делаете его больше, чем нужно, но также увеличиваете виды атак для работающего контейнера.

ОБСУЖДЕНИЕ

Поскольку Docker и Go являются модными технологиями, которые в настоящее время часто можно встретить в связке, мы брали Go, чтобы продемонстрировать, как использовать `ONBUILD` для создания двоичного файла Go.

Существуют и другие примеры образов `ONBUILD`. Это `node:onbuild` и `python:onbuild`, доступные на Docker Hub.

Мы надеемся, что это может вдохновить вас на создание собственного образа ONBUILD, который мог бы помочь вашей организации при работе с общераспространенными шаблонами сборки. Эта стандартизация может помочь еще больше уменьшить несоответствие сопротивления между различными командами.

Резюме

- Вы можете вставлять файлы со своего локального компьютера и из интернета в образы.
- Кеш является важной частью сборки образов, но он может быть ненадежным другом и иногда требует подсказки, чтобы делать то, что вы хотите.
- Вы можете «запретить» кеширование, используя аргументы сборки или директиву DD, или можете полностью игнорировать его с помощью опции no-cache.
- Директива ADD обычно используется для внедрения локальных файлов и папок в собранный образ.
- Конфигурация системы может по-прежнему иметь значение в Docker, и время сборки образа – отличное время для этого.
- Вы можете отлаживать процесс сборки, используя метод «image-stepper» (метод 27), где каждый этап сборки помечается за вас.
- Настройка часового пояса является наиболее распространенной «ошибкой» при настройке контейнеров, особенно если вы не из США или работаете в многонациональной компании.
- Образы с ONBUILD очень просты в использовании, потому что вам вообще не нужно настраивать сборку.

Глава 5

Запуск контейнеров

О чем рассказывается в этой главе:

- использование приложений с графическим интерфейсом в Docker;
- получение информации о контейнерах;
- различные способы завершения работы контейнеров;
- запуск контейнеров на удаленном компьютере;
- использование и управление томами Docker для постоянных общих данных;
- изучение первых шаблонов Docker: контейнеры данных и инструментов разработчика.

При использовании Docker вы не сможете продвинуться далеко вперед, не прибегая к запуску контейнеров, и многое придется понять, если вы хотите использовать все возможности, которые они предоставляют.

В этой главе будут рассмотрены некоторые детали, связанные с запуском контейнеров, изучены конкретные варианты использования и подробно рассмотрены возможности, попутно предоставляемые томами.

5.1. Запуск контейнеров

Хотя большая часть этой книги посвящена запуску контейнеров, есть некоторые практические приемы, связанные с запуском контейнеров на вашем хосте, которые могут быть не сразу очевидны. Мы рассмотрим, как можно заставить работать приложения с графическим интерфейсом, запустить контейнер на удаленном компьютере, проверить состояние контейнеров и их исходных образов, завершить работу контейнера, управлять демонами Docker на удаленных компьютерах и использовать DNS-службу с записями wildcards, чтобы упростить тестирование.

МЕТОД 29 Запуск графического интерфейса пользователя в Docker

Вы уже видели графический интерфейс пользователя, обслуживаемый из контейнера Docker с использованием VNC-сервера, в методе 19. Это один из способов просмотра приложений в вашем контейнере, и он является автономным, требующим использования только VNC-клиента.

К счастью, существует более легкий и хорошо интегрированный способ запуска графических интерфейсов на рабочем столе, но он требует дополнительной настройки с вашей стороны и состоит в монтировании каталога на хосте, который управляет обменом данными с X-сервером, чтобы он был доступен для контейнера.

ПРОБЛЕМА

Вы хотите запускать графические интерфейсы в контейнере, как если бы они были обычными настольными приложениями.

РЕШЕНИЕ

Создайте образ с вашими учетными данными пользователя и программой и смонтируйте туда свой X-сервер.

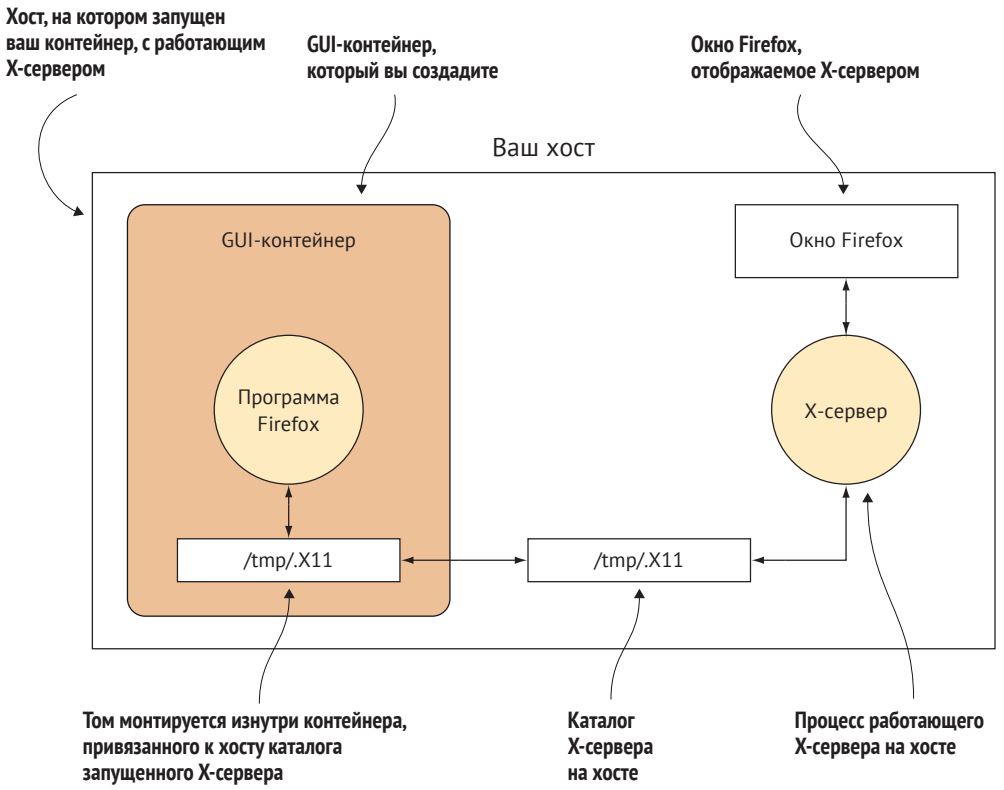


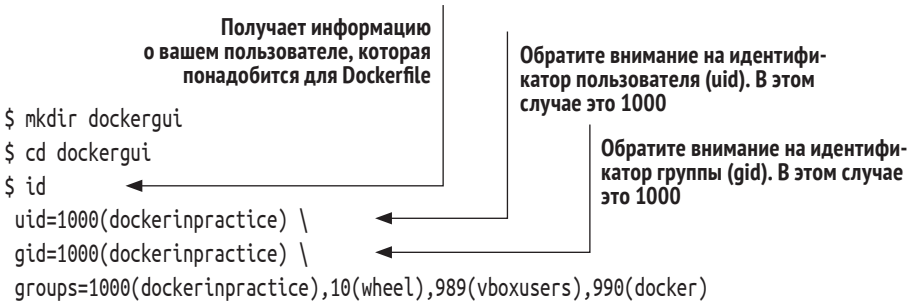
Рис. 5.1 ❖ Общение с X-сервером хоста

На рис. 5.1 показано, как будет работать окончательная настройка.

Контейнер соединяется с хостом через монтирование каталога хоста /tmp/.X11, и именно так контейнер может выполнять действия на рабочем столе хоста.

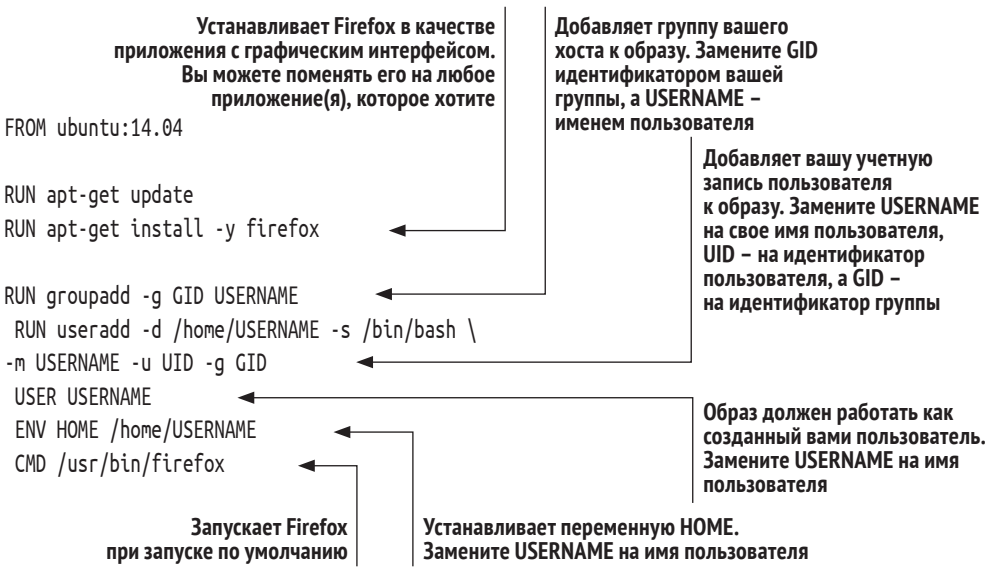
Сначала создайте новый каталог в удобном месте и определите идентификаторы пользователя и группы с помощью команды `id`, как показано в следующем листинге.

Листинг 5.1. Настройка каталога и установление пользовательских данных



Теперь создайте файл с именем Dockerfile.

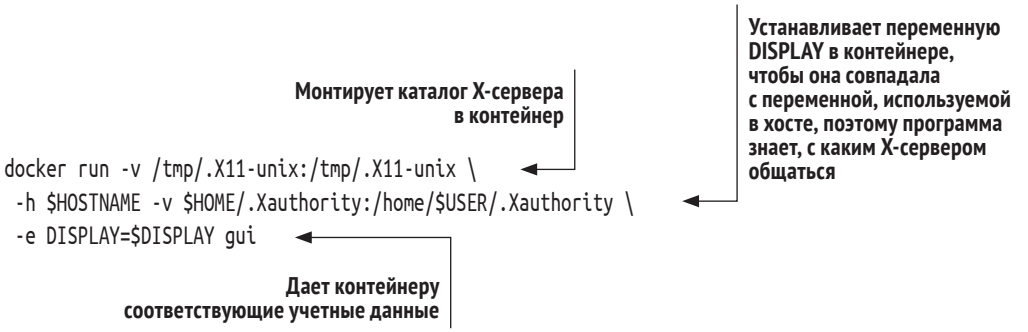
Листинг 5.2. Firefox в Dockerfile



Теперь вы можете выполнить сборку из этого Dockerfile и пометить результат как «gui»:

```
$ docker build -t gui .
```

Запустите его следующим образом:



Вы увидите всплывающее окно Firefox!

ОБСУЖДЕНИЕ

Можете использовать этот метод, чтобы не смешивать работу на десктопе с разработкой. Например, в Firefox вы, возможно, захотите увидеть, как приложение ведет себя без веб-кеша, закладок или истории поиска в воспроизводимом виде с целью тестирования. Если вы видите сообщения о невозможности открыть дисплей при попытке запустить образ и Firefox, см. метод 65 для других способов, позволяющих контейнерам запускать графические приложения, отображаемые на хосте.

Мы понимаем, что некоторые запускают почти все свои приложения в Docker, включая игры! Хотя мы не зашли так далеко, полезно знать, что кто-то, вероятно, уже столкнулся с проблемами, которые вы видите.

МЕТОД 30 Проверка контейнеров

Хотя команды Docker предоставляют вам доступ к информации об образах и контейнерах, иногда может понадобиться знать больше о внутренних метаданных этих объектов.

ПРОБЛЕМА

Вы хотите узнать IP-адрес контейнера.

РЕШЕНИЕ

Используйте команду `docker inspect`.

Команда `docker inspect` дает доступ к внутренним метаданным Docker в формате JSON, включая IP-адрес. Эта команда выводит много информации, поэтому здесь показан только краткий фрагмент метаданных образа.

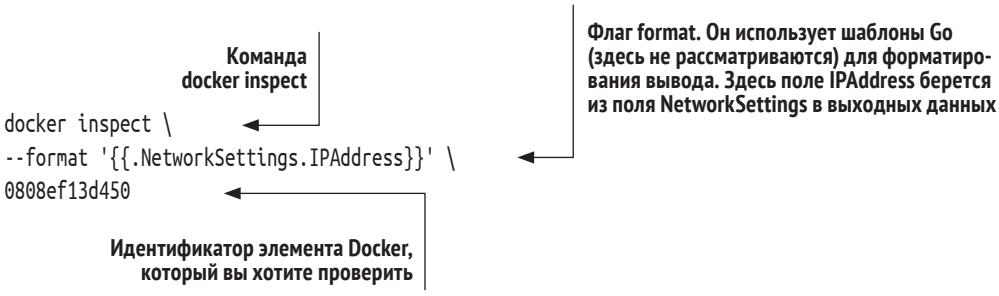
Листинг 5.3. Сырой вывод docker inspect

```
$ docker inspect ubuntu | head
[{
    "Architecture": "amd64",
    "Author": "",
    "Comment": "",
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "/bin/bash"
$
```

Вы можете проверять образы и контейнеры по имени или идентификатору. Очевидно, что их метаданные будут отличаться – например, у контейнера могут быть поля времени выполнения, такие как «Состояние», в которых образ будет отсутствовать (у образа нет состояния).

В этом случае для того чтобы узнать IP-адрес контейнера на вашем хосте, можно использовать команду `docker inspect` с флагом `format`.

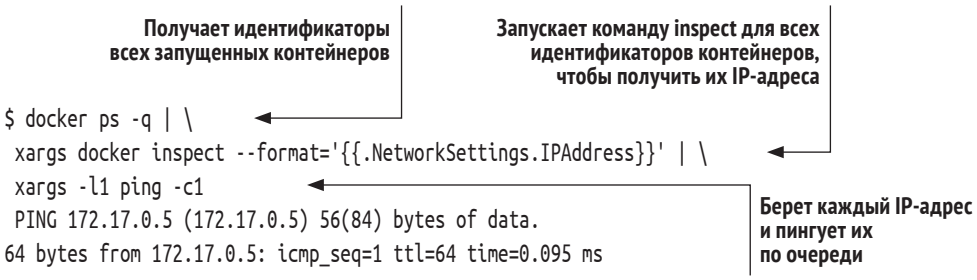
Листинг 5.4. Определение IP-адреса контейнера



Метод может быть полезен для автоматизации, так как этот интерфейс, вероятно, будет более стабильным, чем у других команд Docker.

Следующая команда дает вам IP-адреса всех запущенных контейнеров и пингует их.

Листинг 5.5. Получение IP-адресов запущенных контейнеров и проверка каждого из них по очереди




```
$ docker ps -q | \
 xargs docker inspect --format='{{.NetworkSettings.IPAddress}}' | \
 xargs -l1 ping -c1
 PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.
64 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.095 ms

--- 172.17.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.095/0.095/0.095/0.000 ms
```

Обратите внимание, что, поскольку `ping` принимает только один IP-адрес, нам пришлось передать `xargs` дополнительный аргумент, сообщив ему о необходимости запуска команды для каждой отдельной строки.

> **СОВЕТ.** Если у вас нет запущенных контейнеров, выполните следующую команду:
> `docker run -d ubuntu sleep 1000`

ОБСУЖДЕНИЕ

Проверка контейнеров и способ выполнения команд внутри уже работающего контейнера, приведенный в методе 47, вероятно, являются двумя наиболее важными инструментами в вашем инвентаре, используемом для отладки, когда контейнеры не работают. Проверка больше всего подходит, когда вы считаете, что запустили контейнер, настроенный особым образом, но он ведет себя неожиданно, – в первую очередь, вы должны проверить его, чтобы убедиться, что Docker согласен с вашими ожиданиями в отношении отображения портов и томов контейнера среди прочего.

МЕТОД 31 Чистое уничтожение контейнеров

Если состояние контейнера важно для вас, когда он завершает работу, вы, возможно, захотите понять различие между командами `docker kill` и `docker stop`. Это различие также может быть важно, если вам необходимо аккуратно закрыть приложения для сохранения данных.

ПРОБЛЕМА

Вы хотите чисто завершить работу контейнера.

РЕШЕНИЕ

Используйте команду `docker stop` вместо `docker kill` для чистого завершения работы контейнера.

Важно понять, что `docker kill` ведет себя не так, как стандартная программа `kill` из командной строки.

Программа `kill` работает, посылая сигнал `TERM` (значение сигнала 15) в указанный процесс, если не указано иное. Этот сигнал задает программе, что она должна завершить работу, но не принуждает ее. Большинство программ выполняют нечто вроде очистки при обработке этого сигнала, но программа может делать то, что ей нравится, включая игнорирование сигнала.

Напротив, сигнал `KILL` (значение сигнала 9) принудительно завершает указанную программу.

Заблуждение вызывает тот факт, что `docker kill` использует сигнал `KILL` для запущенного процесса, что не дает процессам в нем обработать завершение. Это означает, что случайные файлы, такие как файлы, содержащие идентификаторы запущенных процессов, могут быть оставлены в файловой системе. В зависимости от способности приложения управлять состоянием это может или не может создавать вам проблемы, если вы снова запустите контейнер.

Еще большую путаницу вызывает тот факт, что команда `docker stop` действует как стандартная команда `kill`, посылая сигнал `TERM` (см. табл. 5.1), за исключением того, что она будет ждать 10 секунд, а затем отправит сигнал `KILL`, если контейнер не остановился.

Таблица 5.1 ❖ Остановка и принудительное выключение

Команда	Сигнал по умолчанию	Значение сигнала по умолчанию
`kill`	`TERM`	15
`docker kill`	`KILL`	9
`docker stop`	`TERM`	15

Таким образом, не запускайте команду `docker kill` так, как если бы вы использовали `kill`. Вероятно, вам лучше всего привыкнуть к применению `docker-stop`.

ОБСУЖДЕНИЕ

Хотя мы рекомендуем команду `docker stop` для повседневного использования, `docker kill` обладает некоторыми дополнительными возможности настройки, которые позволяют выбирать сигнал, отправляемый в контейнер через аргумент `--signal`. Как уже было сказано, по умолчанию используется

значение KILL, но вы также можете отправить TERM или один из менее распространенных сигналов Unix.

Если вы пишете свое собственное приложение, которое запускаете в контейнере, сигнал USR1 может быть вам интересен. Он явно зарезервирован для приложений, чтобы делать с ними все, что захочется, а в некоторых местах он используется как указание для вывода информации о ходе выполнения; или можете использовать его по своему усмотрению. HUP – еще один популярный сигнал, который традиционно интерпретируется серверами и другими долго работающими приложениями для запуска перезагрузки файлов конфигурации и «мягкого» перезапуска. Конечно, убедитесь, что вы сверились с документацией приложения, которое запускаете, прежде чем начать отправку случайных сигналов!

МЕТОД 32 Использование Docker Machine для поддержки работы хостов Docker

Настройка Docker на вашем локальном компьютере, вероятно, была не слишком сложной – для удобства есть сценарий, который вы можете использовать, или это могут быть несколько команд, чтобы добавить соответствующие источники для вашего менеджера пакетов. Но иногда это утомительно, когда вы пытаетесь управлять установками Docker на других хостах.

ПРОБЛЕМА

Вы хотите запустить контейнеры на отдельном хосте Docker из своего компьютера.

РЕШЕНИЕ

Docker Machine является официальным решением для управления установками Docker на удаленных компьютерах.

Данный метод будет полезен, если нужно запустить контейнеры Docker на нескольких внешних хостах. Вам это может понадобиться по ряду причин: для проверки работы сетевой среды между контейнерами Docker, чтобы создать виртуальную машину для работы на собственном физическом хосте; подготовить контейнеры на более мощном компьютере через провайдера VPS; рисковать уничтожить хост каким-нибудь сумасшедшим экспериментом; иметь возможность работать на нескольких облачных провайдерах. Независимо от причины Docker Machine, вероятно, – то, что вам нужно. Docker Machine – это также ворота в более сложные инструменты оркестровки, такие как Docker Swarm.

Что такое DOCKER MACHINE

Docker Machine – это в основном удобная программа. Она оборачивает множество потенциально извилистых инструкций, поддерживая работу внешних хостов и превращая их в несколько простых команд. Если вы знакомы с Vagrant, у него похожее назначение: подготовка и управление другими машинными

средами упрощается благодаря согласованному интерфейсу. Если вы мысленно вернетесь к нашему обзору архитектуры в главе 2, то один из способов взглянуть на Docker Machine – это представить, что она облегчает управление различными демонами Docker из одного клиента (см. рис. 5.2).

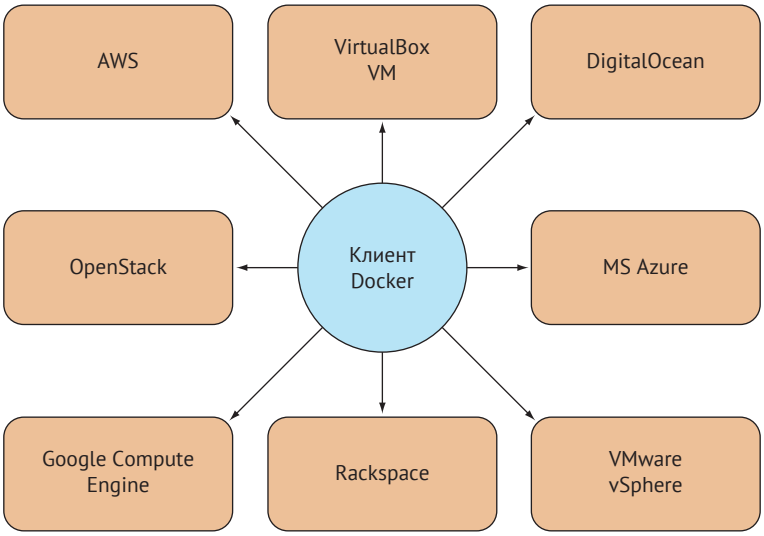


Рис. 5.2 ❖ Docker Machine как клиент внешних хостов

Список хост-провайдеров Docker на рис. 5.2 не является исчерпывающим и, вероятно, будет расти. На момент написания этой главы доступны следующие провайдеры:

- Amazon Web Services;
- Microsoft Azure;
- Rackspace;
- DigitalOcean;
- Microsoft Hyper-V;
- VMware Fusion;
- Google Compute Engine;
- OpenStack;
- VMware vCloud Air;
- IBM;
- SoftLayer;
- Oracle;
- VirtualBox;
- VMware vSphere.

Параметры, которые должны быть указаны для подготовки машины, будут сильно различаться в зависимости от функций, предоставляемых драйвером. С одной стороны, подготовка виртуальной машины Oracle VirtualBox

на вашем компьютере позволяет создать только 3 флага по сравнению с 17 флагами у OpenStack.

> **ПРИМЕЧАНИЕ.** Стоит уточнить, что Docker Machine не является каким-либо видом кластерного решения для Docker. Другие инструменты, такие как Docker Swarm, выполняют эту функцию, и мы рассмотрим их позже.

Установка

Установка включает в себя простой двоичный файл. Ссылки на скачивание и инструкции по установке для различных архитектур доступны здесь: https://github.com/docker/machine/releases.

> **ПРИМЕЧАНИЕ.** Возможно, вы захотите переместить двоичный файл в стандартное расположение, например /usr/bin, и перед продолжением убедиться, что он переименован или является символьной ссылкой на команду `docker-machine`, поскольку у скачанного файла может быть более длинное имя с суффиксом в архитектуре двоичного файла.

Использование DOCKER MACHINE

Чтобы продемонстрировать использование Docker Machine, вы можете начать с создания виртуальной машины с демоном Docker, с которым можете работать.

> **ПРИМЕЧАНИЕ.** Для этого вам понадобится установить Oracle VirtualBox. Он широко доступен в большинстве менеджеров пакетов.

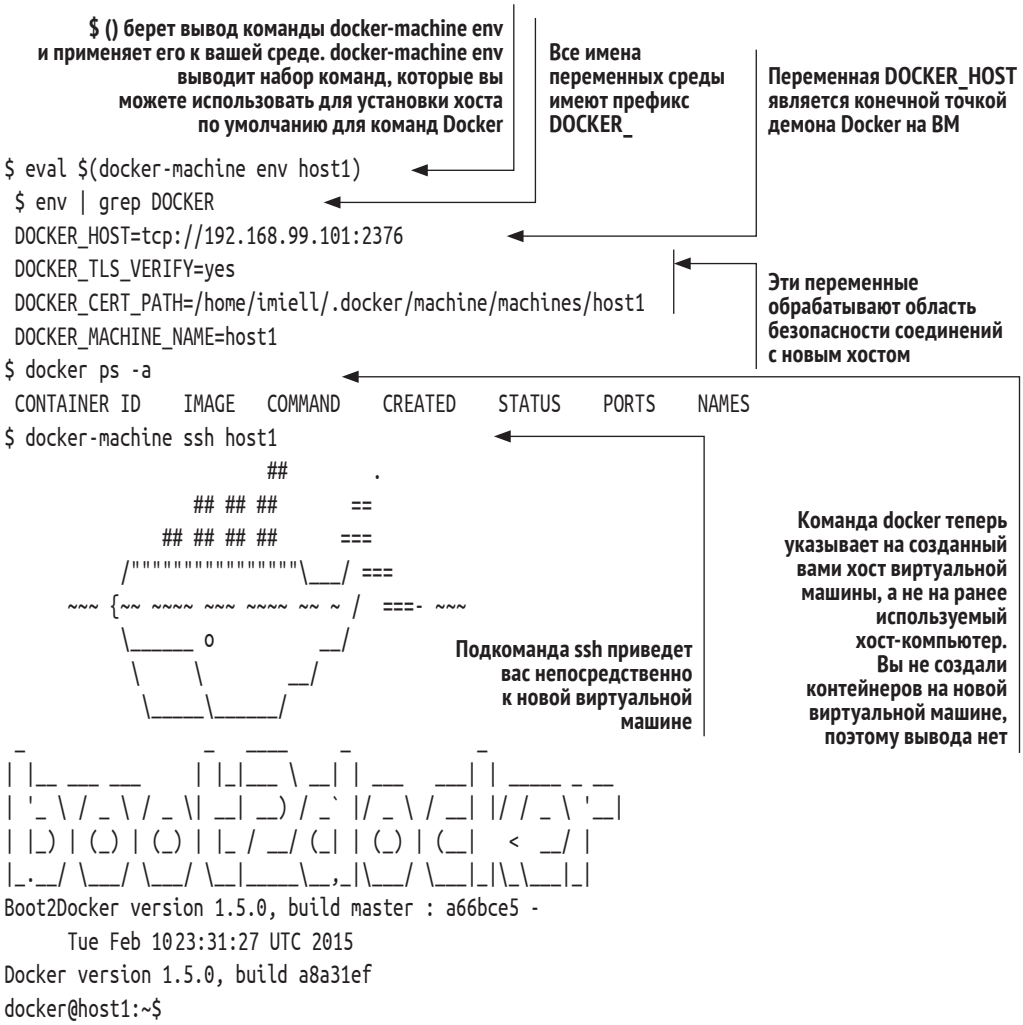
Используйте подкоманду docker-machine для создания нового хоста и укажите его тип, используя флаг --driver. Хост получил имя host1

```
$ docker-machine create --driver virtualbox host1
INFO[0000] Creating CA: /home/imiell/.docker/machine/certs/ca.pem
INFO[0000] Creating client certificate:
➥ /home/imiell/.docker/machine/certs/cert.pem
INFO[0002] Downloading boot2docker.iso to /home/imiell/.docker/machine/cache/
➥ boot2docker.iso...
INFO[0011] Creating VirtualBox VM...
INFO[0023] Starting VirtualBox VM...
INFO[0025] Waiting for VM to start...
INFO[0043] "host1" has been created and is now the active machine.
INFO[0043] To point your Docker client at it, run this in your shell:
➥ $(docker-machine env host1)
```

Ваш компьютер создан

Запустите эту команду, чтобы установить переменную окружения DOCKER_HOST, которая устанавливает хост по умолчанию, где будут выполняться команды Docker

Пользователи Vagrant будут чувствовать себя здесь как дома. Выполнив эти команды, вы создали компьютер, на котором теперь можете управлять Docker. Если следовать инструкциям, приведенным в выходных данных, можно напрямую подключиться к новой виртуальной машине по SSH:



Управление несколькими хостами

Управление несколькими хостами Docker с одного клиентского компьютера может затруднить отслеживание происходящего. Docker Machine поставляется с различными командами управления, чтобы сделать этот процесс проще, как показано в табл. 5.2.

Таблица 5.2 ❖ List of docker-machine commands

Подкоманда	Действие
`create`	Создает новый компьютер
`ls`	Перечисляет хост-компьютеры Docker
`stop`	Останавливает компьютер
`start`	Запускает компьютер
`restart`	Останавливает и запускает компьютер
`rm`	Уничтожает компьютер
`kill`	Убивает компьютер
`inspect`	Возвращает представление метаданных компьютера в формате JSON
`config`	Возвращает конфигурацию, необходимую для подключения к компьютеру
`ip`	Возвращает IP-адрес компьютера
`url`	Возвращает URL-адрес демона Docker на компьютере
`upgrade`	Обновляет версию Docker на хосте до последней версии

В следующем примере перечислено два компьютера. Активный компьютер отмечен звездочкой, и у него есть связанное с ним состояние, аналогичное состоянию контейнеров или процессов:

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM
host1            virtualbox   Running   tcp://192.168.99.102:2376
host2   *        virtualbox   Running   tcp://192.168.99.103:2376
```

> **СОВЕТ.** Вам может быть интересно, как переключиться на исходный экземпляр хост-компьютера. На момент написания этих строк мы не нашли простого способа сделать это. Вы можете уничтожить все компьютеры с помощью команды `docker machine rm` или, если этот вариант вам не подходит, можете вручную сбросить переменные среды, ранее установленные с помощью `unset DOCKER_HOST DOCKER_TLS_VERIFY DOCKER_CERT_PATH`.

ОБСУЖДЕНИЕ

Можно смотреть на это как на превращение машин в процессы, аналогично и сам Docker рассматривают как превращение среды в процесс.

Заманчиво использовать настройку Docker Machine для ручного управления контейнерами на нескольких хостах, но если вы обнаружите, что удаляете контейнеры вручную, снова собираете их и запускаете при изменениях кода, мы рекомендуем вам взглянуть на четвертую часть этой книги. Утомительные задачи, подобные этой, могут быть выполнены компьютерами на отлично. В методе 87 приводится официальное решение от Docker Inc. для создания автоматического кластера контейнеров. Метод 84 может быть привлекателен, если вам нравится идея унифицированного представления кластера, но вы также предпочитаете сохранять полный контроль над тем, где ваши контейнеры в конечном итоге будут работать.

МЕТОД 33 Запись Wildcard

При использовании Docker очень часто работает много контейнеров, которые должны обращаться к центральной или внешней службе. При тестировании или разработке таких систем для этих служб обычно используется статический IP-адрес. Но для многих систем на базе Docker, таких как OpenShift, IP-адреса недостаточно. Подобные приложения требуют поиска DNS.

Обычное решение в этой ситуации – отредактировать файл /etc/hosts на хостах, где вы запускаете свои службы. Но это не всегда возможно. Например, у вас может не быть доступа для редактирования файла. И это не всегда практично. У вас может быть слишком много хостов, которые нужно обслуживать, или могут мешать другие специальные кеши поиска DNS.

В этих случаях есть решение, которое использует «настоящие» DNS-серверы.

ПРОБЛЕМА

Вам нужен преобразуемый в DNS URL-адрес для определенного IP-адреса.

РЕШЕНИЕ

Используйте веб-сервис NIP.IO для преобразования IP-адреса в URL-адрес без какой-либо настройки DNS.

Это действительно просто. NIP.IO – это веб-сервис, который автоматически превращает IP-адрес в URL-адрес. Вам просто нужно заменить раздел «IP» URL-адреса http://IP.nip.io на желаемый IP-адрес.

Допустим, IP-адрес, который вы хотите преобразовать в URL-адрес, – «10.0.0.1». Ваш URL может выглядеть так:

```
http://myappname.10.0.0.1.nip.io
```

где `myappname` – это предпочтительное имя вашего приложения, `10.0.0.1` обозначает IP-адрес, который вы хотите преобразовать, а `nip.io` – это «реальный» домен в интернете, управляющий этой службой поиска DNS.

`myappname` является необязательным, поэтому этот URL-адрес будет преобразован в тот же IP-адрес:

```
http://10.0.0.1.nip.io
```

Этот метод удобен во всех контекстах, а не только при использовании служб на основе Docker.

Должно быть очевидно, что этот метод не подходит для рабочей или надлежащей среды UAT, поскольку он отправляет DNS-запросы третьей стороне и раскрывает информацию о вашей внутренней структуре IP-адресов. Но это может быть очень удобным инструментом для разработки.

Используя эту службу с HTTPS, убедитесь, что URL-адрес (или подходящая запись wildcard) указан в сертификате, который вы применяете.

5.2. Тома

Контейнеры – это мощная концепция, но иногда не все, к чему вы хотите получить доступ, готово к инкапсуляции. Это может быть эталонная база данных Oracle, хранящаяся в большом кластере, к которому вы хотите подключиться для тестирования. Или, может быть, у вас есть большой устаревший сервер, уже настроенный с помощью двоичных файлов, которые нельзя с легкостью воспроизвести.

Когда вы начнете работать с Docker, большинство вещей, к которым нужно получить доступ, скорее всего, будут данными и программами, внешними по отношению к вашему контейнеру. Мы проведем вас от простого монтирования файлов из хоста до более сложных шаблонов контейнеров: контейнер данных и контейнер инструментов разработчика. Мы также продемонстрируем своего прагматичного любимца для дистанционного монтирования по сети, для работы которого требуется только SSH-соединение, и рассмотрим способы совместного использования данных по протоколу BitTorrent.

Тома являются основной частью Docker, а проблема внешних ссылок на данные является еще одной быстро меняющейся областью экосистемы Docker.

МЕТОД 34 Тома Docker: проблемы персистентности

Большая часть возможностей контейнеров связана с тем фактом, что они инкапсулируют такое количество состояния файловой системы среды, сколько полезно.

Однако иногда вам не хочется помещать файлы в контейнер. Это могут быть большие файлы, которые нужно использовать совместно в контейнерах или управлять ими по отдельности. Классическим примером является большая централизованная база данных, и вы хотите, чтобы ваш контейнер имел к ней доступ, но чтобы и другие (возможно, более традиционные) клиенты имели доступ наряду с вашими новомодными контейнерами.

Решением в этом случае являются *тома*, механизм Docker для управления файлами вне жизненного цикла контейнера. Хотя это идет вразрез

с философией «развертывания контейнеров в любом месте» (вы не сможете развернуть свой контейнер, зависящий от базы данных, где нет, например, совместимой базы данных, доступной для монтирования), это полезная функция для реального использования Docker.

ПРОБЛЕМА

Вы хотите получить доступ к файлам на хосте из контейнера.

РЕШЕНИЕ

Используйте флаг тома Docker для доступа к файлам хоста в рамках контейнера. Рисунок 5.3 иллюстрирует использование флага тома для взаимодействия с файловой системой хоста.

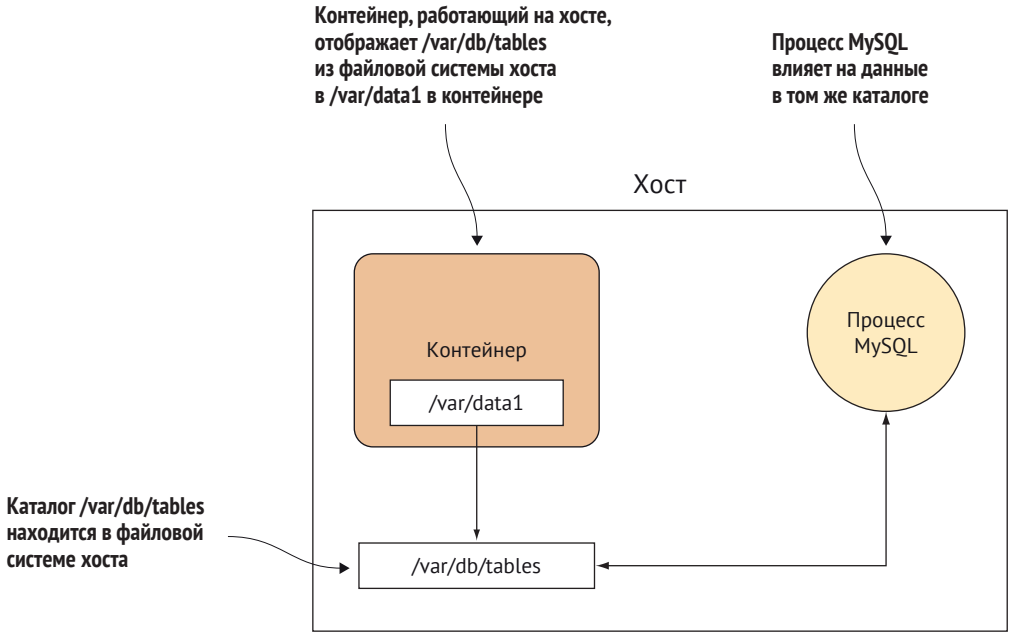


Рис. 5.3 ❖ Том внутри контейнера

Следующая команда показывает каталог хоста /var/db/tables, который монтируется в /var/data1, и его можно открыть, чтобы запустить контейнер на рис. 5.3.

```
$ docker run -v /var/db/tables:/var/data1 -it debian bash
```

Флаг `-v` указывает на то, что требуется том, внешний по отношению к контейнеру. Последующий аргумент задает спецификацию тома в виде двух каталогов, разделенных двоеточием, что указывает Docker отображать внешний каталог /var/db/tables в каталог контейнера /var/data1. Как внешние каталоги, так и каталоги контейнера будут созданы, если они не существуют.

Остерегайтесь отображения поверх существующих каталогов. Каталог контейнера будет отображен, даже если он уже существует в образе. Это означает, что каталог, который вы отображаете в контейнер, фактически исчезнет. Забавные вещи случаются при попытке отобразить каталог ключей! Попробуйте смонтировать пустой каталог поверх /bin, например.

Также обратите внимание: предполагается, что тома не сохраняются в файлах Dockerfile. Если вы добавите том, а затем внесете изменения в эту папку в Dockerfile, изменения не будут сохранены в результирующем образе.

> **ВНИМАНИЕ!** Вы можете столкнуться с трудностями, если на хосте работает SELinux. При применении SELinux контейнер может не иметь возможности записи в каталог /var/db/tables. Вы увидите ошибку «Отказано в доступе». Если вам нужно обойти это, необходимо будет поговорить со своим системным администратором (при наличии оного) или отключить SELinux (только с целью разработки). Смотрите метод 113 для получения дополнительной информации о SELinux.

ОБСУЖДЕНИЕ

Обеспечение доступа к файлам из хоста в контейнере является одной из наиболее распространенных операций, которые мы выполняем при экспериментировании с отдельными контейнерами, – контейнеры должны быть эфемерными, и слишком легко отбросить один из них, потратив значительное количество времени, работая над некоторыми файлами в одном. Лучше быть уверенным, что файлы в безопасности, что бы ни случилось.

Здесь также есть преимущество, заключающееся в том, что обычные накладные расходы на копирование файлов в контейнеры способом, описанным в методе 114, просто отсутствуют. Базы данных, подобные той, что приведена в методе 77, являются очевидным бенефициаром, если они становятся большими.

Наконец, вы увидите несколько методов, которые используют `-v/var/run/docker.sock:/var/run/docker.sock`, один из них – метод 45. Это обеспечивает доступ к специальному файлу сокета Unix для контейнера и демонстрирует важную способность данного метода: вы не ограничены так называемыми обычными файлами, вы также можете разрешить более необычные варианты использования на основе файловой системы. Но если у вас возникают проблемы с правами доступа на узлах устройства (например), вам, возможно, придется обратиться к методу 93, чтобы получить представление о том, что делает флаг `-privileged`.

МЕТОД 35 Распределенные тома и Resilio Sync

При экспериментировании с Docker в команде вы, возможно, захотите обмениваться большими объемами данных между членами команды, но у вас может не быть ресурсов для общего сервера с достаточной емкостью. Ленивое

решение этой проблемы – копирование последних файлов от других членов команды, когда они вам нужны; это быстро выходит из-под контроля при работе в большой группе.

Решение состоит в том, чтобы использовать децентрализованный инструмент для совместного использования файлов – выделенный ресурс не требуется.

ПРОБЛЕМА

Вы хотите совместно использовать тома между хостами через интернет.

РЕШЕНИЕ

Примените технологию Resilio, чтобы совместно использовать тома через интернет.

На рис. 5.4 показана установка, к которой вы стремитесь.

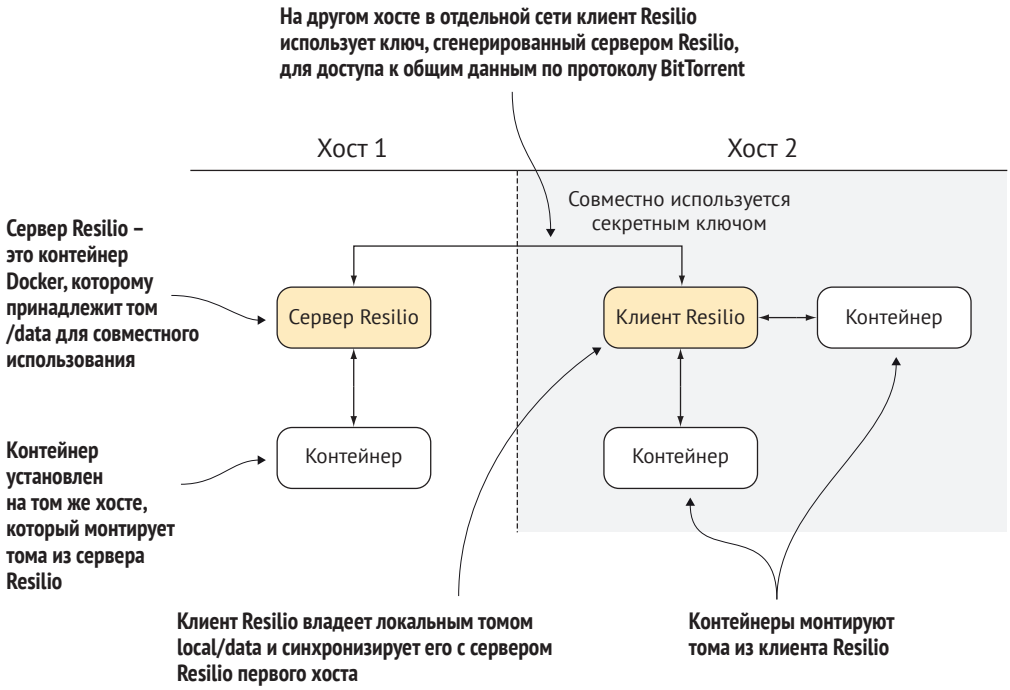
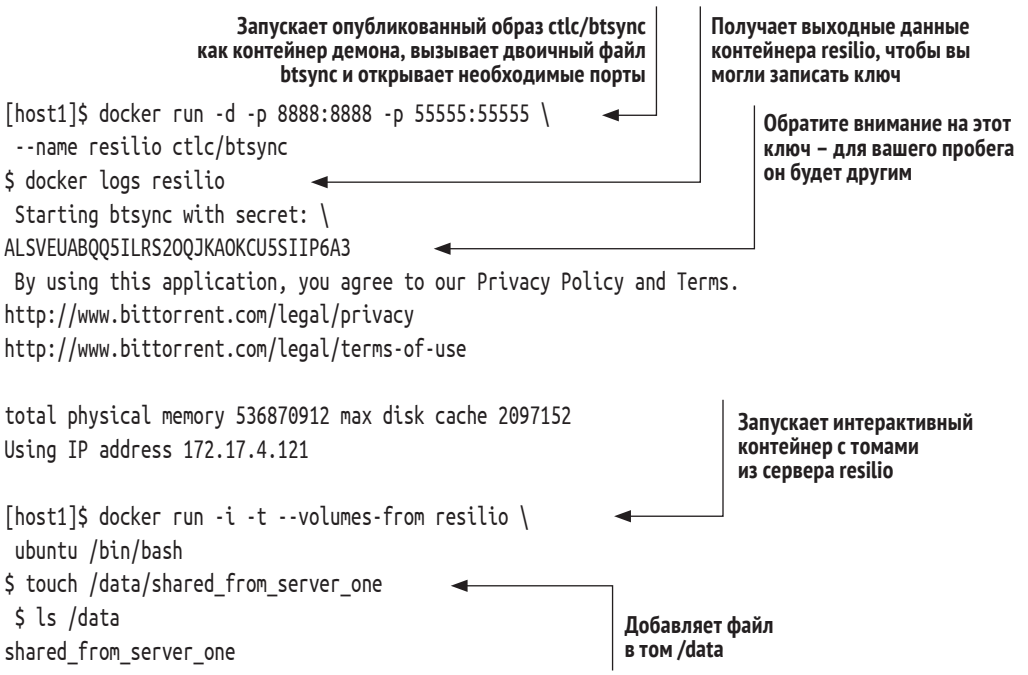


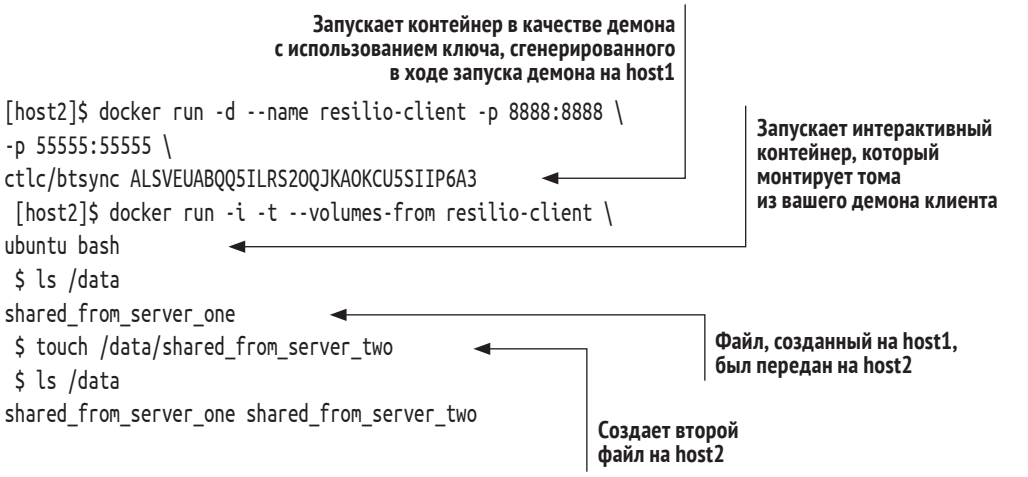
Рис. 5.4 ❖ Использование Resilio

Конечным результатом является том (/data), который удобно синхронизируется через интернет без какой-либо сложной настройки.

На основном сервере выполните следующие команды, чтобы настроить контейнеры на первом хосте:



На втором сервере откройте терминал и выполните следующие команды для синхронизации тома:



Вернувшись к работающему контейнеру host1, вы должны увидеть, что файл синхронизирован между хостами так же, как и первый файл:

```
[host1]$ ls /data
shared_from_server_one shared_from_server_two
```

ОБСУЖДЕНИЕ

Синхронизация файлов происходит без каких-либо гарантий, поэтому вам, возможно, придется подождать, пока данные синхронизируются. Особенно это касается больших файлов.

> **ПРЕДУПРЕЖДЕНИЕ.** Поскольку данные могут отправляться через интернет и обрабатываются по протоколу, который не контролируется вами, не стоит полагаться на этот метод при наличии каких-либо существенных ограничений безопасности, масштабируемости или производительности.

Мы только продемонстрировали, что этот метод работает между двумя контейнерами, как уже упоминалось в начале, но он также должен работать со многими членами команды. Помимо очевидного варианта использования больших файлов, которые не подходят для управления версиями, кандидаты на распространение включают в себя резервные копии и, возможно, сами образы Docker, особенно если этот метод используется в сочетании с эффективным механизмом сжатия, подобным тому, что был продемонстрирован в методе 72. Во избежание конфликтов убедитесь, что образы всегда идут в одном направлении (например, с компьютера сборки на множество серверов) или следуйте согласованному процессу обновления.

МЕТОД 36 Сохранение истории bash вашего контейнера

Экспериментирование внутри контейнера, знание того, что вы можете все уничтожить, когда закончите, может быть освобождающим опытом. Но при этом вы теряете ряд преимуществ. Одно из них, с которым мы много раз сталкивались, – это забывание последовательности команд, выполняемых внутри контейнера.

ПРОБЛЕМА

Вы хотите поделиться историей bash вашего контейнера с историей вашего хоста.

РЕШЕНИЕ

Используйте флаг `-e`, монтирование Docker и псевдоним bash, чтобы автоматически делиться историей bash вашего контейнера с историей хоста.

Чтобы понять эту проблему, мы покажем вам простой сценарий, когда потеря этой истории просто раздражает.

Представьте, что вы экспериментируете с контейнерами Docker, и в разгар работы делаете что-то интересное и многоразовое. Для этого примера мы будем использовать простую команду `echo`, но конкатенация программ, которая приводит к полезному выводу, может быть длинной и сложной:

```
$ docker run -ti --rm ubuntu /bin/bash
$ echo my amazing command
```

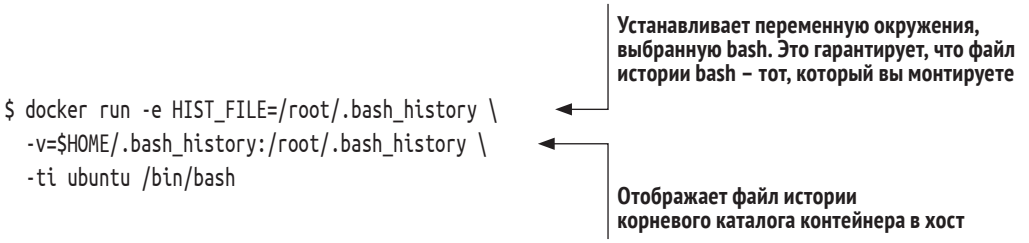
```
$ exit
```

Через некоторое время требуется снова вызвать невероятную команду echo, запущенную ранее. К сожалению, вы не можете вспомнить это, и больше нет терминальной сессии на экране для прокрутки. По привычке вы пытаетесь просмотреть историю bash на хосте:

```
$ history | grep amazing
```

Ничего не возвращается, потому что история bash хранится в удаленном теперь контейнере, а не на хосте, к которому вы вернулись.

Чтобы поделиться историей bash с хостом, вы можете использовать монтирование тома при запуске ваших образов Docker. Вот пример:



> **СОВЕТ.** Возможно, вы захотите отделить bash историю контейнеров от истории своего хоста. Один из способов сделать это – изменить значение первой части предыдущего аргумента -v.

Вводить это каждый раз – сущее наказание, поэтому для удобства пользователей можно настроить псевдоним, поместив его в файл ~ / .bashrc:

```
$ alias dockbash='docker run -e HIST_FILE=/root/.bash_history \
 -v=$HOME/.bash_history:/root/.bash_history
```

По-прежнему непросто, потому что нужно не забывать набирать dockbash, если вы хотите выполнить команду docker run. Чтоб было удобнее, можно добавить в файл ~ / .bashrc это:

Листинг 5.6. Псевдоним функции для автоматического монтирования истории bash

```
      -e HIST_FILE=/root/.bash_history \
      -v $HOME/.bash_history:/root/.bash_history "$@"
  else
    /usr/bin/docker "$@"
  fi
}
alias docker=basher
```

Передает аргументы после «run» в среду выполнения Docker

Выполняет команду docker с исходными аргументами без изменений

Устанавливает псевдоним для команды docker, когда она вызывается в командной строке для созданной вами функции basher. Это гарантирует, что вызов docker будет перехвачен до того, как bash найдет двоичный файл docker в пути

ОБСУЖДЕНИЕ

Теперь, когда вы в следующий раз откроете оболочку bash и выполните любую команду `docker run`, команды, запускаемые в этом контейнере, будут добавлены в историю bash вашего хоста.

Убедитесь, что путь к Docker указан правильно. Например, он может быть расположен в /bin/docker.

> **ПРИМЕЧАНИЕ.** Вам необходимо выйти из исходного сеанса bash хоста, чтобы файл истории обновился. Это связано с особенностью bash и тем, как она обновляет историю bash, хранящуюся в памяти. Если вы сомневаетесь, выйдите из всех известных вам сеансов bash, а затем запустите один из них, чтобы убедиться, что ваша история актуальна.

Ряд инструментов командной строки с приглашениями также хранят историю, один из примеров – SQLite (он хранит историю в файле .sqlite_history). Если вы не хотите использовать интегрированные решения для ведения журналов, доступные в Docker, описанные в методе 102, можете использовать аналогичную методику, чтобы приложение выполняло запись в файл, который выходит за пределы контейнера. Имейте в виду, что сложности ведения журналов, такие как ротация, означают, что будет проще использовать том каталога журналов, а не просто файл.

МЕТОД 37 Контейнеры данных

Если вы много используете тома на хосте, управлять запуском контейнера может быть сложно. Вы также можете захотеть, чтобы данные управлялись исключительно Docker и не были общедоступными на хосте. Один из способов управлять этим более четко – использовать шаблон проектирования data-only container.

ПРОБЛЕМА

Вы хотите использовать внешний том внутри контейнера, но нужно, чтобы только Docker имел доступ к файлам.

РЕШЕНИЕ

Запустите контейнер данных и используйте флаг `--volumes-from` при запуске других контейнеров.

На рис. 5.5 показана структура шаблона «Контейнер данных» и объясняется, как он работает. Ключевым моментом, который стоит отметить, является то, что на втором хосте контейнерам не нужно знать, где находятся данные на диске. Все, что им нужно знать, – это имя контейнера данных, и они готовы к работе. Это может сделать работу контейнеров более переносимой.

Еще одним преимуществом такого подхода по сравнению с прямым отображением каталогов хоста является то, что доступ к данным файлам управляется Docker, а это означает, что процесс, не связанный с Docker, с меньшей вероятностью повлияет на содержимое.

> **ПРИМЕЧАНИЕ.** Люди обычно не понимают, нужно ли запускать контейнер только для данных. Не нужно! Он просто должен существовать, быть запущенным на хосте, и его незачем удалять.

Давайте рассмотрим простой пример, чтобы вы могли понять, как использовать этот метод.

Сначала запускаете свой контейнер данных:

```
$ docker run -v /shared-data --name dc busybox \
  touch /shared-data/somefile
```

Аргумент `-v` не отображает том в каталог хоста, поэтому создает каталог в рамках ответственности этого контейнера. Этот каталог заполняется одним файлом с помощью `touch`, и контейнер уже есть – контейнер данных не нужно запускать, чтобы использовать. Мы взяли небольшой, но функциональный образ busybox, чтобы уменьшить количество дополнительного багажа, необходимого нашему контейнеру данных.

Затем запускаете еще один контейнер, чтобы получить доступ к только что созданному вами файлу:

```
docker run -t -i --volumes-from dc busybox /bin/sh
/ # ls /shared-data
Somefile
```

ОБСУЖДЕНИЕ

Флаг `--volumes-from` позволяет вам обращаться к файлам из контейнера данных, монтируя их в текущем контейнере, – вам просто нужно передать ему идентификатор контейнера с определенными томами. В образе busybox отсутствует bash, поэтому нужно запустить более простую оболочку, чтобы убедиться, что вам доступна папка /shared-data из контейнера dc.

Можно запустить любое количество контейнеров, выполняющих чтение и запись в указанные тома контейнера данных.

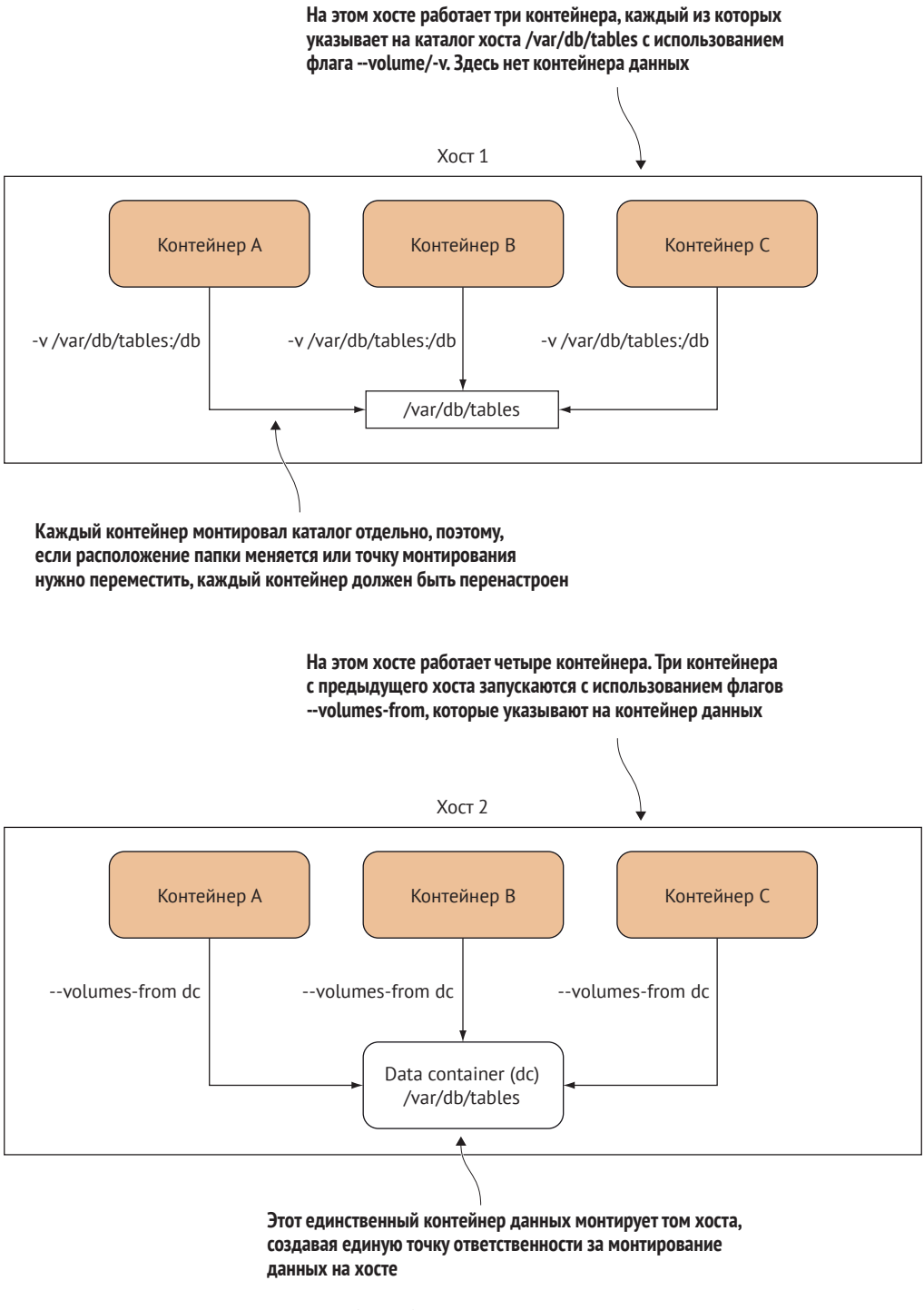


Рис. 5.5 ❖ Шаблон «Контейнер данных»

Вам не нужно использовать этот шаблон для использования томов – может оказаться сложнее управлять этим подходом, чем простым монтированием каталога хоста. Однако, если вы хотите четко делегировать ответственность за управление данными в одной точке, управляемой в Docker и не подверженной другими хост-процессами, контейнеры данных могут быть вам полезны.

> **ПРЕДУПРЕЖДЕНИЕ.** Если ваше приложение записывает логи из нескольких контейнеров в один и тот же контейнер данных, важно убедиться, что каждый файл журнала контейнера делает запись в уникальный путь к файлу. Если этого не сделать, разные контейнеры могут перезаписать или усечь файл, что приведет к потере данных, или они могут записать чередуемые данные, которые труднее анализировать. Точно так же, если вы вызываете `--volumes-from` из контейнера данных, вы позволяете этому контейнеру потенциально накладывать каталоги поверх ваших, так что будьте осторожны с конфликтами имен.

Важно понимать, что этот шаблон может привести к интенсивному использованию диска, что может быть относительно сложным для отладки. Поскольку Docker управляет томом в контейнере только для данных и не удаляет том при завершении работы последнего контейнера, который обращается к нему, все данные на томе будут сохранены. Это должно предотвратить нежелательную потерю данных. Чтобы получить совет как управлять этим, см. метод 43.

МЕТОД 38 Удаленное монтирование тома с использованием SSHFS

Мы обсуждали монтирование локальных файлов, но вскоре возникает вопрос о том, как монтировать удаленные файловые системы. Возможно, вы хотите совместно использовать справочную базу данных на удаленном сервере и рассматривать ее, например, как локальную.

Хотя теоретически можно настроить NFS на вашей хост-системе и на сервере, а затем получить доступ к файловой системе, смонтировав этот каталог, для большинства пользователей существует более быстрый и простой способ, не требующий настройки на стороне сервера (при условии наличия SSH-доступа).

> **ПРИМЕЧАНИЕ.** Чтобы метод сработал, вам потребуются привилегии суперпользователя, а также необходимо будет установить FUSE (модуль ядра Linux «Filesystem in Userspace»). Вы можете определить, есть ли он у вас, запустив в терминале `ls /dev/fuse`.

ПРОБЛЕМА

Вы хотите смонтировать удаленную файловую систему без необходимости настройки на стороне сервера.

РЕШЕНИЕ

Используйте технологию под названием SSHFS для монтирования удаленной файловой системы, чтобы она была локальной для вашего компьютера.

Этот метод работает с использованием модуля ядра FUSE с SSH, чтобы обеспечить вам стандартный интерфейс для файловой системы, в то время как в фоновом режиме все коммуникации идут через SSH. SSHFS также предоставляет различные закулисные функции (такие как удаленное чтение файлов), чтобы создать иллюзию, будто файлы являются локальными. В результате после входа пользователя на удаленный сервер они будут видеть файлы, как если бы они были локальными. Рисунок 5.6 помогает объяснить это.

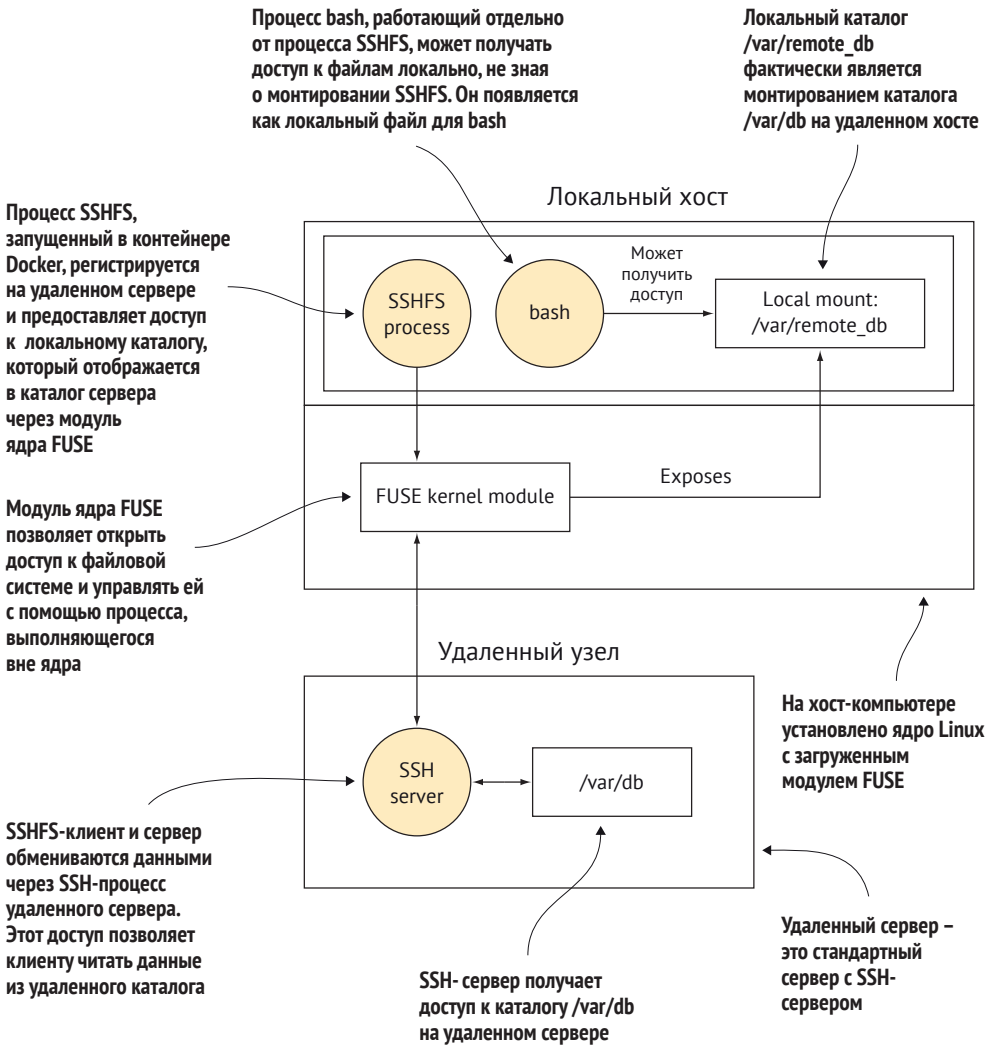


Рис. 5.6 ❖ Монтирование удаленной файловой системы с помощью SSHFS

> **ВНИМАНИЕ!** Хотя этот метод не использует функциональность томов Docker, а файлы видны через файловую систему, он не дает постоянства на уровне контейнеров. Все изменения происходят только в файловой системе удаленного сервера.

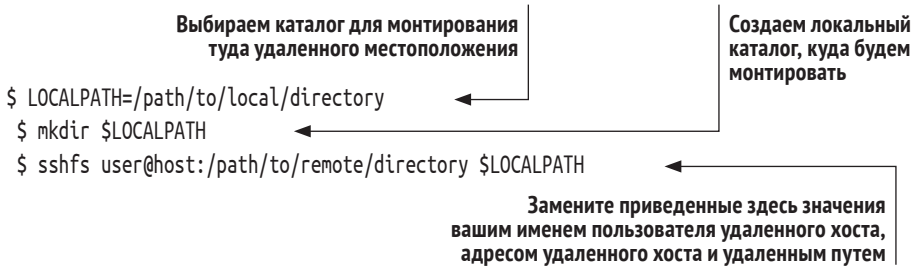
Вы можете начать с выполнения следующих команд, адаптированных к вашей среде.

Первый шаг – запустить контейнер, используя `--privileged` на своем хост-компьютере:

```
$ docker run -t -i --privileged debian /bin/bash
```

Затем, после запуска, выполните команду `apt-get update && apt-get install sshfs` из контейнера, чтобы установить SSHFS.

После успешной установки SSHFS зайдите на удаленный хост:



Теперь вы увидите содержимое пути на удаленном сервере в папке, которую только что создали.

> **СОВЕТ.** Проще всего монтировать в каталог, который вы только что создали, но также можно монтировать уже существующий каталог с имеющимися в наличии файлами, если вы используете опцию `-o nonempty`.. Смотрите справочную страницу SSHFS для получения дополнительной информации.

Чтобы гладко демонтировать файлы, используйте команду `fusermount`, заменяя путь:

```
fusermount -u /path/to/local/directory
```

ОБСУЖДЕНИЕ

Это отличный способ быстро настроить удаленное монтирование из контейнеров (и на стандартных компьютерах с Linux) с минимальными усилиями.

Хотя в этом методе речь шла только о SSHFS, успешное управление ею открывает замечательный (а иногда и странный) мир файловых систем FUSE внутри Docker. Вам предоставляется целый ряд возможностей: от хранения данных в Gmail до распределенной файловой системы GlusterFS для хранения петабайтов данных на множестве компьютеров.

МЕТОД 39 Совместное использование данных через NFS

В более крупной компании общие каталоги NFS, вероятно, уже будут использоваться – NFS – это хорошо зарекомендовавший себя вариант передачи файлов из центрального расположения. Для Docker, чтобы получить тягу, обычно довольно важно иметь возможность получить доступ к этим общим файлам.

Docker не поддерживает NFS «из коробки», и установка NFS-клиента на каждый контейнер, чтобы вы могли монтировать удаленные папки, – не лучшая практика.

Вместо этого предлагается использовать один контейнер в качестве транслятора из NFS в более удобную для Docker концепцию: тома.

ПРОБЛЕМА

Вам нужен беспрепятственный доступ к удаленной файловой системе через NFS.

РЕШЕНИЕ

Используйте контейнер данных инфраструктуры для обеспечения доступа к вашей удаленной файловой системе NFS.

Этот метод основан на методе 37, в котором мы создали контейнер данных для управления данными в работающей системе.

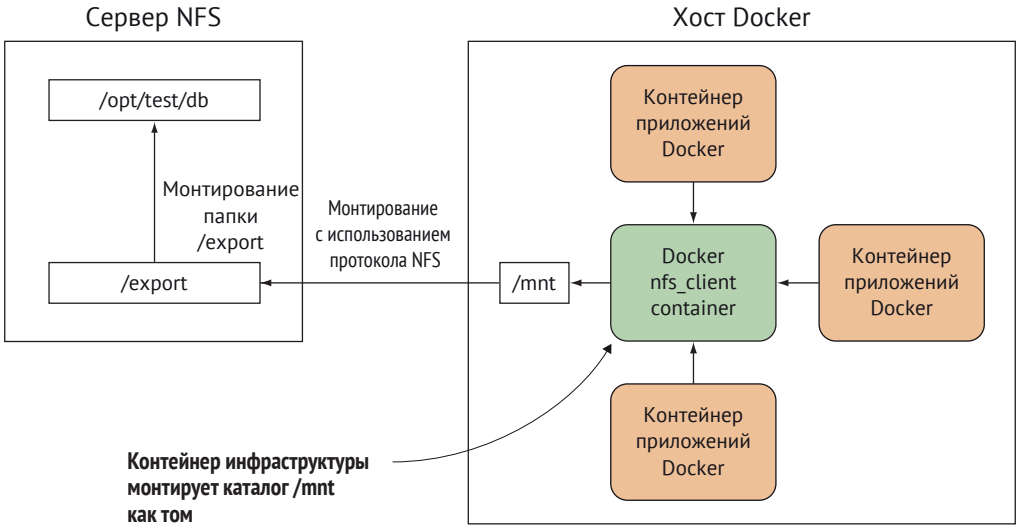


Рис. 5.7 ❖ Контейнер инфраструктуры, обеспечивающий доступ по NFS

На рис. 5.7 показана идея этого метода в абстрактном плане. Сервер NFS представляет внутренний каталог как папку /export, которая монтируется на хосте с применением опции --bind. Затем хост Docker монтирует эту папку,

используя протокол NFS, в свою папку /mnt. После этого создается так называемый контейнер инфраструктуры, который связывает папку mnt.

На первый взгляд это может показаться немного навороченным, но преимущество заключается в том, что этот способ обеспечивает уровень косвенности в отношении контейнеров Docker: все, что им нужно сделать, – это смонтировать тома из предварительно согласованного контейнера инфраструктуры, а ответственный за инфраструктуру может беспокоиться о внутреннем устройстве, доступности, сети и т. д.

Подробное рассмотрение NFS выходит за рамки этой книги. В данном методе мы просто выполним шаги по настройке такого общего ресурса на одном хосте, располагая компоненты сервера NFS на том же хосте, что и контейнеры Docker. Это было проверено на Ubuntu 14.04.

Предположим, вы хотите совместно использовать содержимое папки вашего хоста /opt/test/db, в которой находится файл mybigdb.db.

В качестве пользователя root установите сервер NFS и создайте каталог export с открытыми правами доступа:

```
# apt-get install nfs-kernel-server
# mkdir /export
# chmod 777 /export
```

> **ПРИМЕЧАНИЕ.** Мы создали общий ресурс NFS с открытыми правами доступа, что не является безопасным способом перехода к производственной системе. Мы выбрали этот подход в интересах упрощения данного учебного курса. Безопасность NFS – сложная и разнообразная тема, которая выходит за рамки книги. Подробнее о Docker и безопасности см. главу 14.

Теперь смонтируйте каталог db в каталог export, используя опцию `--bind`:

```
# mount --bind /opt/test/db /export
```

Теперь вы должны увидеть содержимое каталога /opt/test/db в /export:

> **СОВЕТ.** Если вы хотите, чтобы это сохранялось после перезагрузки, добавьте эту строку в файл /etc/fstab: `/opt/test/db /export none bind 0 0`

Теперь добавьте в файл /etc/exports строку:

```
/export        127.0.0.1(ro,fsid=0,insecure,no_subtree_check,async)
```

Для подтверждения концепции мы монтируем локально на `127.0.0.1`, что несколько противоречит цели. В реальном сценарии вы можете закрыть его для класса IP-адресов, таких как `192.168.1.0/24`. Если вам нравится играть с огнем, можете открыть его, используя * вместо `127.0.0.1`. В целях безопасности мы монтируем read-only (ro), но можно монтировать read-write, заменив `ro` на `rw`. Помните, что, сделав это, вам нужно будет добавить флаг `no_root_squash` после флага `async`, но подумайте о безопасности, прежде чем выходить из этой песочницы.

Смонтируйте каталог через NFS в каталог /mnt, экспортируйте файловые системы, которые вы указали ранее в /etc/exports, а затем перезапустите службу NFS, чтобы получить изменения:

```
# mount -t nfs 127.0.0.1:/export /mnt
# exportfs -a
# service nfs-kernel-server restart
```

Теперь вы готовы запустить контейнер инфраструктуры:

```
# docker run -ti --name nfs_client --privileged
➥ -v /mnt:/mnt busybox /bin/true
```

А теперь можно запустить – без привилегий или без знания базовой реализации – каталог, к которому хотите получить доступ:

```
# docker run -ti --volumes-from nfs_client debian /bin/bash
root@079d70f79d84:/# ls /mnt
myb
root@079d70f79d84:/# cd /mnt
root@079d70f79d84:/mnt# touch asd
touch: cannot touch `asd': Read-only file system
```

ОБСУЖДЕНИЕ

Шаблон централизованного монтирования общего ресурса с привилегированным доступом для использования другими в нескольких контейнерах – вещь довольная мощная. Он может значительно упростить рабочие процессы разработки.

> **СОВЕТ.** Если вам нужно управлять большим количеством контейнеров, можете упростить себе задачу, используя для контейнера, который предоставляет путь /opt/database/live, соглашение об именовании, такое как `--name nfs_client_opt_database_`.

> **СОВЕТ.** Помните, что этот метод обеспечивает безопасность только через неизвестность (а это вообще не безопасность). Как вы увидите позже, любой, кто может эффективно запустить исполняемый файл Docker, обладает привилегиями пользователя root на хосте.

Контейнеры инфраструктуры для обеспечения доступа и абстрагирования деталей в некотором смысле являются эквивалентом инструментов обнаружения служб для сетевой среды – точные детали того, как работает служба или где она обитает, не важны. Вам просто нужно знать ее имя.

Оказывается, прежде вы уже видели, как используется `--volumes-from`, в методе 35.

Детали немного отличаются, потому что доступ предоставляется инфраструктуре, работающей *внутри* контейнера, а не на хосте, но принцип использования имен для обращения к доступным томам остается. Вы могли бы даже

поменять этот контейнер на тот, что используется в этом методе и, если он настроен правильно, приложения не заметят разницы в том, где они ищут, чтобы получить свои файлы.

МЕТОД 40 Контейнер dev tools

Если вы инженер, который часто работает на чужих компьютерах, сражаясь, не имея программ или конфигурации, которые есть в вашей прекрасной среде разработки «уникальной, как снежинка», этот метод может быть вам полезен. Точно так же, если вы хотите поделиться своей средой разработки со всеми наворотами с другими, Docker легко может это сделать.

ПРОБЛЕМА

Вы хотите получить доступ к своей среде разработки на чужих компьютерах.

РЕШЕНИЕ

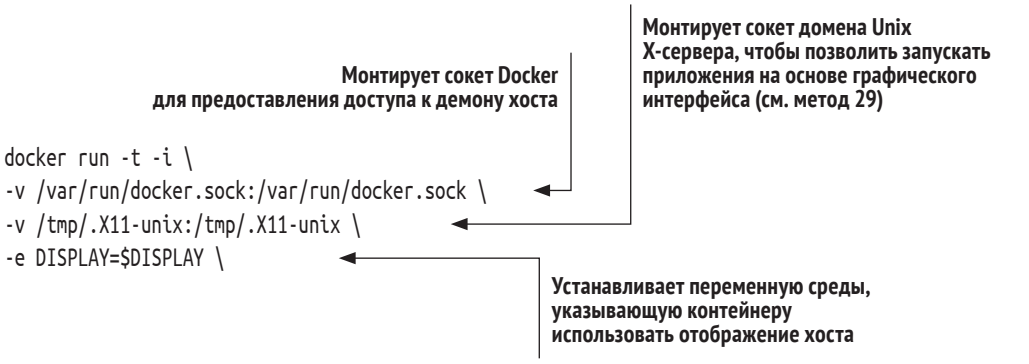
Создайте образ Docker со своими настройками и поместите его в реестр.

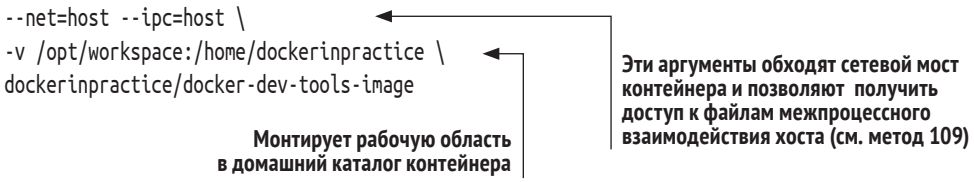
В качестве демонстрации мы будем использовать один из наших образов dev tools. Вы можете скачать его, выполнив команду `docker pull dockerinpractice/docker-dev-tools-image`. Репозиторий доступен по адресу https://github.com/docker-in-practice/docker-dev-tools-image, если вы хотите проверить Dockerfile.

Выполнить запуск контейнера просто – `docker run -t -i docker-inpractice/docker-dev-tools-image` предоставит вам оболочку в нашей среде разработки. Вы можете получить доступ к нашим точечным файлам и, возможно, отправить нам несколько советов относительно настройки.

Реальную силу этого метода можно увидеть, когда он сочетается с другими. В следующем листинге вы видите контейнер инструментов разработчика, используемый для отображения графического интерфейса пользователя в сети хоста и стеках IPDC, а также для монтирования кода хоста.

Листинг 5.7. Запуск образа dev-tools с графическим интерфейсом





Предыдущая команда дает вам среду с доступом к ресурсам хоста:

- сеть;
- демон Docker (для запуска обычных команд Docker, как на хосте);
- файлы межпроцессного взаимодействия (IPC);
- X-сервер для запуска приложений на основе графического интерфейса, если это необходимо.

> **ПРИМЕЧАНИЕ.** Как всегда при монтировании каталогов хоста, будьте осторожны, чтобы не смонтировать какие-либо важные каталоги, так как это может привести к повреждению. Как правило, лучше избегать монтирования любого каталога хоста в root.

ОБСУЖДЕНИЕ

Мы упомянули, что у вас есть доступ к X-серверу, поэтому стоит обратиться к методу 29, чтобы вспомнить о некоторых возможностях.

Для некоторых более инвазивных инструментов разработки, возможно, для проверки процессов на хосте, вам может понадобиться изучить метод 109, чтобы понять, как предоставить права на просмотр некоторых (по умолчанию) ограниченных частей вашей системы. Метод 93 также важен – просто потому, что если контейнер может видеть части вашей системы, это не обязательно означает, что у него есть полномочия на их изменение.

Резюме

- Следует обратиться к томам, если вам нужно получить внешние данные изнутри контейнера.
- SSHFS – это простой способ получить доступ к данным на других компьютерах без дополнительной настройки.
- Запуск приложений с графическим интерфейсом пользователя в Docker требует лишь небольшой подготовки вашего образа.
- Вы можете использовать контейнеры данных, чтобы абстрагироваться от местоположения ваших данных.

Глава 6

Повседневное использование Docker

О чем рассказывается в этой главе:

- управление контейнером и объемом пространства томов;
- отключение от контейнеров без их остановки;
- визуализация родословной вашего образа Docker на графе;
- запуск команд напрямую на ваших контейнерах из хоста.

Как и в любом программном проекте умеренной сложности, в Docker есть множество укромных уголков и закоулков, о которых важно знать, если вы хотите, чтобы ваш опыт был максимально плавным.

Методы, приведенные в этой главе, покажут вам наиболее важные из них, а также познакомят с внешними инструментами, созданными третьими лицами для удовлетворения собственных нужд. Думайте об этом как о своем наборе инструментов Docker.

6.1. Оставаться в полном порядке

Если вы чем-то похожи на нас (и если вы внимательно читаете эту книгу), ваша растущая зависимость от Docker будет означать, что вы запускаете множество контейнеров и скачиваете различные образы на выбранный вами хост.

Со временем Docker будет занимать все больше и больше ресурсов, и потребуются какие-то действия по обслуживанию контейнеров и томов. Мы покажем, как это происходит и почему. Мы также познакомим вас с визуальными инструментами для поддержания вашей среды Docker в чистоте и порядке на случай, если захочется выйти из командной строки.

Запуск контейнеров – дело очень хорошее, но вы довольно быстро обнаружите, что хотите сделать больше, чем просто запустить одну команду на переднем плане. Мы рассмотрим выход из запущенного контейнера без его уничтожения и выполнение команд внутри работающего контейнера.

МЕТОД 41 Запуск Docker без использования sudo

Демон Docker работает в фоновом режиме на вашем компьютере как пользователь root, предоставляя ему значительную мощность, которую он дает вам, пользователю. Результатом этого является необходимость использования sudo, но это может быть неудобно и сделает невозможным использование некоторых сторонних инструментов Docker.

ПРОБЛЕМА

Вы хотите иметь возможность запускать команду `docker` без использования `sudo`.

РЕШЕНИЕ

Официальное решение – добавить себя в группу `docker`.

Docker управляет правами доступа вокруг сокета домена Docker Unix через группу пользователей. По соображениям безопасности дистрибутивы не делают вас частью этой группы по умолчанию, поскольку она фактически предоставляет полный root-доступ к системе.

Добавив себя в эту группу, вы сможете использовать команду `docker` от своего имени:

```
$ sudo addgroup -a username docker
```

Перезапустите Docker, полностью выйдите и снова войдите в систему или перезагрузите компьютер, если так проще. Теперь вам не нужно набирать `sudo` или устанавливать псевдоним для запуска Docker от своего имени.

ОБСУЖДЕНИЕ

Это чрезвычайно важный метод для ряда инструментов, используемых далее в книге. В общем, любой, кто хочет общаться с Docker (без запуска в контейнере), будет нуждаться в доступе к сокету Docker, требуя `sudo` либо настройку, описанную в этом методе. Docker Compose, представленный в методе 76, официальный инструмент от Docker Inc., является примером такого инструмента.

МЕТОД 42 Содержание контейнеров в порядке

Новые пользователи Docker часто сталкиваются с тем, что за короткое время вы можете получить в своей системе множество контейнеров в различных состояниях, а стандартных инструментов для управления этим в командной строке не существует.

ПРОБЛЕМА

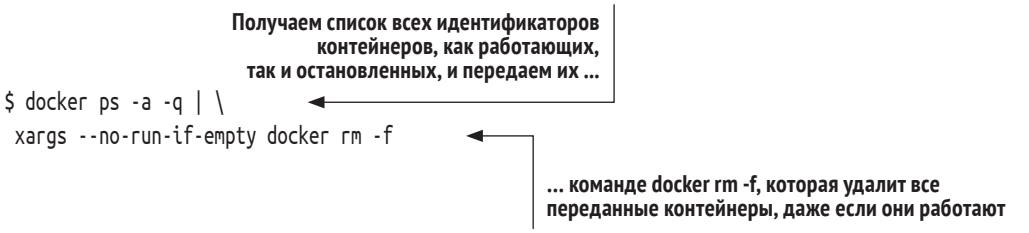
Вы хотите удалить контейнеры в своей системе.

РЕШЕНИЕ

Установите псевдонимы для запуска команд, которые убирают старые контейнеры.

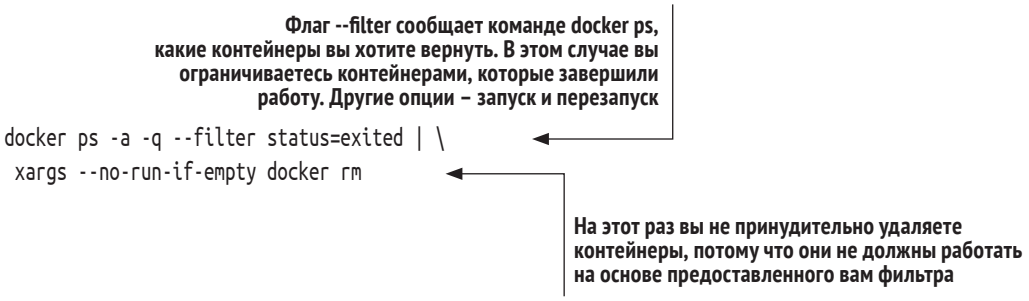
Самый простой подход – удалить все контейнеры. Очевидно, что это какой-то ядерный вариант, который следует использовать, только если вы уверены, что это то, что нужно.

Следующая команда удалит все контейнеры на вашем хост-компьютере.



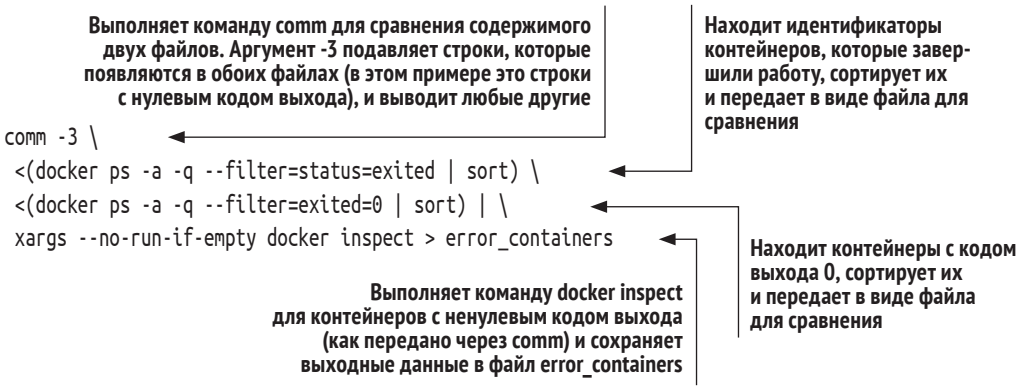
Если объяснять кратко, `xargs` берет каждую строку ввода и передает их все в качестве аргументов последующей команде. Мы передали здесь дополнительный аргумент `--no-run-if-empty`, который вообще не запускает команду, если не было вывода из предыдущей команды, чтобы избежать ошибки.

Если у вас есть запущенные контейнеры, которые вы, возможно, пожелаете сохранить, но хотите удалить все те, что завершили работу, можете отфильтровать элементы, возвращаемые командой `docker ps`:



Фактически удаление всех остановленных контейнеров является настолько распространенным вариантом использования, что Docker специально для этого добавил команду `docker container prune`.. Однако данная команда ограничена только этим вариантом использования, и вам потребуется обратиться к командам, приведенным в этом методе для более сложных манипуляций с контейнерами.

В качестве примера более сложного варианта использования следующая команда выведет список всех контейнеров с ненулевым кодом ошибки. Это может понадобиться, если в вашей системе много контейнеров и вы хотите автоматизировать проверку и удаление любого контейнера, который неожиданно завершил работу:



> **СОВЕТ.** Если вы раньше этого не видели, синтаксис `<(команда)` называется *заменой процесса*. Он позволяет рассматривать вывод команды как файл и передавать его другой команде, что может быть полезно, когда передача вывода по программному каналу невозможна.

Предыдущий пример довольно сложный, но он показывает силу, которую вы можете получить, комбинируя различные утилиты. Он выводит все идентификаторы остановленных контейнеров, а затем выбирает только те, которые имеют ненулевой код выхода (т. е. те, которые завершили работу неожиданным образом). Если вы изо всех сил стараетесь следовать этому, запуск каждой команды по отдельности и их разбор помогут в изучении строительных блоков.

Такая команда может быть полезна для сбора контейнерной информации о производстве. Возможно, вы захотите адаптировать ее для запуска cron, чтобы очистить контейнеры, которые завершили работу ожидаемым образом.

Сделать эти однострочники доступными в виде команд

Вы можете добавлять команды в качестве псевдонимов, чтобы их было легче выполнять при каждом входе на хост. Для этого добавьте в конец файла ~ / .bashrc строки:

```
alias dockernuke='docker ps -a -q | \
xargs --no-run-if-empty docker rm -f'
```

При следующем входе в систему при запуске `dockernuke` из командной строки будут удалены все контейнеры Docker, найденные в вашей системе.

Мы обнаружили, что это экономит удивительное количество времени. Но будьте осторожны! Слишком легко удалить производственные контейнеры таким образом, как мы можем засвидетельствовать. И даже если вы достаточно осторожны, чтобы не устранять работающие контейнеры, вы все равно можете удалить неработающие, но все еще полезные контейнеры только для данных.

ОБСУЖДЕНИЕ

Во многих методах, приведенных в этой книге, в конечном итоге создаются контейнеры, особенно при знакомстве с Docker Compose в методе 76 и в главах, посвященных оркестровке, – в конце концов, оркестровка связана с управлением несколькими контейнерами. Обсуждаемые здесь команды могут оказаться полезными для очистки ваших компьютеров (локальных или удаленных), чтобы начать все сначала по окончании каждого метода.

МЕТОД 43 Содержание томов в порядке

Хотя тома – это мощная функция Docker, у них есть существенный недостаток. Поскольку тома могут совместно использоваться различными контейнерами, их нельзя удалить при удалении контейнера, который их смонтировал. Представьте себе сценарий, показанный на рис. 6.1.

«Легко!» – подумаете вы. «Удалите том, когда удален последний обращающийся контейнер!» Действительно, Docker мог бы воспользоваться таким вариантом, и это тот подход, который используют языки программирования, применяющие механизм сборки мусора, когда они удаляют объекты из памяти: если ни один другой объект к нему не обращается, его можно удалить.

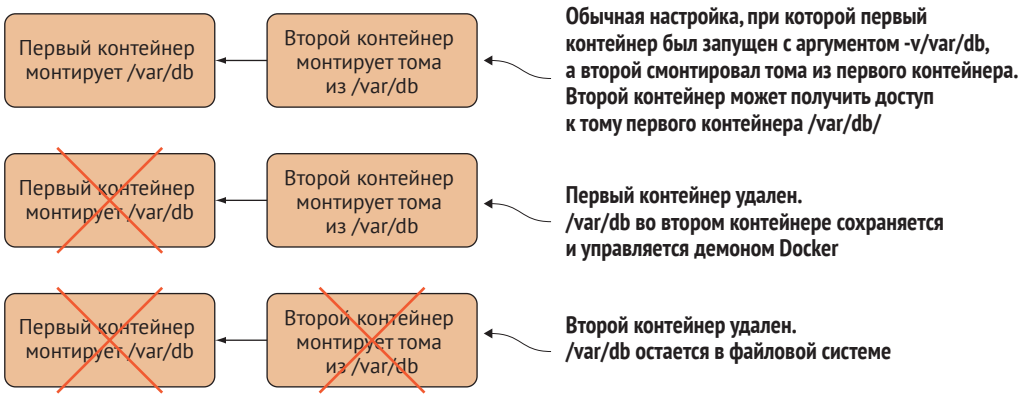


Рис. 6.1 ❖ Что происходит с /var/db при удалении контейнеров?

Но Docker посчитал, что это может оставить людей беззащитными перед случайной потерей ценных данных, и предпочел предоставить пользователю принимать решение о том, следует ли удалять том при удалении контейнера. К сожалению, побочным эффектом этого является то, что по умолчанию тома остаются на хост-диске демона Docker, до тех пор пока не будут удалены вручную. Если эти тома насыщены данными, ваш диск может заполниться, поэтому полезно знать о способах управления брошенными томами.

ПРОБЛЕМА

Вы используете слишком много дискового пространства, потому что на вашем хосте присутствуют брошенные точки монтирования.

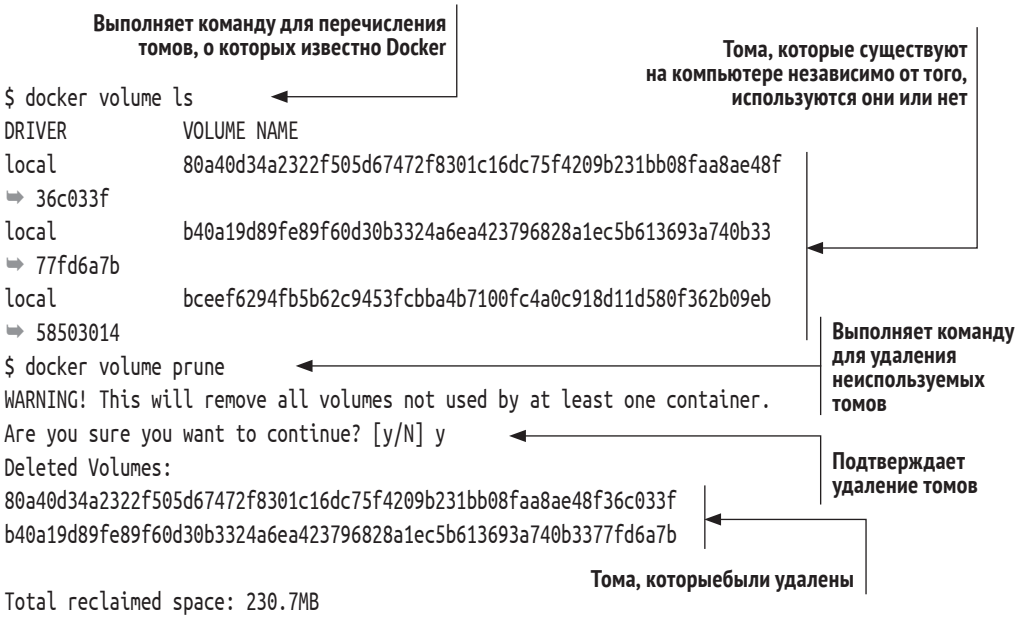
РЕШЕНИЕ

Используйте флаг `-v` при вызове команды `docker rm` или подкоманды `docker volume`, чтобы уничтожить их, если забудете.

В сценарии на рис. 6.1 вы можете убедиться, что /var/db удален, если вы всегда вызываете команду `docker rm`, используя флаг `-v`. Флаг `-v` удаляет все связанные тома, если ни один другой контейнер еще не смонтировал их. К счастью, Docker достаточно умен, чтобы знать, подключен ли какой-либо другой контейнер, поэтому никаких неприятных сюрпризов нет.

Самый простой подход состоит в том, чтобы привыкнуть вводить флаг `-v` всякий раз, когда вы удаляете контейнер. Таким образом, вы сохраняете контроль над удалением томов. Но проблема этого подхода заключается в том, что вы не всегда хотите удалять тома. Если вы записываете в них много данных, вполне вероятно, что не захочется потерять эти данные. Кроме того, если возникнет такая привычка, она, вероятно, станет автоматической, и вы поймете, что удалили что-то важное, только когда уже будет слишком поздно.

В этих сценариях можно использовать команду, которая была добавлена в Docker после долгих попыток и множества сторонних решений: `docker volume prune`. С ее помощью будут удалены все неиспользуемые тома:



Если вы хотите пропустить запрос на подтверждение, возможно, для автоматизированного сценария, можете передать `-f` команде `docker volume prune`.

> **СОВЕТ.** Если вы хотите восстановить данные с неудаленного тома, на который больше не ссылаются контейнеры, можете использовать команду `docker volume inspect`, чтобы обнаружить каталог, где находится том (вероятно, в /var/lib/docker/volume/), затем просмотреть его как пользователь root.

ОБСУЖДЕНИЕ

Удаление томов, вероятно, не то, что вам нужно будет делать очень часто, так как большие файлы в контейнере обычно монтируются с хост-компьютера и не хранятся в каталоге данных Docker. Но стоит проводить очистку каждую неделю или около того, чтобы избежать их накопления, особенно если вы используете контейнеры данных из метода 37.

МЕТОД 44 Отключение от контейнеров без их остановки

При работе с Docker вы часто будете оказываться в положении, когда есть интерактивная оболочка, но выход из нее приводит к прекращению работы контейнера, поскольку это основной процесс. К счастью, есть способ отключиться от контейнера (и, если хотите, можете использовать команду `docker attach` для повторного подключения к нему).

ПРОБЛЕМА

Вы хотите отключиться от контейнера, не останавливая его.

РЕШЕНИЕ

Используйте встроенную комбинацию клавиш в Docker для выхода из контейнера.

Docker тщательно реализовал последовательность клавиш, которая вряд ли понадобится любому другому приложению и которую также нельзя нажать случайно.

Допустим, вы запустили контейнер с помощью команды `docker run -t -i -p 9005:80 ubuntu /bin/bash`, а затем установили веб-сервер Nginx, используя `apt-get`. Вы хотите проверить, что он доступен с вашего хоста с помощью быстрой команды `curl` для `localhost:9005`.

Нажмите сочетание клавиш **Ctrl-P**, а затем **Ctrl-Q**. Обратите внимание, что не нужно нажимать все три клавиши одновременно.

> **ПРИМЕЧАНИЕ.** Если вы используете `--rm` и отключаетесь, контейнер все равно будет удален после завершения работы либо потому, что завершается команда, либо вы останавливаете ее вручную.

ОБСУЖДЕНИЕ

Этот метод полезен, если вы запустили контейнер, но, возможно, забыли запустить его в фоновом режиме, как показано в методе 2. Он также позволяет свободно подключаться к контейнерам и отключаться от них, если вы хотите проверить, как они работают или вносят какой-то вклад.

МЕТОД 45 Использование Portainer для управления демоном Docker

При демонстрации Docker может быть трудно показать, как различаются контейнеры и образы – строки в терминале не являются визуальными. Кроме того, инструменты командной строки Docker могут быть недружественными, если вы хотите завершить и удалить определенные контейнеры из числа многих. Эта проблема была решена с помощью инструмента «укажи и щелкни» для управления образами и контейнерами на хосте.

ПРОБЛЕМА

Вы хотите управлять контейнерами и образами на своем хосте без использования интерфейса командной строки.

РЕШЕНИЕ

Используйте Portainer – инструмент, созданный одним из основных участников Docker.

Portainer начинал свое существование как DockerUI, и вы можете прочитать о нем и найти исходный код на странице https://github.com/portainer/portainer. Поскольку предварительных условий нет, можно сразу перейти к его запуску:

```
$ docker run -d -p 9000:9000 \
-v /var/run/docker.sock:/var/run/docker.sock \
portainer/portainer -H unix:///var/run/docker.sock
```

Будет запущен контейнер Portainer в фоновом режиме. Если вы сейчас перейдете по адресу http://localhost:9000, то увидите панель инструментов, на которой кратко представлена информация для Docker на вашем компьютере.

Функция управления контейнерами, вероятно, является одной из наиболее полезных здесь: перейдите на страницу «Контейнеры», и вы увидите список ваших работающих контейнеров (включая Portainer) с возможностью отображения их всех. Отсюда вы можете выполнять массовые операции над контейнерами (например, останавливать их) или щелкнуть по имени контейнера, чтобы получить более подробную информацию о нем и выполнить отдельные операции, относящиеся к нему. Например, будет показана опция для удаления работающего контейнера.

Страница «Образы» очень похожа на страницу «Контейнеры» и также позволяет выбрать несколько образов и выполнять с ними массовые операции. Если вы нажмете на идентификатор образа, вам будет предложено несколько интересных опций, таких как создание контейнера из образа и тегирование образа.

Помните, что Portainer может отставать от официальной функциональности Docker, – если вы хотите использовать самые последние и лучшие функциональные возможности, возможно, придется прибегнуть к командной строке.

ОБСУЖДЕНИЕ

Portainer – один из многих интерфейсов, доступных для Docker, и один из самых популярных с множеством функций и активной разработкой. В качестве примера вы можете использовать его для управления удаленными компьютерами, возможно, после запуска на них контейнеров по методу 32.

МЕТОД 46 Создание графа зависимостей образов Docker

Система наложения файлов в Docker – чрезвычайно мощная концепция, которая может сэкономить пространство и значительно ускорить сборку программного обеспечения. Но как только вы начнете использовать много образов, может быть трудно понять, как они связаны. Команда `docker images -a` вернет список всех слоев в вашей системе, но такой способ не удобен для пользователя – гораздо проще визуализировать связи между вашими образами, создав их дерево в виде изображения с помощью Graphviz.

Это также демонстрирует способность Docker упрощать сложные задачи. Установка всех компонентов для создания образа на хост-компьютере ранее включала бы в себя длинный ряд подверженных ошибкам шагов, но с помощью Docker его можно превратить в одну переносимую команду, которая с гораздо меньшей вероятностью не будет работать.

ПРОБЛЕМА

Вы хотите визуализировать дерево образов, хранящихся на вашем хосте.

РЕШЕНИЕ

Используйте образ, который мы создали (на основе образа CenturyLink Labs) с помощью этой функциональности, чтобы на выходе получить PNG-файл или веб-представление. Этот образ содержит сценарии, которые используют Graphviz для создания файла изображения PNG.

Данный метод применяет образ Docker в `dockerinpractice/docker-image-graph`. Со временем этот образ может устареть и перестать работать, поэтому следует выполнить следующие команды, чтобы убедиться, что он актуален.

Листинг 6.1. Сборка обновленного образа docker-image-graph (необязательно)

```
$ git clone https://github.com/docker-in-practice/docker-image-graph
$ cd docker-image-graph
$ docker build -t dockerinpractice/docker-image-graph
```

Все, что вам нужно сделать в вашей команде `run`, – это смонтировать сокет сервера Docker, и все готово, как показано в следующем листинге.

Листинг 6.2. Генерация изображения дерева слоев

```
$ docker run --rm \
 -v /var/run/docker.sock:/var/run/docker.sock \
 dockerinpractice/docker-image-graph > docker_images.png
```

Удаляет контейнер при создании образа

Монтирует сокет домена Unix сервера Docker, поэтому вы можете получить доступ к серверу Docker из контейнера. Если вы изменили значение по умолчанию для демона Docker, это не сработает

Определяет образ и создает PNG-файл в качестве артефакта

На рис. 6.2 показан PNG-файл дерева образов с одного из наших компьютеров. Из этого рисунка видно, что у образов node и golang: 1.3 общий корень, и что golang: runtime совместно использует глобальный корень только с образом golang: 1.3. Аналогично образ mesosphere собран из того же корня, что и образ ubuntu-upstart.

Вам может быть интересно, что такое глобальный корень. Это псевдо-образ, размер которого – ровно 0 байтов.

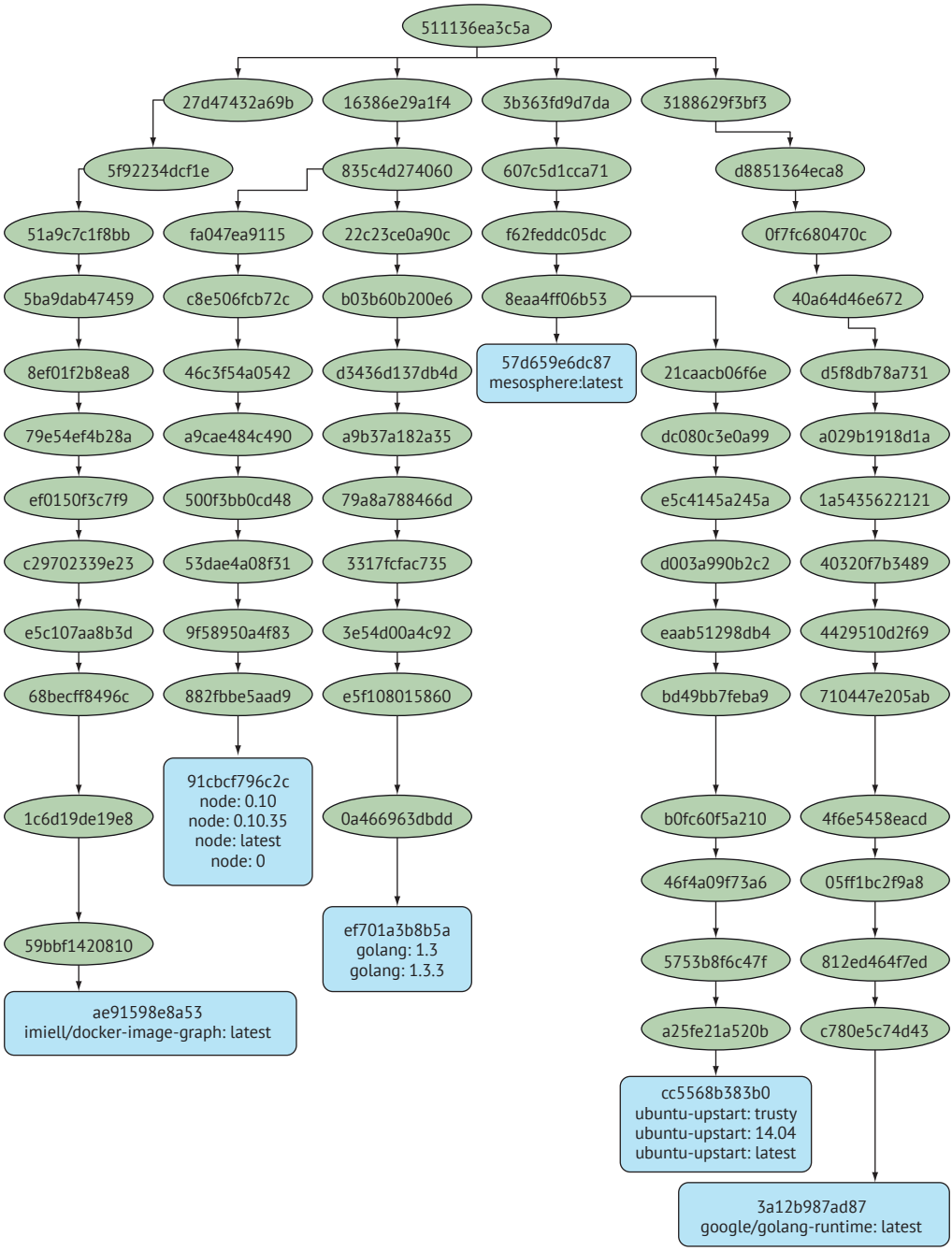


Рис. 6.2 ❖ Диаграмма дерева образов

ОБСУЖДЕНИЕ

Когда вы начинаете создавать больше образов Docker, возможно, в рамках непрерывной доставки, описанной в главе 9, может быть сложно отслеживать историю образа и то, на чем он собран. Это особенно важно, если вы пытаетесь ускорить доставку, используя совместно больше слоев для оптимизации по размеру. Периодическое извлечение всех образов и создание графа может быть отличным способом для отслеживания.

МЕТОД 47 Прямое действие: выполнение команд в контейнере

На заре становления Docker многие пользователи добавляли SSH-серверы к своим образам, чтобы иметь возможность доступа к ним с помощью оболочки извне. Docker осуждал это, так как он рассматривал контейнер как виртуальную машину (а мы знаем, что контейнеры не являются таковыми) и добавлял издержки процесса в систему, которая в этом не нуждается. Многие возразили, что после запуска простого способа попасть в контейнер не было. В результате Docker ввел команду `exec`, которая была намного более точным решением проблемы воздействия и проверки внутренних компонентов контейнеров после их запуска. Именно эту команду мы обсудим здесь.

ПРОБЛЕМА

Вы хотите выполнять команды в работающем контейнере.

РЕШЕНИЕ

Используйте команду `docker exec`.

Следующая команда запускает контейнер в фоновом режиме (с `-d`) и сообщает ему, чтобы он спал вечно (ничего не делал). Мы назовем эту команду `sleeper`.

```
docker run -d --name sleeper debian sleep infinity
```

Теперь, когда вы запустили контейнер, можете выполнять с ним различные действия с помощью команды `exec`. Ее можно расценивать как ту, у которой есть три режима, перечисленные в табл. 6.1.

Таблица 6.1 ❖ Режимы exec

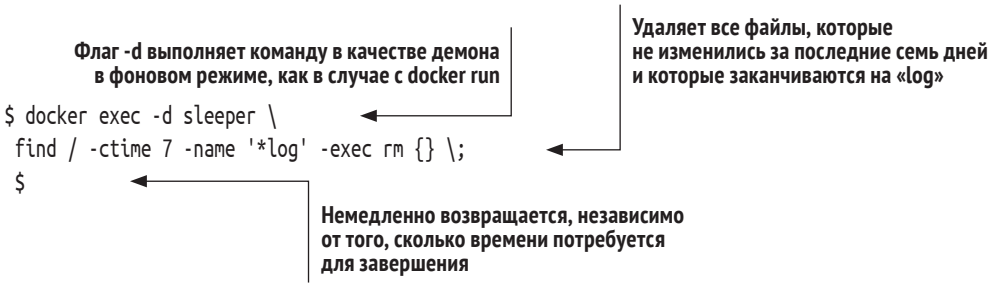
Режим	Описание
Базовый	Выполняет команду в контейнере синхронно с командной строкой
Демон	Выполняет команду в фоновом режиме в контейнере
Интерактивный	Выполняет команду и позволяет пользователю взаимодействовать с ней

Сначала рассмотрим базовый режим. Следующая команда выполняет команду echo внутри спящего контейнера.

```
$ docker exec sleeper echo "hello host from container"
hello host from container
```

Обратите внимание, что структура этой команды очень напоминает docker run, но вместо идентификатора образа мы присваиваем ей идентификатор работающего контейнера. Команда echo относится к двоичному файлу echo внутри контейнера, а не снаружи.

Режим демона запускает команду в фоновом режиме; вы не увидите вывод в своем терминале. Это может быть полезно для обычных служебных задач, когда нужно запустить команду и забыть, например, при очистке файлов журнала:



Наконец, у нас есть интерактивный режим. Он позволяет выполнять любые команды из контейнера. Чтобы задействовать его, вам, как правило, понадобится указать, что оболочка должна работать в интерактивном режиме, что в следующем коде является bash:

```
$ docker exec -i -t sleeper /bin/bash
root@d46dc042480f:/#
```

Аргументы -i и -t делают то, с чем вы уже сталкивались при работе с docker run, – они делают команду интерактивной и настраивают TTY-устройство, чтобы оболочки работали правильно. После этого у вас появится приглашение в контейнере.

ОБСУЖДЕНИЕ

Вход внутрь контейнера – важный этап отладки, когда что-то идет не так или если вы хотите выяснить, что делает контейнер. Часто невозможно использовать метод подключения и отключения, задействованный в методе 44, потому что процессы в контейнерах обычно запускаются на переднем плане, что делает невозможным получение доступа к приглашению оболочки. Поскольку exec позволяет указать исполняемый файл, который вы хотите запустить, это не проблема … если в файловой системе контейнера действительно есть этот файл.

В частности, при использовании метода 58 для создания контейнера с одним двоичным файлом вы не сможете запустить оболочку. В этом случае

нужно применить метод 57 как способ с минимальными издержками для разрешения exec.

МЕТОД 48 Вы находитесь в контейнере Docker?

При создании контейнеров обычно помещают логику в сценарии оболочки, а не пытаются напрямую писать сценарий в файле Dockerfile. Или же у вас могут быть разные сценарии, используемые во время работы контейнера. В любом случае задачи, которые они выполняют, часто тщательно приспособлены для использования внутри контейнера и могут представлять опасность при запуске на «обычной» машине. В подобных ситуациях полезно иметь защитные ограждения для предотвращения случайного выполнения вне контейнера.

ПРОБЛЕМА

Ваш код должен знать, работаете ли вы из контейнера Docker.

РЕШЕНИЕ

Проверьте наличие файла /.dockerenv. Если он есть, скорее всего, вы находитесь в контейнере Docker.

Обратите внимание, что это не железная гарантия, – если кто-либо или что-либо удалило файл /.dockerenv, эта проверка может дать неверные результаты. Такие сценарии маловероятны, но в худшем случае вы получите ложный положительный результат без каких-либо побочных эффектов; вы будете думать, что не находитесь в контейнере Docker и в худшем случае *не* запустите потенциально разрушительный фрагмент кода.

Более реалистичный сценарий состоит в том, что это недокументированное поведение Docker было изменено или удалено в более новой версии Docker (или вы используете версию, созданную до того, как это поведение было впервые реализовано).

Код может быть частью сценария запуска bash, как показано в следующем листинге, за которым следует оставшаяся часть кода запуска сценария.

Листинг 6.3. Сценарий оболочки завершается ошибкой, если выполняется вне контейнера

```
#!/bin/bash
if ! [ -f /.dockerenv ]
then
    echo 'Not in a Docker container, exiting.'
    exit 1
fi
```

Конечно, можно использовать противоположную логику, чтобы определить, что вы не работаете в контейнере, если вам это нужно:

Листинг 6.4. Сценарий оболочки завершается ошибкой, если выполняется внутри контейнера

```
#!/bin/bash
if [ -f /.dockerenv ]
then
    echo 'In a Docker container, exiting.'
    exit 1
fi
```

В этом примере для определения существования файла используется bash, но у подавляющего большинства языков программирования будут свои способы определения существования файлов в файловой системе контейнера (или хоста).

ОБСУЖДЕНИЕ

Вам, наверное, интересно, как часто возникает такая ситуация. Достаточно часто случается так, что это постоянно обсуждается на форумах Docker, где разгораются какие-то религиозные споры по поводу того, является ли это допустимым вариантом использования или что-то еще в дизайне вашего приложения не так.

Оставив в стороне эти обсуждения, вы легко можете оказаться в ситуации, когда нужно переключить путь кода в зависимости от того, находитесь ли вы в контейнере Docker или нет. Один из таких примеров, с которыми мы столкнулись, – это использование файла Makefile для сборки контейнера.

Резюме

- Вы можете настроить свой компьютер так, чтобы запускать Docker без использования команды `sudo`.
- Используйте встроенные команды Docker для очистки неиспользуемых контейнеров и томов.
- Можно активировать внешние инструменты, чтобы по-новому предоставлять информацию о своих контейнерах.
- Команда `docker exec` – это правильный способ попасть внутрь работающего контейнера; не устанавливайте SSH.

Глава 7

Управление конфигурацией: наводим порядок в доме

О чем рассказывается в этой главе:

- управление сборкой образов с помощью файлов Dockerfile;
- сборка образов с использованием традиционных инструментов управления конфигурацией;
- управление секретной информацией, необходимой для сборки образов;
- уменьшение размера образов для более быстрой, легкой и безопасной доставки.

Управление конфигурацией – это искусство управления средами, чтобы они были стабильными и предсказуемыми. Такие инструменты, как Chef и Puppet, пытались облегчить бремя системных администраторов, связанное с управлением несколькими компьютерами. В некоторой степени Docker также уменьшает это бремя, делая программную среду изолированной и переносимой. Тем не менее методы управления конфигурацией необходимы для производства образов Docker, и это важная для изучения тема.

К концу этой главы вы узнаете, как интегрировать существующие инструменты с Docker, решить некоторые специфичные для Docker проблемы, такие как удаление секретов из слоев, и следовать рекомендациям по минимизации вашего окончательного образа. По мере того как вы получаете больше опыта, работая с Docker, эти методы дадут вам возможность собирать образы для любых нужд, которые вы пытаетесь удовлетворить.

7.1. Управление конфигурацией и файлы Dockerfile

Файлы Dockerfile считаются стандартным способом создания образов Docker. Они часто сбивают с толку с точки зрения их значения для управления конфигурацией. У вас может возникнуть много вопросов (особенно если имеется опыт работы с другими инструментами управления конфигурацией), таких как:

- Что произойдет, если базовый образ изменится?
- Что произойдет, если устанавливаемые мной пакеты изменятся, и я выполню повторную сборку?
- Это заменяет Chef/Puppet/Ansible?

На самом деле файлы Dockerfile довольно просты: начиная с заданного образа, Dockerfile определяет серию команд оболочки и мета-инструкций для Docker, которые будут создавать желаемый конечный образ.

Файлы Dockerfile предоставляют общий, простой и универсальный язык для подготовки образов Docker. Внутри них вы можете использовать все что угодно для достижения желаемого конечного состояния: вызвать Puppet, копировать в другой сценарий или в целую файловую систему!

Сначала мы рассмотрим, как можно справиться с некоторыми незначительными проблемами, которые приносят файлы Dockerfile. Затем перейдем к более важным вопросам, которые только что изложили.

МЕТОД 49 Создание надежных специальных инструментов с помощью ENTRYPOINT

Потенциал Docker, позволяющий запускать команды *где угодно*, означает, что комплексные специальные инструкции или сценарии, которые запускаются в командной строке, могут быть предварительно сконфигурированы и обернуты в упакованный инструмент.

Неправильно истолкованная инструкция `ENTRYPOINT` – важная часть этого. Вы увидите, что она позволяет создавать образы Docker в качестве инструментов, которые хорошо инкапсулированы, четко определены и достаточно гибки, чтобы быть полезными.

ПРОБЛЕМА

Вы хотите определить команду, которую будет выполнять контейнер, но оставить аргументы команды на усмотрение пользователя.

РЕШЕНИЕ

Используйте инструкцию `ENTRYPOINT`.

В качестве демонстрации мы представим простой сценарий в корпорации, где обычной задачей администратора является очистка старых файлов журнала. Зачастую это подвержено ошибкам, и люди случайно удаляют не те вещи, поэтому мы будем использовать образ Docker, чтобы уменьшить риск возникновения проблем.

Следующий сценарий (который вы должны назвать «clean_log», когда сохраняете его) удаляет журналы за определенное количество дней, где число дней передается в качестве параметра командной строки. Создайте новую папку в любом месте с любым именем, перейдите в нее и поместите туда clean_log.

Листинг 7.1. Сценарий оболочки clean_log

```
#!/bin/bash
echo "Cleaning logs over $1 days old"
find /log_dir -ctime "$1" -name '*log' -exec rm {} \;
```

Обратите внимание, что очистка журнала происходит в папке /log_dir. Эта папка будет существовать только при монтировании во время выполнения. Возможно, вы также заметили, что нет проверки того, был ли аргумент передан в сценарий. Причина этого будет раскрыта по мере прохождения данного метода.

Теперь давайте откроем файл Dockerfile в том же каталоге, чтобы создать образ с помощью сценария, выполняемым в качестве определенной команды или *точки входа*.

Листинг 7.2. Создание образа с помощью сценария clean_log

```
FROM ubuntu:17.04
ADD clean_log /usr/bin/clean_log
 RUN chmod +x /usr/bin/clean_log
ENTRYPOINT ["/usr/bin/clean_log"]
 CMD ["7"]
```

Добавляет предыдущий корпоративный сценарий clean_log к образу

Определяет точку входа для этого образа как сценарий clean_log

Определяет аргумент по умолчанию для команды entrypoint (7 дней)

> **СОВЕТ.** Вы заметите, что мы обычно предпочитаем форму массива для `CMD` и `ENTRYPOINT` (например, `CMD ["/usr/bin/command"]`) поверх формы оболочки (`CMD /usr/bin/command`). Это связано с тем, что форма оболочки автоматически добавляет команду `/bin/bash -c` к введенной вами команде, что может привести к непредвиденному поведению. Иногда, однако, форма оболочки более полезна (см. метод 55).

Разница между `ENTRYPOINT` и `CMD` часто сбивает с толку. Ключевым моментом для понимания является то, что точка входа всегда будет запускаться при запуске образа, даже если для вызова `docker run` указана команда. Если вы попытаетесь ввести команду, она добавит это как аргумент к точке входа, заменив значение по умолчанию, определенное в инструкции `CMD`. Вы можете переопределить точку входа, только если явно передаете флаг `--entrypoint`

команде `docker run`. Это означает, что запуск образа с помощью команды /bin/bash не даст вам оболочки; скорее, он предоставит /bin/bash в качестве аргумента сценарию clean_log.

Тот факт, что аргумент по умолчанию определяется инструкцией `CMD`, означает, что предоставленный аргумент проверять не нужно. Вот как можно создать и вызвать этот инструмент:

```
docker build -t log-cleaner .
docker run -v /var/log/myapplogs:/log_dir log-cleaner 365
```

После того как образ собран, его вызывают путем монтирования `/var/log/myapplogs` в каталог, который будет использоваться сценарием, и передачи `365` для удаления файлов журнала старше года, а не недели.

Если кто-то попытается использовать образ неправильно, не указав количество дней, он получит сообщение об ошибке:

```
$ docker run -ti log-cleaner /bin/bash
Cleaning logs over /bin/bash days old
find: invalid argument `-name' to `-ctime'
```

ОБСУЖДЕНИЕ

Этот был довольно тривиальный пример, но вы можете себе представить, что корпорация способна применять его для централизованного управления сценариями, используемыми в своем имуществе, чтобы их можно было поддерживать и безопасно распространять в частном реестре.

Вы можете просмотреть и использовать образ, который мы создали в этом методе, на странице https://hub.docker.com/r/dockerinpractice/log-cleaner/.

МЕТОД 50 Предотвращение перемещения пакетов путем указания версий

Файлы Dockerfile имеют простой синтаксис и ограниченную функциональность. Они могут значительно помочь при уточнении требований вашей сборки и стабильности производства образов, но не могут гарантировать повторяемость сборок. Мы изучим один из многочисленных подходов к решению этой проблемы и снижению риска неприятных сюрпризов, когда происходит изменение основных зависимостей управления пакетами.

Этот метод полезен для того, чтобы избежать моментов типа «вчера работало», и может быть знаком вам, если вы использовали классические инструменты управления конфигурацией. Создание образов Docker принципиально отличается от обслуживания сервера, но некоторые усвоенные уроки все еще применимы.

> **ПРИМЕЧАНИЕ.** Этот метод будет работать только для образов на основе Debian, таких как Ubuntu. Пользователи Yum могут найти аналогичные методы, чтобы это работало с их менеджером пакетов.

ПРОБЛЕМА

Вы хотите убедиться, что ваши deb-пакеты соответствуют ожидаемым вами версиям.

РЕШЕНИЕ

Запустите сценарий, чтобы определить версии всех зависимых пакетов в системе, настроенной по вашему желанию. Затем установите конкретные версии в свой файл Dockerfile, чтобы убедиться, что эти версии соответствуют вашим ожиданиям.

Базовую проверку версий можно выполнить с помощью вызова `apt-cache` в подходящей вам системе:

```
$ apt-cache show nginx | grep ^Version:
Version: 1.4.6-1ubuntu3
```

Затем можете указать версию в вашем Dockerfile следующим образом:

```
RUN apt-get -y install nginx=1.4.6-1ubuntu3
```

Этого может быть достаточно. Нет гарантии, что все зависимости от этой версии nginx имеют те же версии, которые вы первоначально верифицировали.

Вы можете получить информацию обо всех этих зависимостях, добавив флаг `--recurse` к аргументу:

```
apt-cache --recurse depends nginx
```

Вывод этой команды пугающе огромен, поэтому получить список требований к версии довольно сложно. К счастью, мы поддерживаем образ Docker (что же еще?), чтобы вам было проще. Он выводит строку `RUN`, которую вы должны поместить в свой Dockerfile, чтобы убедиться, что версии всех зависимостей верны.

```
$ docker run -ti dockerinpractice/get-versions vim
RUN apt-get install -y \
vim=2:7.4.052-1ubuntu3 vim-common=2:7.4.052-1ubuntu3 \
vim-runtime=2:7.4.052-1ubuntu3 libacl1:amd64=2.2.52-1 \
libc6:amd64=2.19-0ubuntu6.5 libc6:amd64=2.19-0ubuntu6.5 \
libgpm2:amd64=1.20.4-6.1 libpython2.7:amd64=2.7.6-8 \
libselinux1:amd64=2.2.2-1ubuntu0.1 libselinux1:amd64=2.2.2-1ubuntu0.1 \
libtinfo5:amd64=5.9+20140118-1ubuntu1 libattr1:amd64=1:2.4.47-1ubuntu1 \
libgcc1:amd64=1:4.9.1-0ubuntu1 libgcc1:amd64=1:4.9.1-0ubuntu1 \
libpython2.7-stdlib:amd64=2.7.6-8 zlib1g:amd64=1:1.2.8.dfsg-1ubuntu1 \
libpcre3:amd64=1:8.31-2ubuntu2 gcc-4.9-base:amd64=4.9.1-0ubuntu1 \
gcc-4.9-base:amd64=4.9.1-0ubuntu1 libpython2.7-minimal:amd64=2.7.6-8 \
mime-support=3.54ubuntu1.1 mime-support=3.54ubuntu1.1 \
libbz2-1.0:amd64=1.0.6-5 libdb5.3:amd64=5.3.28-3ubuntu3 \
```

```
libexpat1:amd64=2.1.0-4ubuntu1 libffi6:amd64=3.1~rc1+r3.0.13-12 \
libncursesw5:amd64=5.9+20140118-1ubuntu1 libreadline6:amd64=6.3-4ubuntu2 \
libsqlite3-0:amd64=3.8.2-1ubuntu2 libssl1.0.0:amd64=1.0.1f-1ubuntu2.8 \
libssl1.0.0:amd64=1.0.1f-1ubuntu2.8 readline-common=6.3-4ubuntu2 \
debconf=1.5.51ubuntu2 dpkg=1.17.5ubuntu5.3 dpkg=1.17.5ubuntu5.3 \
libnewt0.52:amd64=0.52.15-2ubuntu5 libslang2:amd64=2.2.4-15ubuntu1 \
vim=2:7.4.052-1ubuntu3
```

В какой-то момент ваша сборка завершится неудачно, потому что версия больше не доступна. Когда это произойдет, вы сможете увидеть, какой пакет был изменен, и просмотреть изменения, чтобы определить, подходит ли он для ваших конкретных образов.

В этом примере предполагается, что вы берете ubuntu:14.04. При использовании другого варианта Debian разветвите репозиторий, измените инструкцию `FROM` и соберите ее. Репозиторий доступен на странице https://github.com/docker-in-practice/get-versions.git.

Хотя этот метод может помочь вам в случае со стабильностью сборки, он ничего не делает в плане безопасности, поскольку вы все еще скачиваете пакеты из репозитория, над которым у вас нет прямого контроля.

ОБСУЖДЕНИЕ

Этот метод может потребовать много усилий, чтобы гарантировать, что текстовый редактор именно такой, как вы ожидаете. В полевых условиях, однако, перемещение пакетов может привести к ошибкам, которые невероятно трудно определить. Библиотеки и приложения могут незаметно перемещаться в сборках из одного дня в другой, и выясняя, что произошло, вы можете испортить себе день.

Закрепляя версии как можно плотнее в своем файле Dockerfile, вы гарантируете, что произойдет одно из двух. Либо сборка завершится успешно, и ваше программное обеспечение будет вести себя так же, как и вчера, либо она завершится неудачно из-за того, что часть программного обеспечения изменилась, и вам придется повторно протестировать свой конвейер разработки. Во втором случае вы знаете, что изменилось, и можете свести на нет любые сбои, которые последуют за этим конкретным изменением.

Дело в том, что, когда вы продолжаете сборку и интеграцию, уменьшение количества меняющихся переменных сокращает время, затрачиваемое на отладку. Это означает деньги для вашего бизнеса.

МЕТОД 51 Замена текста с помощью perl -p -i -e

Нередко при сборке образов с помощью файлов Dockerfile вам необходимо заменить определенные элементы текста в нескольких файлах. Для этого существует множество решений, но мы познакомим вас с несколько необычным фаворитом, который особенно удобен в файлах Dockerfile.

ПРОБЛЕМА

Вы хотите изменить определенные строки в файлах во время сборки.

РЕШЕНИЕ

Используйте команду `perl -p -i -e`.
Мы рекомендуем эту команду по нескольким причинам:

- в отличие от `sed -i` (команда с похожим синтаксисом и эффектом) эта команда работает с несколькими файлами из коробки, даже если сталкивается с проблемой с одним из файлов. Это означает, вы можете запустить ее в каталоге с шаблоном поиска '*', не опасаясь, что она внезапно сломается при добавлении каталога в более позднюю ревизию пакета;
- как и в случае с `sed`, вы можете заменить косую черту в поиске и заменить команды другими символами;
- ее легко запомнить (мы называем ее командой «perl pie»).

> **ПРИМЕЧАНИЕ.** Этот метод предполагает знание регулярных выражений. Если вы не знакомы с регулярными выражениями, существует множество сайтов, которые могут вам помочь.

Вот типичный пример использования этой команды:

```
perl -p -i -e 's/127\.0\.0\.1/0.0.0.0/g' *
```

В данной команде флаг `-p` просит Perl принять цикл, пока он обрабатывает все видимые строки. Флаг `-i` просит Perl обновить совпадающие строки на месте, а флаг `-e` – рассматривать предоставленную строку как программу Perl. `S` – это инструкция Perl для поиска и замены строк, когда они совпадают на входе. Здесь `127.0.0.1` заменяется на `0.0.0.0`. Модификатор `g` обеспечивает обновление всех совпадений, а не только первого в любой заданной строке. Наконец, знак звездочки (*) применяет обновление ко всем файлам в этом каталоге.

Предыдущая команда выполняет довольно распространенное действие для контейнеров Docker. Она заменяет стандартный IP-адрес локального хоста (`127.0.0.1`) на тот, который указывает «любой» адрес IPv4 (`0.0.0.0`) при использовании в качестве адреса для прослушивания. Многие приложения ограничивают доступ к локальному IP-адресу, прослушивая только этот адрес, и вам часто нужно менять его в конфигурационных файлах на «любой», потому что вы будете получать доступ к приложению со своего хоста, который воспринимается контейнером как внешний компьютер.

> **СОВЕТ.** Если вы не можете получить доступ к приложению в контейнере Docker с хост-компьютера, несмотря на то что порт открыт, может быть, стоит попробовать обновить адреса для прослушивания на 0.0.0.0 в файле конфигурации приложения и выполнить перезапуск. Возможно,

приложение отклоняет вас, потому что вы приходите не с его локального хоста. Использование `--net = host` (будет рассмотрено позже в методе 109) при запуске образа может помочь подтвердить эту гипотезу.

Еще одна приятная особенность `perl -p -i -e` (и `sed`) заключается в том, что вы можете использовать другие символы для замены прямой косой черты, если экранирование косых черт становится неудобным. Вот пример из одного из наших сценариев, в котором добавляется несколько директив в файл сайта Apache по умолчанию. Эта неуклюжая команда,

```
perl -p -i -e 's/\/usr\/share\/www/\/var\/www\/html/g' /etc/apache2/*
```

теперь выглядит так:

```
perl -p -i -e 's@/usr/share/www@/var/www/html/@g' /etc/apache2/*
```

В тех редких случаях, когда вы хотите сопоставить или заменить оба символа / и @, можете попробовать другие, такие как | или #.

ОБСУЖДЕНИЕ

Это один из тех советов, которые применимы не только за пределами мире Docker, но и внутри него. Это полезный инструмент в вашем арсенале.

Мы находим данный метод особенно полезным благодаря его широкому применению за пределами Dockerfiles в сочетании с легкостью, с которой его запоминают: он «прост как пирог», простите за каламбур.

МЕТОД 52 Сращивание образов

Следствием дизайна файлов Dockerfile и их создания образов Docker является то, что конечный образ содержит состояние данных на каждом этапе в Dockerfile. В процессе сборки образов, возможно, потребуется скопировать секреты, чтобы обеспечить работоспособность сборки. Этими секретами могут быть SSH-ключи, сертификаты или файлы паролей. Удаление секретов перед фиксацией вашего образа не обеспечивает вам никакой реальной защиты, так как они будут присутствовать в более высоких слоях конечного образа. Злоумышленник может легко извлечь их.

Одним из способов решения этой проблемы является сращивание результирующего образа.

ПРОБЛЕМА

Вы хотите удалить информацию о секрете из истории слоя вашего образа.

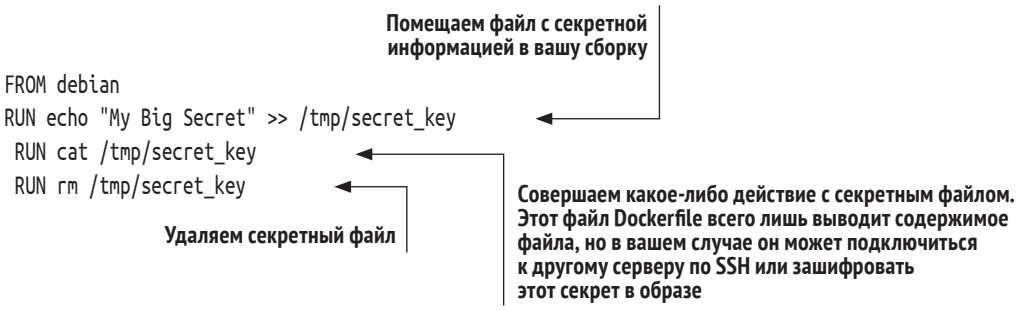
РЕШЕНИЕ

Создайте экземпляр контейнера с помощью образа, экспортируйте его, импортируйте, а затем пометьте исходным идентификатором образа.

Чтобы продемонстрировать сценарий, в котором это может быть полезно, давайте создадим простой файл Dockerfile в новом каталоге, содержащем

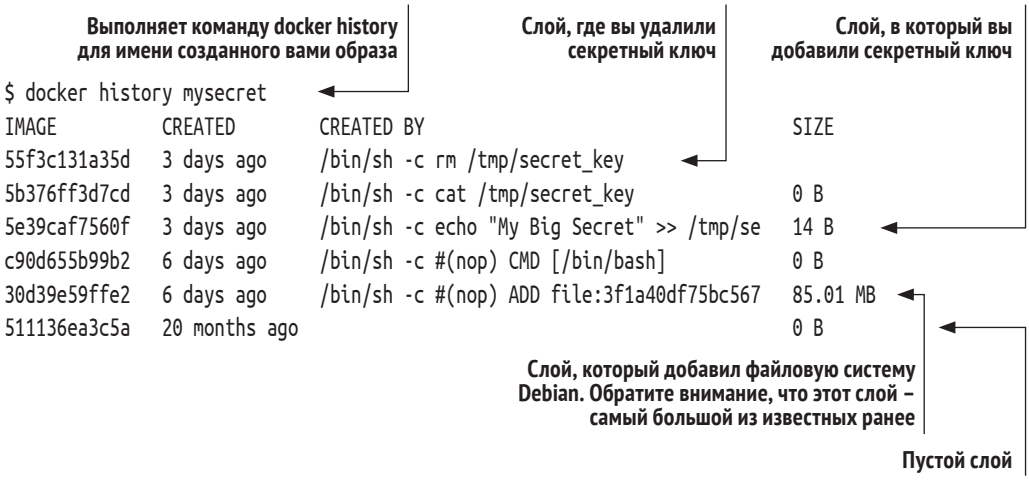
Большой Секрет. Выполните команду `mkdir secrets && cd secrets`, а затем создайте файл Dockerfile в этой папке со следующим содержимым.

Листинг 7.3. Dockerfile, который копирует и удаляет секрет



Теперь выполните команду `docker build -t mysecret`, чтобы собрать и пометить этот Dockerfile.

Как только он будет собран, можете проверить слои результирующего образа Docker с помощью команды `docker history`:

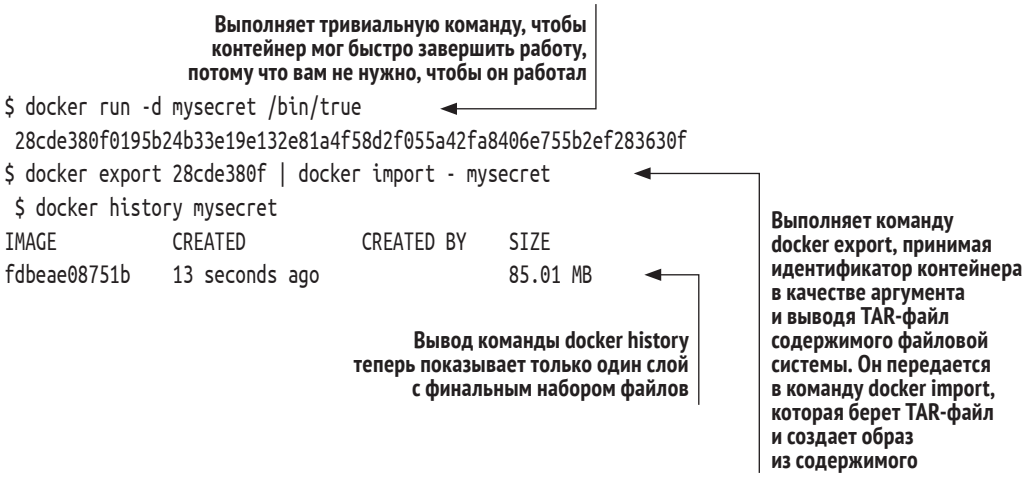


Теперь представьте, что вы скачали этот образ из открытого реестра. Можете просмотреть историю слоя и затем выполнить следующую команду, чтобы раскрыть информацию о секрете:

```
$ docker run 5b376ff3d7cd cat /tmp/secret_key
My Big Secret
```

Здесь мы запустили определенный слой и проинструктировали его, чтобы он использовал команду `cat` для секретного ключа, который мы удалили на более высоком уровне. Как видите, файл доступен.

Теперь у вас есть «опасный» контейнер с секретом внутри, который, как вы видели, можно взломать для получения его секретов. Чтобы сделать этот образ безопасным, вам нужно будет выполнить его *сращивание*. Это означает, вы будете хранить те же данные в образе, но удалите информацию о промежуточных слоях. Для этого нужно экспортировать образ как тривиально выполненный контейнер, а затем повторно импортировать и пометить получившийся образ:



Аргумент для команды `docker import` указывает на то, что вы хотите прочитать TAR-файл из стандартного ввода команды. Последний аргумент для `docker import` указывает, как должен быть помечен импортируемый образ. В этом случае вы перезаписываете предыдущий тег.

Поскольку в образе теперь только один слой, записей о слоях, содержащих секреты, нет. Теперь из образа нельзя извлечь никакие секреты.

ОБСУЖДЕНИЕ

Этот метод достаточно полезен для повторного использования в различных местах этой книги, например в разделе 7.3.

Если вы хотите использовать этот метод, стоит подумать о том, что преимущества многослойных образов при кешировании слоев и времени скачивания могут быть потеряны. Если ваша организация тщательно планирует это, данный метод может сыграть роль в реальном использовании этих образов.

МЕТОД 53 Управление чужими пакетами с помощью Alien

Хотя в большинстве примеров с файлами Dockerfile, приведенных в этой книге (и в интернете), используется образ на основе Debian, реальность разработки программного обеспечения означает, что многие люди не будут иметь дело исключительно с ними.

К счастью, существуют инструменты, которые помогут вам в этом.

ПРОБЛЕМА

Вы хотите установить пакет из чужого дистрибутива.

РЕШЕНИЕ

Используйте инструмент под названием Alien для преобразования пакета. Alien внедрен в образ Docker, который мы будем использовать как часть этого метода.

Alien – это утилита командной строки, предназначенная для преобразования файлов пакетов в различные форматы, перечисленные в табл. 7.1. Нам неоднократно приходилось заставлять работать пакеты из сторонних систем управления пакетами, например файлы .deb в CentOS и файлы .rpm в системах, не основанных на Red Hat.

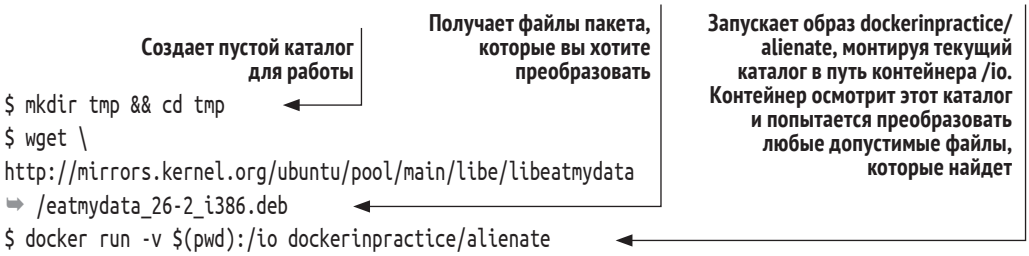
Таблица 7.1 ❖ Форматы пакетов, поддерживаемые Alien

Расширение	Описание
.deb	Пакет Debian
.rpm	Управление пакетами Red Hat
.tgz	TAR-файл, сжатый с помощью утилиты сжатия GNU Zip из дистрибутива Slackware
.pkg	PKG-Пакет для Solaris
.slp	Пакет Stampede

> **ПРИМЕЧАНИЕ.** Для этого метода пакеты Solaris и Stampede полностью не рассматриваются. Solaris требует программного обеспечения, свойственного исключительно Solaris, а Stampede – заброшенный проект.

Исследуя эту книгу, мы обнаружили, что, может быть, несколько проблематично устанавливать Alien в дистрибутивах не на основе Debian. Это книга о Docker, и мы, естественно, решили предоставить инструмент преобразования в формате образа Docker. В качестве бонуса инструмент использует команду `ENTRYPOINT` из метода 49, чтобы упростить использование инструментов.

В качестве примера давайте загрузим и конвертируем (с помощью Alien) пакет eatmydata, который будет использоваться в методе 62.



```
Examining eatmydata_26-2_i386.deb from /io
eatmydata_26-2_i386.deb appears to be a Debian package
eatmydata-26-3.i386.rpm generated
eatmydata-26.slp generated
eatmydata-26.tgz generated
==========================================
/io now contains:
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
eatmydata_26-2_i386.deb
==========================================
$ ls -1
eatmydata_26-2_i386.deb
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
```

Контейнер информирует вас о своих действиях, когда запускает сценарий-обёртку Alien

Файлы были преобразованы в файлы RPM, Slackware TGZ и Stampede

Кроме того, вы можете передать URL-адрес пакета для скачивания и преобразования непосредственно в команду `docker run`:

```
$ mkdir tmp && cd tmp
$ docker run -v $(pwd):/io dockerinpractice/alienate \
http://mirrors.kernel.org/ubuntu/pool/main/libe/libeatmydata
➥ /eatmydata_26-2_i386.deb
wgetting http://mirrors.kernel.org/ubuntu/pool/main/libe/libeatmydata
➥ /eatmydata_26-2_i386.deb
--2015-02-2610:57:28-- http://mirrors.kernel.org/ubuntu/pool/main/libe
➥ /libeatmydata/eatmydata_26-2_i386.deb
Resolving mirrors.kernel.org (mirrors.kernel.org)... 198.145.20.143,
➥ 149.20.37.36, 2001:4f8:4:6f:0:1994:3:14, ...
Connecting to mirrors.kernel.org (mirrors.kernel.org)|198.145.20.143|:80...
➥ connected.
HTTP request sent, awaiting response... 200 OK
Length: 7782 (7.6K) [application/octet-stream]
Saving to: 'eatmydata_26-2_i386.deb'

     0K .......                   100% 2.58M=0.003s

2015-02-2610:57:28 (2.58 MB/s) - 'eatmydata_26-2_i386.deb' saved
➥ [7782/7782]

Examining eatmydata_26-2_i386.deb from /io
eatmydata_26-2_i386.deb appears to be a Debian package
eatmydata-26-3.i386.rpm generated
eatmydata-26.slp generated
```

```
eatmydata-26.tgz generated
==================================================
/io now contains:
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
eatmydata_26-2_i386.deb
==================================================
$ ls -1
eatmydata_26-2_i386.deb
eatmydata-26-3.i386.rpm
eatmydata-26.slp
eatmydata-26.tgz
```

Если вы хотите запустить Alien в контейнере самостоятельно, то можете открыть контейнер следующим образом:

```
docker run -ti --entrypoint /bin/bash dockerinpractice/alienate
```

> **ВНИМАНИЕ!** Alien – инструмент категории «лучшее из возможного», и он не гарантирует работу с пакетами, которые вы ему предоставляете.

ОБСУЖДЕНИЕ

Использование Docker позволило четко сфокусировать внимание на «войнах с дистрибутивами», которые некоторое время бездействовали. Большинство организаций превратились в магазины Red Hat или Debian, которым не нужно было заниматься другими системами упаковки. Получение внутри организации запросов на внедрение образов Docker, которые основаны на дистрибутивах Alien, теперь не редкость.

Вот где этот метод поможет, поскольку «чужие» пакеты могут быть преобразованы в более дружественный формат. Эту тему мы вновь рассмотрим в главе 14, где будем осуждать безопасность.

7.2. Традиционные инструменты управления конфигурацией и Docker

Теперь перейдем к тому, как файлы Dockerfile могут работать вместе с более традиционными инструментами управления конфигурацией.

Мы рассмотрим здесь традиционное управление конфигурацией с помощью `make`, покажем, как можно использовать существующие сценарии Chef для подготовки образов с помощью Chef Solo, а также рассмотрим среду сценариев оболочки, созданную, для того чтобы помочь сторонним разработчикам выполнять сборку образов.

МЕТОД 54 Традиционно: использование make и Docker

В какой-то момент вы можете обнаружить, что наличие нескольких файлов Docker ограничивает процесс сборки. Например, невозможно создать какие-либо выходные *файлы*, если вы ограничиваете себя выполнением команды `docker build` и нет возможности получить переменные из файлов Dockerfile.

Это требование для дополнительных инструментов может быть выполнено рядом инструментов (включая простые сценарии оболочки). В данном методе мы рассмотрим, как использовать почтенный инструмент `make` для работы с Docker.

ПРОБЛЕМА

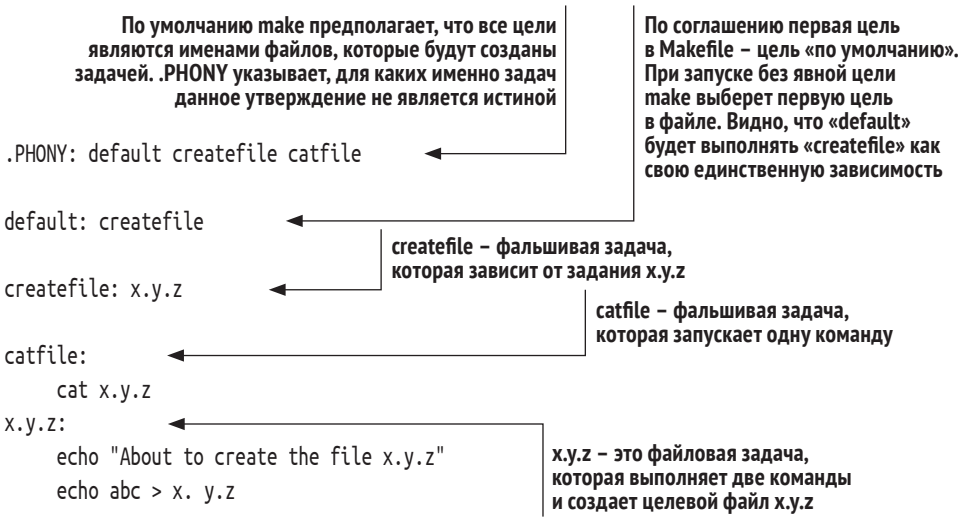
Вы хотите добавить дополнительные задачи к выполнению команды `docker build`.

РЕШЕНИЕ

Возьмите древний (в вычислительном отношении) инструмент под названием `make`.

Если вы ранее не использовали его, `make` – это инструмент, который принимает один или несколько входных файлов и создает выходной, но его также можно использовать в качестве средства запуска задач. Вот простой пример (обратите внимание, что все отступы должны быть вкладками):

Листинг 7.4. Простой файл Makefile



> **ПРЕДУПРЕЖДЕНИЕ.** Все отступы в Makefile должны быть вкладками, и каждая команда в цели запускается в другой оболочке (поэтому переменные среды не будут переноситься в поперечнике).

Если у вас есть предыдущий контент в файле с именем Makefile, можете вызвать любую цель с помощью такой команды, как `make createfile`.

Теперь давайте рассмотрим некоторые полезные шаблоны в Makefile – остальные цели, о которых мы поговорим, будут фальшивыми, поскольку сложно (хотя и возможно) использовать отслеживание изменений файлов для автоматического запуска сборок Docker. Файлы Dockerfile используют кеш слоев, поэтому сборки имеют тенденцию быть быстрыми.

Первый шаг – запустить файл Dockerfile. Поскольку Makefile состоит из команд оболочки, сделать это легко.

Листинг 7.5. Файл Makefile для сборки образа

```
base:
    docker build -t corp/base .
```

Обычные варианты этого файла работают так, как вы и ожидаете (такие как передача файла по программному каналу команде `docker build` для удаления контекста или применение `-f` для использования Dockerfile с другим именем), и вы можете использовать функцию зависимостей `make` для автоматической сборки базовых образов (используется в `FROM`) при необходимости. Например, если вы зафиксировали изменения в нескольких репозиториях в подкаталог с именем repos (это также легко сделать с помощью `make`), можете добавить цель, как показано в следующем листинге.

Листинг 7.6. Makefile для сборки образа в подкаталоге

```
app1: base
    cd repos/app1 && docker build -t corp/app1 .
```

Недостаток состоит в том, что каждый раз, когда ваш базовый образ нуждается в повторной сборке, Docker будет загружать контекст создания образа, включающий в себя все ваши репозитории. Это можно исправить, явно передав TAR-файл контекста создания образа в Docker.

Листинг 7.7. Makefile для сборки образа с определенным набором файлов

```
base:
    tar -cvf - file1 file2 Dockerfile | docker build -t corp/base -
```

Это явное выражение зависимостей обеспечит значительное увеличение скорости, если в вашем каталоге содержится большое количество файлов,

не имеющих отношения к сборке. Вы можете немного изменить эту цель, если хотите сохранить все зависимости сборки в другом каталоге.

Листинг 7.8. Makefile для сборки образа с определенным набором файлов с переименованными путями

```
base:
    tar --transform 's/^deps\///' -cf - deps/* Dockerfile | \
        docker build -t corp/base -
```

Здесь вы добавляете все в каталог deps в контекст создания образа и используете опцию `--transform` (доступна в последних версиях `tar` для Linux), чтобы убрать любые начальные «deps /» из имен файлов. В этом конкретном случае лучше было бы поместить зависимости (deps) и Dockerfile в свои собственные каталоги, чтобы разрешить выполнение обычной команды `docker build`, но полезно знать об этом расширенном использовании, поскольку оно может пригодиться в самых неожиданных местах. Всегда тщательно все обдумывайте, прежде чем использовать его, так как это усложняет процесс сборки.

Простая замена переменных – это относительно простой вопрос, но (как и в случае с `--transform` ранее) хорошенько подумайте, прежде чем воспользоваться ею, – файлы Dockerfile сознательно не поддерживают переменные, чтобы обеспечить легкую воспроизводимость сборок.

Здесь мы будем использовать переменные, переданные `make`, и выполним замену с использованием `sed`, но вы можете передавать и заменять их так, как вам нравится.

Листинг 7.9. Makefile для сборки образа с базовой подстановкой переменных Dockerfile

```
VAR1 ?= defaultvalue
base:
    cp Dockerfile.in Dockerfile
    sed -i 's/{VAR1}/$(VAR1)/' Dockerfile
    docker build -t corp/base .
```

Dockerfile будет обновляться каждый раз, когда запускается базовая цель, и вы можете увеличить число замен переменных, добавив больше строк `sed -i`. Чтобы переопределить значение по умолчанию `VAR1`, выполните команду `make VAR1 = newvalue base`. Если ваши переменные включают косую черту, может понадобиться выбрать другой разделитель `sed`, например `sed -i 's#{VAR1}#$(VAR1)#' Dockerfile`.

Наконец, если вы использовали Docker в качестве инструмента для сборки, нужно знать, как вернуть файлы из Docker. Мы представим пару вариантов в зависимости от вашего варианта использования.

Листинг 7.10. Makefile для копирования файлов из образа

```
singlefile: base
    docker run --rm corp/base cat /path/to/myfile > outfile
multifile: base
    docker run --rm -v $(pwd)/outdir:/out corp/base sh \
        -c "cp -r /path/to/dir/* /out/"
```

Здесь `singlefile` выполняет для файла команду `cat` и передает по программному каналу вывод в новый файл. Преимущество этого подхода заключается в автоматической установке правильного владельца файла, но он становится громоздким для нескольких файлов. При подходе с использованием `multifile` том монтируется в контейнере и все файлы копируются из каталога в том. Вы можете следить за этим с помощью команды `chown`, чтобы установить правильного владельца файлов, но имейте в виду, что вам, вероятно, придется вызывать ее с помощью `sudo`.

Сам проект Docker использует монтирование тома при сборке Docker из исходного кода.

ОБСУЖДЕНИЕ

Может показаться странным, что такой старый инструмент появился в книге, где идет речь о сравнительно новой технологии Docker. Почему бы не использовать более новую технологию сборки, такую как Ant, Maven или любой из множества других доступных инструментов сборки.

Ответ заключается в том, что при всех своих недостатках `make` – это инструмент, который:

- вряд ли исчезнет в ближайшее время;
- хорошо задокументирован;
- очень гибкий;
- широкодоступный.

Когда мы тратили много часов на борьбу с ошибками или плохо документированными (или недокументированными) ограничениями новых технологий сборки или пытались установить зависимости этих систем, функции `make` не раз спасали нас. Также более вероятно, что `make` будет доступен и через пять лет, когда другие инструменты исчезнут или не будут обслуживаться их владельцами.

МЕТОД 55 Создание образов с помощью Chef Solo

Одна из вещей, которая смущает новичков в Docker, – являются ли файлы Dockerfile единственным поддерживаемым инструментом управления конфигурациями и следует ли переносить существующие инструменты

управления конфигурациями в файлы Dockerfile. Ни то, ни другое не является правдой.

Несмотря на то что файлы Dockerfile разработаны как простое и портативное средство подготовки образов, они также достаточно гибки, чтобы позволить любому другому инструменту управления конфигурацией брать руководство на себя. Говоря кратко, если вы можете запустить его в терминале, вы запустите его в Dockerfile.

В качестве демонстрации мы покажем вам, как начать работу с Chef, возможно, наиболее признанным средством управления конфигурацией в Dockerfile. Использование такого инструмента, как Chef, может уменьшить объем работы, необходимой для настройки образов.

> **ПРИМЕЧАНИЕ.** Хотя знакомство с Chef не обязательно, для того чтобы следовать этому методу, какое-то знакомство все же необходимо, чтобы с легкостью довериться ему в первый раз. Полный рассказ об инструменте управления конфигурацией – это сама по себе книга. Благодаря тщательному изучению и исследованиям данный метод может быть использован для хорошего понимания основ Chef.

ПРОБЛЕМА

Вы хотите уменьшить объем работ по настройке с помощью Chef.

РЕШЕНИЕ

Установите Chef в свой контейнер и запустите рецепты, используя Chef Solo в этом контейнере, чтобы подготовить его. Все это делается в файле Dockerfile.

Мы подготовим простой веб-сайт Apache Hello World. Это даст вам представление о том, что Chef может сделать для вашей конфигурации.

Chef Solo не требует настройки внешнего сервера Chef. Если вы уже знакомы с Chef, этот пример можно легко адаптировать, чтобы позволить своим уже существующим сценариям вступать в контакт с сервером Chef, если пожелаете.

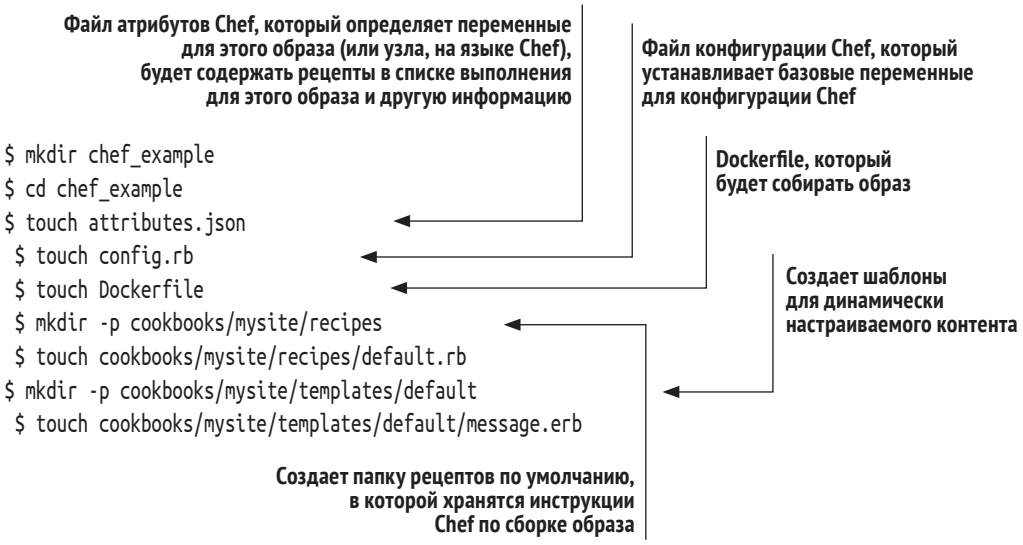
Мы пройдем создание этого примера, но, если вы хотите загрузить рабочий код, он доступен в виде Git-репозитория. Чтобы скачать его, выполните эту команду:

```
git clone\ https://github.com/docker-in-practice/docker-chef-solo-example.git
```

Начнем с простой цели – настройки веб-сервера с помощью Apache, который выдает «Hello World!» (а что же еще?), когда вы обращаетесь к нему. Сайт будет обслуживаться с mysite.com, а на образе будет настроен пользователь `mysiteuser`.

Для начала создайте каталог и настройте его с помощью файлов, которые понадобятся вам для настройки Chef.

Листинг 7.11. Создание необходимых файлов для конфигурации Chef



Сначала мы заполним файл attribute.json.

Листинг 7.12. attribute.json

```
{
        "run_list": [
                "recipe[apache2::default]",
                "recipe[mysite::default]"
        ]
}
```

Этот файл содержит рецепты, которые вы будете запускать. Рецепты apache2 получены из общедоступного репозитория, рецепты mysite написаны здесь.

Затем заполните файл config.rb базовой информацией, как показано в следующем листинге.

Листинг 7.13. config.rb

```
base_dir        "/chef/"
file_cache_path base_dir + "cache/"
cookbook_path   base_dir + "cookbooks/"
verify_api_cert true
```

Этот файл устанавливает основную информацию о местоположении и добавляет параметр конфигурации `verify_api_cert` для подавления несущественной ошибки.

Теперь мы добрались до сути: рецепта образа. Каждый раздел, заканчивающийся словом end в блоке кода, определяет ресурс Chef.

Листинг 7.14. cookbooks/mysite/recipes/default.rb

```
user "mysiteuser" do
    comment "mysite user"
    home "/home/mysiteuser"
    shell "/bin/bash"
end

directory "/var/www/html/mysite" do
    owner "mysiteuser"
    group "mysiteuser"
    mode 0755
    action :create
end

template "/var/www/html/mysite/index.html" do
    source "message.erb"
    variables(
        :message => "Hello World!"
    )
    user "mysiteuser"
    group "mysiteuser"
    mode 0755
end

web_app "mysite" do
    server_name "mysite.com"
    server_aliases ["www.mysite.com","mysite.com"]
    docroot "/var/www/html/mysite"
    cookbook 'apache2'
end
```

Создает пользователя

Создает каталог для веб-контента

Определяет файл, который будет помещен в веб-папку. Этот файл будет создан из шаблона, определенного в атрибуте «source»

Определяет веб-приложение для apache2

В реальном сценарии вам бы пришлось изменить ссылки с mysite на имя своего сайта. Если вы получаете доступ или выполняете тестирование со своего хоста, это не важно

Содержимое сайта находится в файле шаблона. Он содержит одну строку, которую Chef будет читать, подставляя в сообщение «Hello World!» из файла config.rb. Затем Chef запишет замещенный файл в цель шаблона (/var/www/html/mysite/index.html). Тут используется шаблонный язык, который мы не будем здесь освещать.

Листинг 7.15. cookbooks/mysite/templates/default/message.erb

```
<%= @message %>
```

Наконец, вы соединяете все вместе с Dockerfile, который устанавливает предварительные требования и запускает Chef для настройки образа, как показано в следующем листинге.

Листинг 7.16. Dockerfile

```
FROM ubuntu:14.04

RUN apt-get update && apt-get install -y git curl

RUN curl -L \
https://opscode-omnibus-packages.s3.amazonaws.com/ubuntu/12.04/x86_64
➥ /chefdk_0.3.5-1_amd64.deb \
-o chef.deb
RUN dpkg -i chef.deb && rm chef.deb

COPY . /chef

WORKDIR /chef/cookbooks
RUN knife cookbook site download apache2
RUN knife cookbook site download iptables
RUN knife cookbook site download logrotate

RUN /bin/bash -c 'for f in $(ls *gz); do tar -zxf $f; rm $f; done'

RUN chef-solo -c /chef/config.rb -j /chef/attributes.json

CMD /usr/sbin/service apache2 start && sleep infinity
```

Скачивает и устанавливает Chef. Если эта загрузка у вас не работает, проверьте последний код в упомянутом ранее docker-chef-solo-example в этом обсуждении, поскольку теперь может потребоваться более поздняя версия Chef

Копирует содержимое рабочей папки в папку /chef в образе

Перемещается в папку cookbooks и скачивает apache2 и ее зависимости в виде TAR-архивов с помощью утилиты knife

Извлекает скачанные TAR-архивы и удаляет их

Определяет команду по умолчанию для образа. Команда sleep infinity гарантирует, что контейнер не завершит работу сразу после того, как это сделает команда service

Выполняет команду chef для настройки вашего образа. Снабжает его атрибутами и конфигурационными файлами, которые вы уже создали

Теперь вы готовы собрать и запустить \image:

```
docker build -t chef-example .
docker run -ti -p 8080:80 chef-example
```

Если вы сейчас перейдете по адресу http://localhost:8080, то должны увидеть сообщение «Hello World!».

> **ВНИМАНИЕ!** Если сборка Chef занимает много времени и вы используете рабочий процесс Docker Hub, сборка может быть приостановлена. Если это произойдет, вы можете выполнить сборку на компьютере, которым управляете, заплатить за поддерживаемый сервис или разбить этапы сборки на более мелкие куски, чтобы каждый отдельный шаг в файле Docker возвращался быстрее.

Хотя это тривиальный пример, преимущества использования такого подхода должны быть очевидны. Имея относительно простые файлы конфигурации, заботу о деталях получения образа в желаемом состоянии берет на себя инструмент управления конфигурацией. Это не значит, что можно забыть о деталях конфигурации; изменение значений потребует от вас понимания семантики, чтобы вы ничего не сломали. Но этот подход может сэкономить вам много времени и усилий, особенно в проектах, где не нужно слишком вдаваться в детали.

ОБСУЖДЕНИЕ

Цель этого метода – исправить общую путаницу в концепции Dockerfile, в частности, касательно того, что он является конкурентом другим инструментам управления конфигурацией, таким как Chef и Ansible.

На самом деле Docker (как мы говорим в другом месте книги) – это инструмент *упаковки*. Он позволяет вам представлять результаты процесса сборки в предсказуемой и упакованной форме. Как вы решите выполнять сборку, зависит от вас. Можете использовать Chef, Puppet, Ansible, Makefiles, сценарии оболочки или создавать их вручную.

Причина, по которой большинство людей не используют Chef, Puppet и тому подобное при создании образов, заключается в основном в том, что образы Docker, как правило, создаются как одноцелевые и однопроцессные инструменты. Но если у вас уже есть под рукой сценарии конфигурации, почему бы не использовать их повторно?

7.3. Маленький значит красивый

Если вы создаете множество образов и отправляете их туда-сюда, с большой вероятностью возникнет проблема размера. Несмотря на то что Docker может использовать слои, в вашем распоряжении может быть такое количество образов, что управлять ими будет непрактично.

В этих случаях может быть полезно иметь в своей организации передовые наработки, связанные с уменьшением образов до максимально возможного размера. В этом разделе мы покажем вам некоторые из них и даже то, как стандартный образ утилиты может быть уменьшен на порядок величины – гораздо меньший объект.

МЕТОД 56 Хитрости, позволяющие уменьшить образ

Допустим, третье лицо предоставило вам образ, и вы хотите уменьшить его. Самый простой подход – начать с работающего образа и удалить ненужные файлы.

Классические инструменты управления конфигурацией, как правило, не выполняют удаление, если явно не указано иное: вместо этого они запускаются из нерабочего состояния и добавляют новые конфигурации и файлы. Это приводит к появлению систем *снежинок,* созданных для определенной цели, что может сильно отличаться от того, что вы получили бы, если бы запускали инструмент управления конфигурацией на новом сервере, особенно в случае развития конфигурации с течением времени. Благодаря концепции слоев и легким образам в Docker вы можете выполнить этот процесс в обратном порядке и поэкспериментировать с удалением.

ПРОБЛЕМА

Вы хотите уменьшить образ.

РЕШЕНИЕ

Следуйте этим шагам, чтобы уменьшить размер образа, удалив ненужные пакеты и файлы документов:

1. Запустите образ.
2. Войдите в контейнер.
3. Удалите ненужные файлы.
4. Зафиксируйте контейнер как новый образ (см. метод 15).
5. Выполните «сращивание» образа (см. метод 52).

Последние два шага были рассмотрены ранее, поэтому мы разберем здесь только первые три.

Чтобы проиллюстрировать, как это сделать, возьмем образ, созданный в методе 49, и попытаемся уменьшить его.

Сначала запустите образ как контейнер:

```
docker run -ti --name smaller --entrypoint /bin/bash \
dockerinpractice/log-cleaner
```

Поскольку это образ на основе Debian, вы можете начать с просмотра пакетов, которые вам могут не понадобиться, и их удаления. Выполните команду `dpkg -l | awk '{print $ 2}'` и получите список пакетов, установленных в системе.

Затем вы можете просмотреть пакеты, выполнив команду `apt-get purge -y package_name`. Если появляется страшное сообщение, предупреждающее вас о том, что «вы собираетесь сделать что-то потенциально опасное», нажмите **Return** для продолжения.

После того как вы удалили все пакеты, которые можно безопасно удалить, выполните эти команды, чтобы очистить кеш apt:

```
apt-get autoremove
apt-get clean
```

Это относительно безопасный способ уменьшить пространство в ваших образах.

Дополнительная значительная экономия может быть достигнута за счет удаления документов. Например, при выполнении команды `rm -rf /usr/share/doc/* /usr/share/man/* /usr/share/info/*` часто удаляются файлы большого размера, которые вам, вероятно, никогда не понадобятся. Можете перейти на следующий уровень, вручную выполнив команду `rm` для двоичных файлов и библиотек, которые вам не нужны.

Другой областью для богатого выбора является папка /var, которая должна содержать временные данные или данные, несущественные для запуска программ.

Эта команда избавит вас от всех файлов с суффиксом .log:

```
find /var | grep '\.log$' | xargs rm -v
```

Теперь у вас будет образ гораздо меньшего размера, чем раньше, готовый к фиксации.

ОБСУЖДЕНИЕ

Используя этот несколько ручной процесс, вы вполне легко можете получить исходный образ dockerinpractice / log cleaner, уменьшенный до нескольких десятков Мб, и даже сделать его меньше, если у вас есть мотивация. Помните, что из-за концепции слоев, существующей в Docker, вам нужно экспортировать и импортировать образ, как описано в методе 52; в противном случае его общий размер будет включать в себя удаленные файлы.

Метод 59 покажет вам гораздо более эффективный (но рискованный) способ значительно уменьшить размер ваших образов.

> **СОВЕТ.** Пример описанных здесь команд находится по адресу https://github.com/docker-in-practice/log-cleaner-purged, и его можно извлечь с помощью Docker из dockerinpractice /log-cleaner-purged.

МЕТОД 57 Создание маленьких образов Docker с помощью BusyBox и Alpine

Небольшие, удобные в использовании ОС, которые могут быть встроены в маломощный или дешевый компьютер, существуют с момента появления Linux. К счастью, усилия этих проектов были перенаправлены на создание небольших образов Docker, чтобы использовать их там, где важен размер образа.

ПРОБЛЕМА

Вам нужен маленький, функциональный образ.

РЕШЕНИЕ

Используйте небольшой базовый образ, например BusyBox или Alpine, для создания собственных.

Это еще одна область, в которой состояние дел быстро меняется. Два популярных варианта минимальных базовых образов Linux – это BusyBox и Alpine, каждый из которых имеет свои характеристики.

Если ваша цель – без излишеств, но с пользой, BusyBox может вам помочь. Если вы запустите образ BusyBox с помощью следующей команды, произойдет нечто удивительное:

```
$ docker run -ti busybox /bin/bash
exec: "/bin/bash": stat /bin/bash: no such file or directory2015/02/23 >
09:55:38 Error response from daemon: Cannot start container >
73f45e34145647cd1996ae29d8028e7b06d514d0d32dec9a68ce9428446faa19: exec: >
"/bin/bash": stat /bin/bash: no such file or directory
```

BusyBox настолько скуден, что в нем нет bash! Вместо этого он использует ash, которая представляет собой posix-совместимую оболочку – фактически ограниченную версию более продвинутых оболочек, таких как bash и ksh.

```
$ docker run -ti busybox /bin/ash
/ #
```

В результате многих подобных решений образ BusyBox весит менее 2,5 Мб.

> **ВНИМАНИЕ!** BusyBox может содержать другие неприятные сюрпризы. Например, в версии `tar` будет сложно распаковать TAR-файлы, созданные с помощью GNU tar.

Это замечательно, если вы хотите написать небольшой сценарий, который требует только простых инструментов, но если вы хотите запустить что-то еще, вам придется установить его самостоятельно. BusyBox поставляется без управления пакетами.

Другие авторы добавили в BusyBox функциональность управления пакетами. Например, progrium/busybox, возможно, не самый маленький контейнер BusyBox (в настоящее время он составляет менее 5 Мб), но у него есть opkg, а это означает, что вы можете легко устанавливать другие распространенные пакеты, сохраняя размер образа на абсолютном минимуме. Например, если вы не используете bash, можете установить его следующим образом:

```
$ docker run -ti progrium/busybox /bin/ash
/ # opkg-install bash > /dev/null
/ # bash
bash-4.3#
```

После фиксации размер образа составляет 6 Мб.

Еще один интересный образ Docker (который стал стандартом Docker для небольших образов) – это gliderlabs/alpine. Он похож на BusyBox, но имеет более широкий спектр пакетов, который вы можете посмотреть на странице https://pkgs.alpinelinux.org/packages.

Пакеты разработаны так, чтобы быть легкими при установке. Для конкретного примера возьмем файл Dockerfile, образ которого составляет чуть более четверти гигабайта.

Листинг 7.17. Ubuntu плюс mysql-client

```
FROM ubuntu:14.04
RUN apt-get update -q \
&& DEBIAN_FRONTEND=noninteractive apt-get install -qy mysql-client \
&& apt-get clean && rm -rf /var/lib/apt
ENTRYPOINT ["mysql"]
```

> **СОВЕТ.** `DEBIAN_FRONTEND=noninteractive` перед `apt-get install` гарантирует, что при установке не запрашивается ввод данных. Поскольку вы не можете с легкостью спроектировать ответы на вопросы при запуске команд, это часто полезно при работе с файлами Dockerfile.

Напротив, следующий листинг дает образ размером чуть более 36 Мб.

Листинг 7.18. Alpine plus mysql-client

```
FROM gliderlabs/alpine:3.6
RUN apk-install mysql-client
ENTRYPOINT ["mysql"]
```

ОБСУЖДЕНИЕ

Это та область, в которой за последние пару лет произошло много изменений.

Базовый образ Alpine вытеснил BusyBox и стал чем-то вроде стандарта Docker благодаря некоторой поддержке самой Docker Inc.

Кроме того, другие более «стандартные» базовые образы сидели на диете.

Образ Debian составлял примерно 100 Мб, когда мы готовили второе издание этой книги, – гораздо меньше, чем было изначально.

Здесь стоит упомянуть один момент: существует много дискуссий по поводу того, как уменьшить размер образов или использовать базовые образы меньшего размера, в то время как решать нужно не это. Помните, что зачастую лучше потратить время и усилия на преодоление существующих узких мест, чем достичь теоретических преимуществ, которые могут принести небольшую отдачу.

МЕТОД 58 Модель минимальных контейнеров Go

Хотя сокращать размер рабочих контейнеров путем удаления избыточных файлов может быть полезно, существует еще один вариант – компиляция минимальных двоичных файлов без зависимостей.

Это существенно упрощает задачу управления конфигурацией: если нужно развернуть только один файл и пакеты не требуются, значительное количество инструментов управления конфигурацией становится избыточным.

ПРОБЛЕМА

Вы хотите создать двоичные образы Docker без внешних зависимостей.

РЕШЕНИЕ

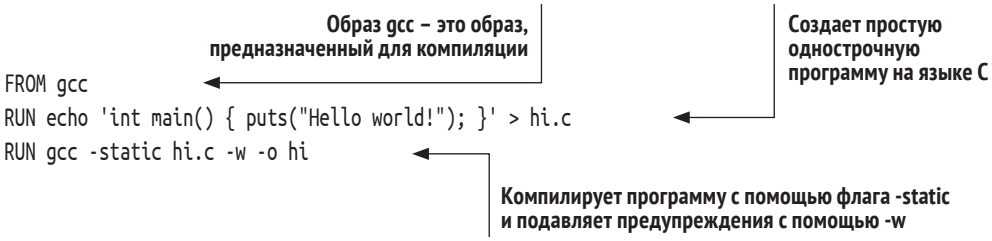
Создайте статически связанный двоичный файл, который не будет пытаться загрузить какие-либо системные библиотеки при запуске.

Чтобы продемонстрировать, как это может быть полезно, мы сначала создадим небольшой образ Hello World с помощью маленькой программы на языке C. Потом мы продолжим, чтобы показать вам, как сделать что-то эквивалентное для более полезного применения.

Минимальный двоичный файл Hello World

Чтобы создать минимальный двоичный файл Hello World, сначала создайте новый каталог и файл Dockerfile, как показано в следующем листинге.

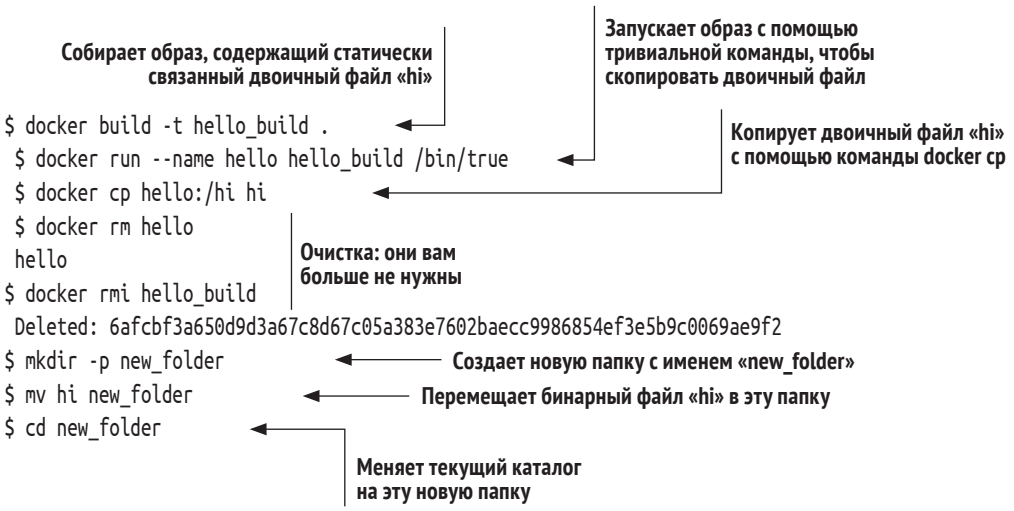
Листинг 7.19. Dockerfile Hello



Предыдущий Dockerfile компилирует простую программу Hello World без зависимостей.

Теперь вы можете собрать его и извлечь этот двоичный файл из контейнера, как показано в следующем листинге.

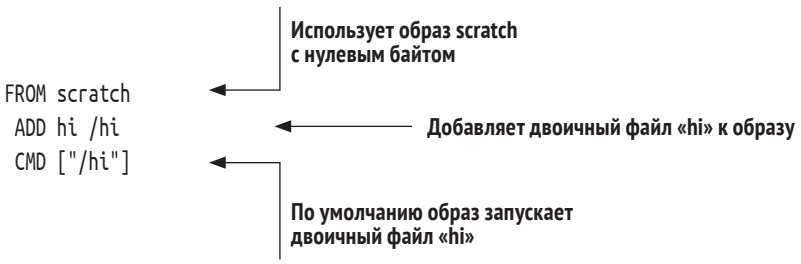
Листинг 7.20. Извлечение двоичного файла из образа



Теперь у вас есть статически собранный двоичный файл в новом каталоге, и вы перешли в него.

Создайте еще один Dockerfile, как показано в следующем листинге.

Листинг 7.21. Минимальный Dockerfile Hello



Создайте и запустите его, как показано в следующем листинге.

Листинг 7.22. Создание минимального контейнера

```
$ docker build -t hello_world .
Sending build context to Docker daemon 931.3 kB
Sending build context to Docker daemon
```

```
Step 0 : FROM scratch
 --->
Step 1 : ADD hi /hi
 ---> 2fe834f724f8
Removing intermediate container 01f73ea277fb
Step 2 : ENTRYPOINT /hi
 ---> Running in 045e32673c7f
 ---> 5f8802ae5443
Removing intermediate container 045e32673c7f
Successfully built 5f8802ae5443
$ docker run hello_world
Hello world!
$ docker images | grep hello_world
hello_world      latest      5f8802ae5443      24 seconds ago      928.3 kB
```

Образ собирается, запускается и весит менее 1 Мб.

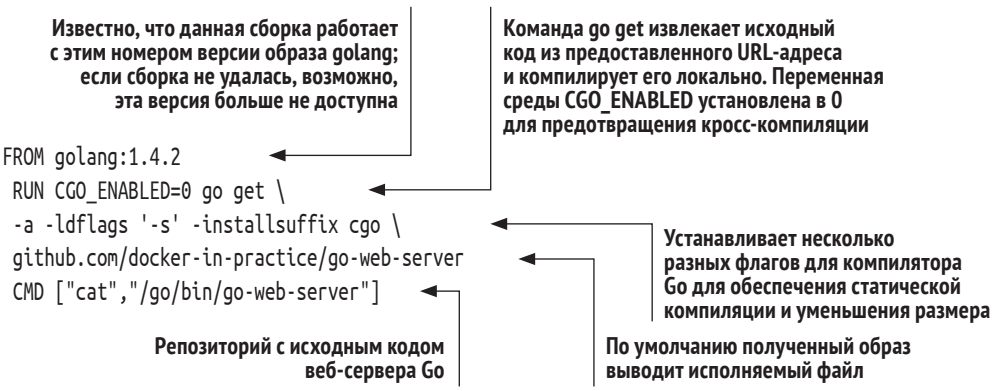
Минимальный образ веб-сервера в GO

Это был довольно простой пример, но вы можете применить тот же принцип к программам, созданным в Go. Интересной особенностью языка Go является то, что такие статические двоичные файлы относительно легко создавать.

Чтобы продемонстрировать эту возможность, мы создали простой веб-сервер в Go, код которого доступен по адресу https://github.com/docker-in-practice/go-web-server.

Dockerfile для создания этого простого веб-сервера показан в следующем листинге.

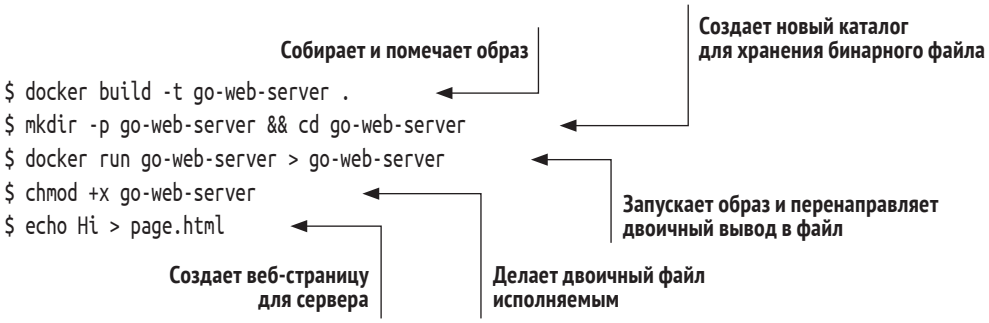
Листинг 7.23. Dockerfile для статической компиляции веб-сервера Go



Если вы сохраните этот Dockerfile в пустой каталог и соберете его, у вас теперь будет образ, содержащий программу. Поскольку вы указали команду образа по умолчанию для вывода исполняемого содержимого, теперь просто

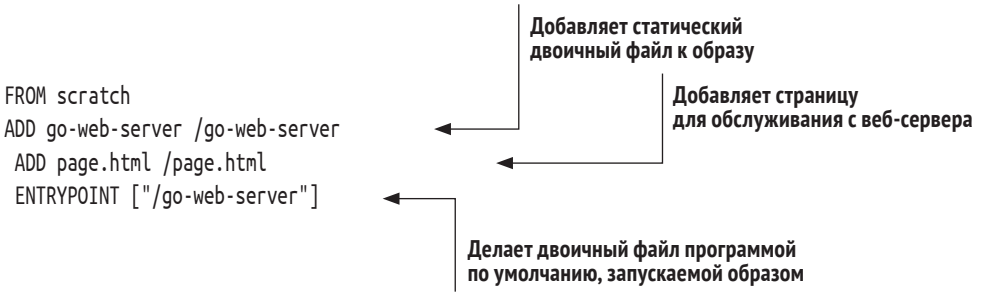
нужно запустить образ и отправить вывод в файл на вашем хосте, как показано в следующем листинге.

Листинг 7.24. Получение веб-сервера Go из образа



Теперь, как и в случае с двоичным файлом «hi», у вас есть двоичный файл без библиотечных зависимостей или необходимости доступа к файловой системе. Поэтому мы создадим файл Dockerfile из образа `scratch` с нулевым байтом и добавим в него двоичный файл, как и раньше.

Листинг 7.25. Файл go web server



Теперь соберите его и запустите образ. Полученный образ – размером чуть более 4 Мб.

Листинг 7.26. Сборка и запуск образа go web server

```
$ docker build -t go-web-server .
$ docker images | grep go-web-server
go-web-server    latest    de1187ee87f3    3 seconds ago    4.156 MB
$ docker run -p 8080:8080 go-web-server -port 8080
```

Вы можете получить к нему доступ по адресу http://localhost:8080. Если порт уже используется, замените 8080-е порты в предыдущем коде на порт по вашему выбору.

ОБСУЖДЕНИЕ

Если вы можете связывать приложения в один двоичный файл, зачем вообще возиться с Docker? Перемещайте двоичный файл, запускайте несколько копий и т. д.

Вы можете делать это, если хотите, но вы лишитесь:

- всех инструментов управления контейнерами в экосистеме Docker;
- метаданных в образах Docker, которые документируют важную информацию о приложении, такую как порты, тома и метки;
- изоляции, которая обеспечивает Docker оперативную мощь.

В качестве конкретного примера etcd по умолчанию является статическим двоичным файлом, но когда мы рассмотрим его в методе 74, то продемонстрируем его внутри контейнера, чтобы было легче увидеть, как один и тот же процесс будет работать на нескольких компьютерах, и упростить развертывание.

МЕТОД 59 Использование inotifywait для сокращения размера контейнеров

Теперь перейдем на новый уровень, используя отличный инструмент, который сообщает, к каким файлам выполняется обращение, когда мы запускаем контейнер.

Его можно назвать атомным вариантом, поскольку его реализация для промышленного применения может быть довольно рискованной. Но это, возможно, станет поучительным средством изучения вашей системы, даже если вы не будете использовать его по-настоящему, – важная часть управления конфигурацией понимает, что нужно вашему приложению для правильной работы.

ПРОБЛЕМА

Вы хотите уменьшить свой контейнер до минимально возможного набора файлов и прав доступа.

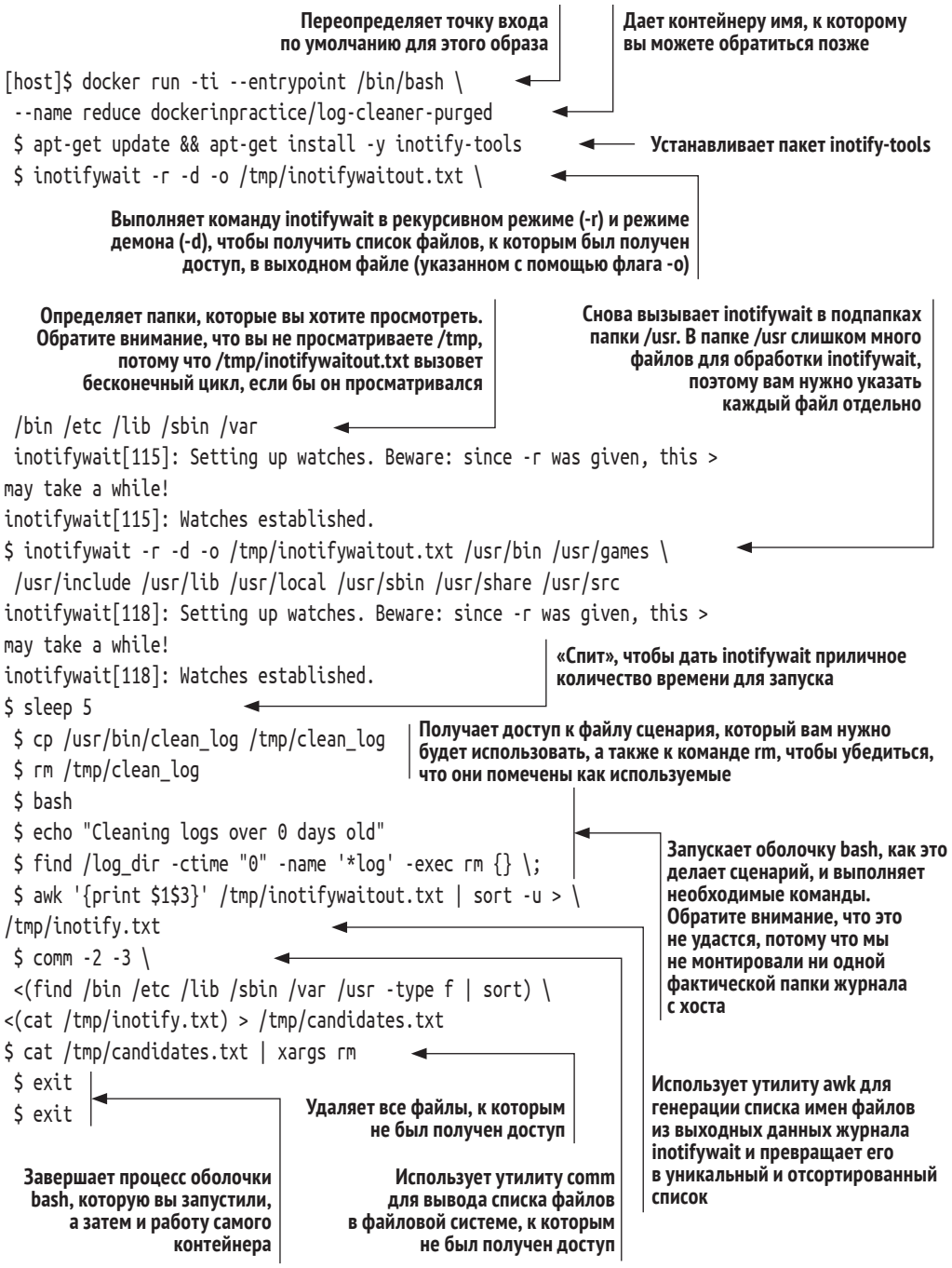
РЕШЕНИЕ

Отслеживайте, к каким файлам получает доступ ваша программа с помощью inotify, а затем удаляйте все файлы, которые окажутся неиспользованными.

На высоком уровне вам нужно знать, к каким файлам идет доступ при выполнении команды в контейнере. Если вы удалите все другие файлы в файловой системе контейнера, теоретически у вас останется все, что нужно.

В этом пошаговом руководстве мы будем использовать образ из метода 56. Вы установите inotify-tools, а затем выполните команду `inotifywait`, чтобы получить отчет о том, к каким файлам был получен доступ. Затем запустите симуляцию точки входа образа (сценарий log_clean). После этого, используя сгенерированный файл отчета, вы удалите любой файл, к которому не был получен доступ.

Листинг 7.27. Выполнение шагов ручной установки при мониторинге с помощью inotifywait

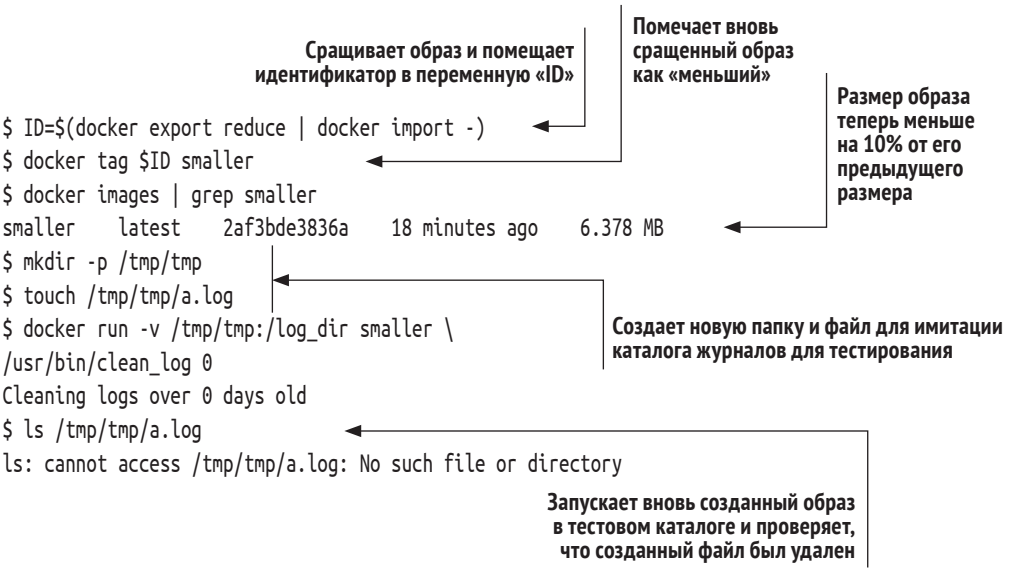


На данный момент вы:

- разместили файлы, чтобы увидеть, к каким файлам есть доступ;
- выполнили все команды для имитации запуска сценария;
- выполнили команды, чтобы обеспечить доступ к необходимому сценарию и утилите rm;
- получили список всех файлов, к которым не было доступа во время запуска;
- удалили все файлы, к которым не был получен доступ.

Теперь вы можете «срастить» этот контейнер (см. метод 52), чтобы создать новый образ и проверить, что он все еще работает.

Листинг 7.28. Сращивание образа и его запуск



Мы сократили размер этого образа с 96 Мб до примерно 6,5 Мб, и он все еще работает. Вот это экономия!

> **ВНИМАНИЕ!** Данный метод, как и разгон вашего процессора, не является оптимизацией для излишне доверчивых. Этот конкретный пример хорошо работает, потому что это приложение, которое весьма ограничено в объеме, но ваше критически важное бизнес-приложение, вероятно, будет более сложным и динамичным в плане получения доступа к файлам. Вы можете легко удалить файл, к которому не было доступа при запуске, но в некоторых случаях это необходимо.

Если вы немного нервничаете из-за возможного повреждения образа, удаляя файлы, которые понадобятся позже, можете использовать /tmp/candidates.txt, чтобы получить список самых больших файлов, которые не были затронуты, например:

```
cat /tmp/candidates.txt | xargs wc -c | sort -n | tail
```

Тогда вы можете удалить файлы большего размера, которые, как вы уверены, не понадобятся вашему приложению. Здесь тоже могут быть большие победы.

ОБСУЖДЕНИЕ

Хотя этот метод был представлен как метод Docker, он попадает в категорию «обычно полезных» методов, которые могут применяться в других контекстах. Это особенно полезно при отладке процессов, когда вы не совсем понимаете, что происходит, и хотите увидеть, к каким файлам обращаются. `strace` – еще один вариант, который можно использовать, но `inotifywait` в некоторых отношениях является более простым инструментом для этой цели.

Этот общий подход также применяется в качестве одной из возможностей атаки в методе 97 в контексте уменьшения поверхности атаки контейнера.

МЕТОД 60 Большое может быть красивым

Хотя этот раздел посвящен тому, как сделать образы маленькими, стоит помнить, что меньше не значит лучше. Как мы обсудим, относительно большой монолитный образ может быть эффективнее, чем маленький.

ПРОБЛЕМА

Вы хотите уменьшить использование дискового пространства и пропускную способность сети из-за образов Docker.

РЕШЕНИЕ

Создайте универсальный, большой, монолитный базовый образ для вашей организации.

Парадоксально, но большой монолитный образ может сэкономить вам дисковое пространство и пропускную способность сети.

Напомним, что Docker использует механизм копирования при записи, когда его контейнеры работают. Это означает, что у вас могут быть запущены сотни контейнеров Ubuntu, но для каждого из них используется только небольшое количество дополнительного дискового пространства.

Если на вашем сервере Docker много разных, меньших по размеру образов, как на рис. 7.1, может быть использовано больше дискового пространства, чем при наличии одного большего монолитного образа со всем необходимым.

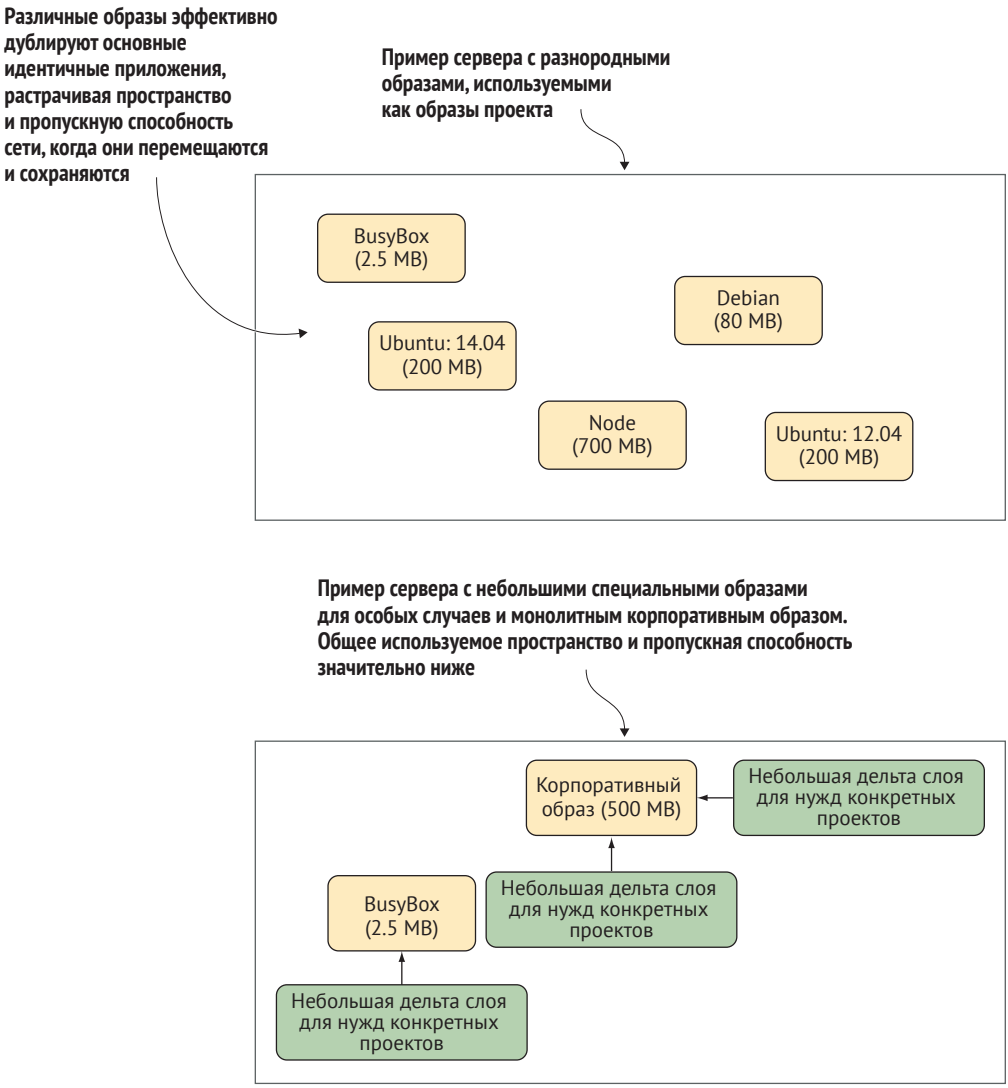


Рис. 7.1 ❖ Сравнение большого числа маленьких базовых образов и маленького количества больших базовых образов

Возможно, вы помните о принципе общей библиотеки. Совместно используемая библиотека может загружаться несколькими приложениями одновременно, уменьшая объем диска и памяти, необходимых для запуска требуемых программ. Таким же образом общий базовый образ для вашей организации может сэкономить место, так как его нужно скачать только один раз, и в нем должно быть все, что вам нужно. Программы и библиотеки, которые ранее требовались для нескольких образов, теперь требуются только один раз.

Кроме того, могут быть и другие преимущества совместного использования монолитного, централизованно управляемого образа в группах. Поддержка этого образа может быть централизованной, улучшения можно использовать совместно, а проблемы со сборкой должны быть решены только один раз.

Если вы собираетесь использовать этот метод, вот некоторые вещи, на которые стоит обратить внимание:

- базовый образ должен быть в первую очередь надежным. Если он не будет работать согласованно, пользователи не станут его использовать;
- изменения в базовом образе должны где-то отслеживаться, чтобы это можно было увидеть и чтобы пользователи могли сами отлаживать проблемы;
- регрессионные тесты необходимы для уменьшения путаницы при обновлении ванильного образа;
- будьте осторожны с тем, что вы добавляете в основу – после того как это добавлено в базовый образ, его трудно удалить, и образ может быстро увеличиться в размерах.

ОБСУЖДЕНИЕ

Мы использовали этот метод, чтобы добиться успеха в нашей компании-разработчике численностью 600 человек. Месячная сборка основных приложений была объединена в большой образ и опубликована во внутреннем реестре Docker. По умолчанию команды разработчиков будут основываться на так называемом корпоративном образе «ваниль» и, если необходимо, создавать специальные слои поверх него.

Стоит взглянуть на метод 12 для получения дополнительных сведений о монолитных контейнерах, особенно об образе phusion/base image, разработанного с учетом выполнения нескольких процессов.

Резюме

- `ENTRYPOINT` – еще один способ запуска контейнеров Docker, который позволяет настроить параметры среды выполнения.
- Сращивание образов можно использовать для предотвращения утечки секретов из вашей сборки через слои образа.
- Пакеты, чуждые дистрибутиву выбранного вами базового образа, могут быть интегрированы с помощью Alien.
- Традиционные инструменты сборки, такие как make, а также современные инструменты, такие как Chef, все еще играют свою роль в мире Docker.
- Образы Docker можно уменьшить, используя базовые образы меньшего размера, языки, соответствующие задаче, или удалив ненужные файлы
- Стоит подумать, является ли размер вашего образа наиболее важной задачей, которую нужно решить.

Часть 3

Docker и DevOps

Теперь вы готовы вывести Docker за пределы своей среды разработки и начать использовать его на других этапах поставки программного обеспечения. Автоматизация сборки и тестирования является краеугольным камнем движения DevOps. Мы продемонстрируем мощь Docker с помощью автоматизации жизненного цикла доставки программного обеспечения, развертываний и реалистичного тестирования среды.

Глава 8 покажет различные методы для обеспечения и улучшения непрерывной интеграции, делая поставки программного обеспечения более надежными и масштабируемыми.

Глава 9 посвящена непрерывной доставке. Мы объясним, что такое непрерывная доставка, и рассмотрим, как использовать Docker для улучшения этого аспекта своего конвейера разработки.

В главе 10 показано, как можно в полном объеме использовать модель сетевой среды Docker, формируя мультиконтейнерные сервисы, моделируя реалистичные сети и создавая сети по требованию.

Эта часть ведет вас от разработки вплоть до момента, когда вы можете задуматься о запуске Docker для промышленного применения.

Глава 8

Непрерывная интеграция: ускорение конвейера разработки

О чем рассказывается в этой главе:

- использование рабочего процесса Docker Hub в качестве инструмента непрерывной интеграции;
- ускорение сборок с интенсивным вводом-выводом;
- использование Selenium для автоматического тестирования;
- запуск Jenkins в Docker;
- использование Docker в качестве ведомого устройства Jenkins;
- масштабирование процесса непрерывной интеграции в соответствии с темпами разработки.

В этой главе мы рассмотрим различные методы, которые будут использовать Docker для активизации и улучшения усилий по непрерывной интеграции.

К настоящему времени вы должны понимать, насколько хорошо Docker подходит для автоматизации. Его легкий характер и мощь, которую он дает для переноса сред из одного места в другое, могут сделать его ключевым фактором непрерывной интеграции. Мы обнаружили, что методы, описанные в этой главе, неоценимы для того, чтобы осуществить процесс непрерывной интеграции в рамках компании.

К концу главы вы поймете, как Docker может сделать этот процесс более быстрым, стабильным и воспроизводимым. Используя инструменты тестирования, такие как Selenium, и расширяя возможности сборки с помощью плагина

Jenkins Swarm, вы увидите, как Docker может помочь вам получить еще больше от процесса непрерывной интеграции.

> **ПРИМЕЧАНИЕ.** Если вы не знаете, *непрерывная интеграция* – это стратегия жизненного цикла программного обеспечения, используемая для ускорения процесса разработки. Автоматически перезапуская тесты каждый раз, когда в кодовую базу вносятся существенные изменения, вы получаете более быструю и стабильную поставку, поскольку в поставляемом программном обеспечении есть базовый уровень стабильности.

8.1. Автоматические сборки Docker Hub

Функция автоматической сборки Docker Hub упоминалась в методе 10, хотя мы не вдавались в подробности. Говоря кратко, если вы укажете на Git-репозиторий, содержащий Dockerfile, Docker Hub будет обрабатывать процесс создания образа, делая его доступным для скачивания. Повторная сборка образа будет инициироваться при любых изменениях в Git-репозитории, что превращает его в весьма полезную часть процесса непрерывной интеграции.

МЕТОД 61 Использование рабочего процесса Docker Hub

Этот метод познакомит вас с рабочим процессом Docker Hub, который позволяет инициировать повторную сборку ваших образов.

> **ПРИМЕЧАНИЕ.** Для этого раздела вам потребуется учетная запись на docker.com, связанная с учетной записью GitHub или Bitbucket. Если вы еще не настроили и не связали их, на страницах github.com и bitbucket.org доступны инструкции.

ПРОБЛЕМА

Вы хотите автоматически проверять и разместить (push) изменения при изменении кода.

РЕШЕНИЕ

Настройте репозиторий Docker Hub и свяжите его со своим кодом.
Хотя сборка Docker Hub не сложна, необходимо выполнить ряд шагов:

1. Создайте свой репозиторий на GitHub или BitBucket.
2. Клонируйте новый репозиторий Git.
3. Добавьте код в свой репозиторий.
4. Зафиксируйте изменения.
5. Поместите данные в Git-репозиторий.
6. Создайте новый репозиторий в Docker Hub.

7. Свяжите репозиторий Docker Hub с репозиторием Git.
8. Дождитесь завершения сборки Docker Hub.
9. Зафиксируйте и разместите (push) изменения.
10. Дождитесь завершения второй сборки Docker Hub.

> **ПРИМЕЧАНИЕ.** Как Git, так и Docker используют термин «репозиторий» для обозначения проекта. Это может вызвать путаницу. Git-репозиторий и репозиторий Docker – это не одно и то же, хотя здесь мы связываем два типа репозиториев.

Создайте свой репозиторий на GitHub или BitBucket

Создайте новый репозиторий на GitHub или Bitbucket. Можете дать ему любое имя.

Клонируйте новый репозиторий Git

Клонируйте ваш новый Git-репозиторий на свой хост-компьютер. Команда для этого будет доступна на домашней странице проекта Git.

Измените каталог в этот репозиторий.

Добавьте код в свой репозиторий

Теперь вам нужно добавить код в проект.

Можете добавить любой файл Dockerfile, который вам нравится, но в следующем листинге показан пример, доказавший свою эффективность. Он состоит из двух файлов, представляющих простую среду инструментов разработки; устанавливает несколько предпочтительных утилит и выводит версию bash, которая у вас есть.

Листинг 8.1. Dockerfile – простой контейнер инструментов разработчика

```
FROM ubuntu:14.04
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y nmap
RUN apt-get install -y socat
RUN apt-get install -y openssh-client
RUN apt-get install -y openssl
RUN apt-get install -y iotop
RUN apt-get install -y strace
RUN apt-get install -y tcpdump
RUN apt-get install -y lsof
RUN apt-get install -y inotify-tools
RUN apt-get install -y sysstat
RUN apt-get install -y build-essential
RUN echo "source /root/bash_extra" >> /root/.bashrc
```

Устанавливает полезные пакеты

Добавляет строку к корневому bashrc в исходный bash_extra

```
ADD bash_extra /root/bash_extra
CMD ["/bin/bash"]
```

Добавляет bash_extra из источника в контейнер

Теперь нужно будет добавить файл bash_extra, к которому вы обращались, и дать ему следующее содержимое:

```
bash --version
```

Этот файл только для иллюстрации. Он показывает, что вы можете создать bash-файл, который инициализируется при запуске. В этом случае он отображает версию bash, которую вы используете в своей оболочке, но может содержать всевозможные вещи, которые настраивают вашу оболочку в желаемое состояние.

Зафиксируйте изменения

Чтобы зафиксировать свой исходный код, используйте эту команду:

```
git commit -am "Initial commit"
```

Поместите данные в Git-репозиторий

Теперь вы можете поместить исходный код на сервер Git с помощью этой команды:

```
git push origin master
```

Создайте новый репозиторий в Docker Hub

Далее вам нужно создать репозиторий для этого проекта в Docker Hub. Перейдите на страницу https://hub.docker.com и убедитесь, что совершили вход. Затем нажмите **Создать** и выберите **Создать автоматическую сборку**.

Только в первый раз нужно будет пройти через процесс привязки аккаунта.

Вы увидите приглашение связать свою учетную запись с размещенным сервисом Git. Выберите свой сервис и следуйте инструкциям, чтобы привязать свой аккаунт. Вам может быть предложен вариант полного или более ограниченного доступа к Docker Inc. для интеграции. Если вы выбираете более ограниченный, следует прочитать официальную документацию для вашей конкретной службы, чтобы определить, какую дополнительную работу, возможно, понадобится проделать в ходе оставшихся этапов.

Свяжите репозиторий Docker Hub с репозиторием Git

Вы увидите экран с выбором служб Git. Выберите службу исходного кода, которую вы используете (GitHub или Bitbucket), и свой новый репозиторий из предоставленного списка.

Перед вами будет страница с параметрами конфигурации сборки. Можете оставить значения по умолчанию и нажать кнопку **Create Repository** (Создать репозиторий) внизу.

Дождитесь завершения сборки Docker Hub

Вы увидите страницу с сообщением, объясняющим, что ссылка работает. Нажмите на ссылку Build Details.

Далее будет страница, которая показывает детали сборок. Под историей сборок – запись для первой сборки. Если вы ничего не видите в списке, может понадобиться нажать кнопку, чтобы инициировать сборку вручную. В поле **Состояние** рядом с идентификатором сборки отображаются надписи: **Ожидание**, **Завершено**, **Выполняется сборка** или **Ошибка**. Если все хорошо, вы увидите одну из первых трех. Если это надпись **Ошибка**, значит, что-то пошло не так, и вам нужно нажать на идентификатор сборки, чтобы увидеть, в чем ошибка.

> **ПРИМЕЧАНИЕ.** Для начала сборки может потребоваться какое-то время, поэтому если в течение некоторого времени вы будете видеть надпись **Ожидание**, то это совершенно нормально.

Периодически нажимайте кнопку **Обновить**, пока не увидите, что сборка завершена. После этого вы можете извлечь образ с помощью команды `docker pull`, указанной в верхней части той же страницы.

Зафиксируйте и разместите изменения

Теперь вы приняли решение, что нужно больше информации о своей среде, когда входите в систему, поэтому вы хотите вывести детали дистрибутива, в котором работаете. Для этого добавьте эти строки в файл bash_extra, чтобы он теперь выглядел следующим образом:

```
bash --version
cat /etc/issue
```

Затем зафиксируйте изменения и разместите, как показано в шагах 4 и 5.

Дождитесь завершения второй сборки Docker Hub

Если вы вернетесь на страницу сборки, в разделе «История сборок» должна появиться новая строка, и вы можете следовать этой сборке, как в шаге 8.

> **СОВЕТ.** Вам будет отправлено электронное письмо при наличии в сборке ошибки (если все в порядке, письма не будет), поэтому, как только вы привыкнете к этому рабочему процессу, нужно будет только проверить его, если вы получите электронное письмо.

Теперь можно использовать рабочий процесс Docker Hub. Вы быстро привыкнете к этому фреймворку и найдете его бесценным для поддержания ваших сборок в актуальном состоянии и снижения когнитивной нагрузки при повторной сборке файлов Dockerfiles вручную.

ОБСУЖДЕНИЕ

Поскольку Docker Hub – это канонический источник образов, помещение туда данных во время процесса непрерывной интеграции может сделать

некоторые вещи более простыми (например, распространение образов третьим лицам). Куда проще, когда нет необходимости самим запускать процесс сборки, и это дает вам дополнительные преимущества, такие как галочка напротив списка в Docker Hub, указывающая, что сборка была выполнена на доверенном сервере.

Наличие дополнительной уверенности в сборках помогает вам соблюдать *контракт Docker* в методе 70 – в методе 113 мы рассмотрим, как определенные компьютеры иногда могут влиять на сборки Docker, поэтому использование полностью независимой системы полезно для повышения уверенности в конечных результатах.

8.2. Более эффективные сборки

Непрерывная интеграция подразумевает более частую повторную сборку вашего программного обеспечения и тесты. Хотя Docker облегчает доставку непрерывной интеграции, следующая проблема, с которой вы можете столкнуться, – это увеличение нагрузки на ваши вычислительные ресурсы.

Мы рассмотрим способы уменьшения этого давления с точки зрения дискового ввода-вывода, пропускной способности сети и автоматического тестирования.

МЕТОД 62 Ускорение сборок с интенсивным вводом-выводом с помощью eatmydata

Поскольку Docker отлично подходит для автоматического выполнения сборок, с течением времени вы, вероятно, будете выполнять большое количество сборок с интенсивным дисковым вводом-выводом. Задания Jenkins, сценарии повторного создания базы данных и масштабные проверки кода – все это сильно ударит по вашим дискам. В этих случаях вы будете благодарны за любое увеличение скорости, которое можете получить как для экономии времени, так и для минимизации многих дополнительных накладных расходов, возникающих в результате конфликта ресурсов.

Данный метод продемонстрировал увеличение скорости до 1: 3, и наш опыт подтверждает это. И это неплохо!

ПРОБЛЕМА

Вы хотите ускорить сборки с интенсивным вводом/выводом.

РЕШЕНИЕ

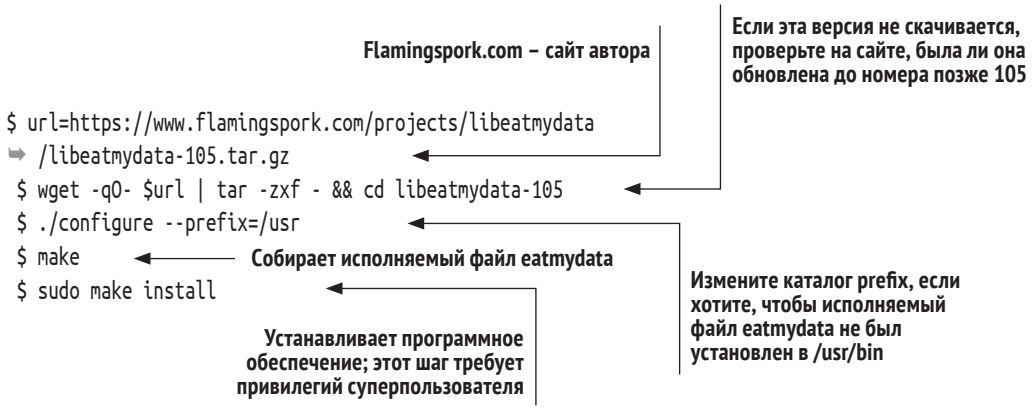
eatmydata – это программа, которая принимает ваши системные вызовы для записи данных и делает их очень быстрыми, минуя работу, необходимую для сохранения этих изменений. Это влечет за собой некоторый недостаток с точки зрения безопасности, поэтому программа не рекомендуется для обычного использования, но весьма полезна для сред, не предназначенных для сохранения, таких как тестирование.

Установка eatmydata

Чтобы установить eatmydata в свой контейнер, у вас есть несколько вариантов:

- если вы используете дистрибутив на основе deb, можете выполнить команду `apt-get install`;
- если вы используете дистрибутив на основе rpm, то сможете запустить команду `rpm --install`, выполнив поиск в интернете и загрузку. Такие сайты, как rpmfind.net, являются хорошим началом;
- в крайнем случае, и, если у вас установлен компилятор, вы можете загрузить и скомпилировать ее напрямую, как показано в следующем листинге.

Листинг 8.2. Компиляция и установка eatmydata



Использование eatmydata

После того, как libeatmydata будет установлена в образе (из пакета или из источника), запустите сценарий оболочки eatmydata перед любой командой, чтобы воспользоваться ею:

```
docker run -d mybuildautomation eatmydata /run_tests.sh
```

На рис. 8.1 на высоком уровне показано, как eatmydata экономит ваше время обработки.

> **ВНИМАНИЕ!** eatmydata пропускает шаги, чтобы гарантировать безопасную запись данных на диск, поэтому существует риск, что данные еще не окажутся на диске, когда программа будет думать, что это так. В случае с тестовыми прогонами это обычно не важно, потому что данные одноразовые, но не используйте eatmydata для ускорения среды, где данные имеют значение!

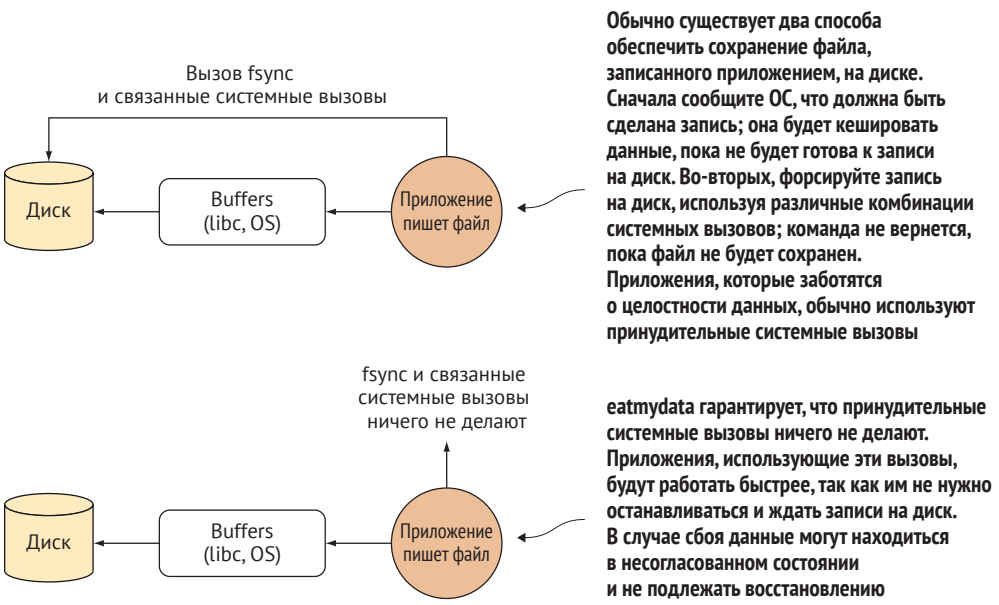


Рис. 8.1 ❖ Приложение выполняет запись на диск без (вверху) и с (внизу) использованием eatmydata

Имейте в виду, что выполнение команды `eatmydata docker run...` для запуска контейнера Docker, возможно, после установки eatmydata на хост или монтирования сокета Docker не даст ожидаемого эффекта из-за архитектуры клиент/сервер Docker, описанной во второй главе. Вместо этого вам нужно установить eatmydata внутри каждого отдельного контейнера, в котором вы хотите ее использовать.

ОБСУЖДЕНИЕ

Хотя точные варианты использования могут отличаться, место, где вы должны сразу же применить это, – метод 68. Очень редко, когда целостность данных в задании непрерывной интеграции имеет значение, – обычно вас просто интересует успех или неудача и журналы в случае неудачи.

Еще один важный метод – метод 77. База данных – это место, где целостность данных действительно имеет большое значение (любая популярная база данных будет спроектирована так, чтобы не потерять данные в случае потери мощности компьютером), но если вы просто проводите какие-то тесты или эксперименты, дополнительные затраты не нужны.

МЕТОД 63 Настройка кеша пакетов для более быстрой сборки

Поскольку Docker часто перестраивает сервисы для разработки, тестирования и производства, вы можете быстро добраться до точки, где постоянно обращаетесь к сети. Одной из основных причин этого является скачивание

файлов пакетов из интернета. Это может быть медленным (и дорогостоящим) даже на одном компьютере. Данный метод показывает, как настроить локальный кеш для загрузок пакетов, включая apt и yum.

ПРОБЛЕМА

Вы хотите ускорить сборку за счет сокращения сетевого ввода-вывода.

РЕШЕНИЕ

Установите прокси Squid для вашего менеджера пакетов. На рис. 8.2 показано, как работает этот метод.

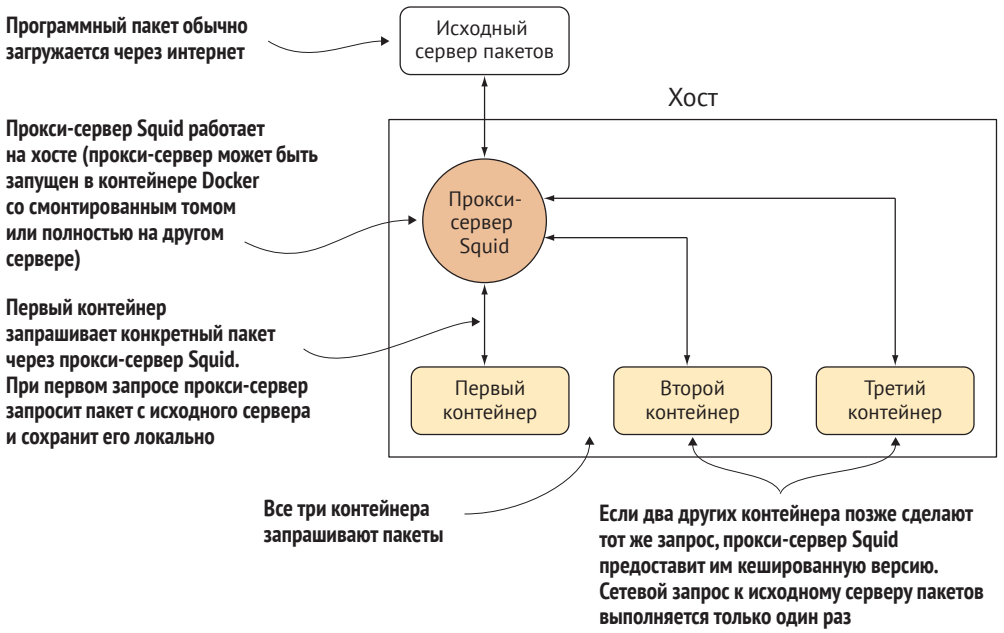


Рис. 8.2 ❖ Использование прокси-сервера Squid для кеширования пакетов

Поскольку вызовы пакетов сначала отправляются на локальный прокси-сервер Squid и запрашиваются через интернет только в первый раз, для каждого пакета должен быть только один запрос через интернет. Если у вас сотни контейнеров, которые скачивают одни и те же большие пакеты из интернета, это может сэкономить вам много времени и денег.

> **ПРИМЕЧАНИЕ.** При настройке этого на вашем хосте могут возникнуть проблемы с конфигурацией сети. В следующих разделах даются советы, чтобы определить, так ли это, но, если вы не знаете, как действовать дальше, вам, возможно, придется обратиться за помощью к дружелюбному сетевому администратору.

Debian

Для пакетов Debian (иначе называемых apt или .deb) установка проще, поскольку существует предварительно упакованная версия.

На хосте на основе Debian выполните эту команду:

```
sudo apt-get install squid-deb-proxy
```

Убедитесь, что служба запущена с помощью соединения с портом 8000:

```
$ telnet localhost 8000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

Нажмите сочетание клавиш **Ctrl-]**, а затем **Ctrl-d**, чтобы выйти, если вы видите предыдущий вывод. Если нет, то Squid либо неправильно установлен, либо установлен на нестандартный порт.

Чтобы настроить свой контейнер на использование этого прокси-сервера, мы предоставили следующий пример файла Dockerfile. Имейте в виду, что IP-адрес хоста с точки зрения контейнера способен меняться от запуска к запуску. По этой причине может возникнуть желание преобразовать этот Dockerfile в сценарий, который будет запускаться из контейнера перед установкой нового программного обеспечения.

Листинг 8.3. Настройка образа Debian для использования прокси-сервера apt

```
FROM debian
RUN apt-get update -y && apt-get install net-tools
 RUN echo "Acquire::http::Proxy \"http://$( \
route -n | awk '/^0.0.0.0/ {print $2}' \
 ):8000\";" \
 > /etc/apt/apt.conf.d/30proxy
 RUN echo "Acquire::http::Proxy::ppa.launchpad.net DIRECT;" >> \
    /etc/apt/apt.conf.d/30proxy
CMD ["/bin/bash"]
```

Гарантирует, что инструмент route установлен

Чтобы определить IP-адрес хоста с точки зрения контейнера, выполняет команду route и использует awk для извлечения соответствующего IP-адреса из вывода (см. метод 67)

Выведенные строки с соответствующим IP-адресом и конфигурацией добавляются в файл конфигурации прокси-сервера apt

Порт 8000 используется для подключения к прокси-серверу Squid на хост-компьютере

Yum

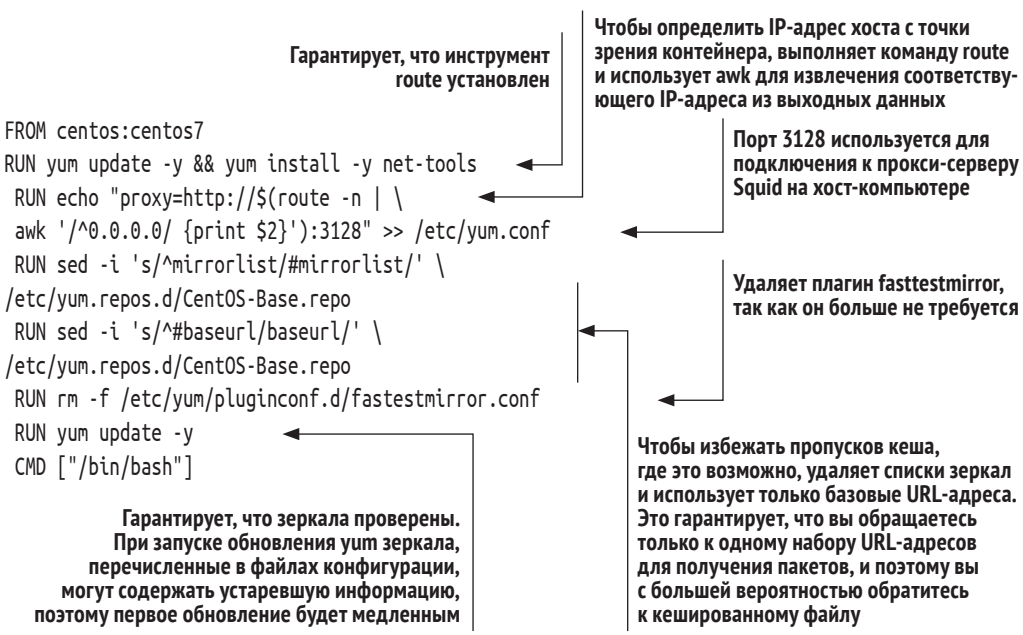
Установите Squid, используя менеджер пакетов. Затем вам понадобится изменить конфигурацию Squid, чтобы увеличить пространство кэша. Откройте файл /etc/squid/squid.conf и замените закомментированную строку, начинающуюся с `#cache_dir ufs /var/spool/squid`, на: `cache_dir ufs /var/spool/squid 10000 16 256`. В результате этого будет создано пространство размером 10 000 Мб, которого должно быть достаточно. Убедитесь, что служба запущена, подключившись к порту 3128 по протоколу Telnet:

```
$ telnet localhost 3128
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

Нажмите сочетание клавиш **Ctrl-]**, а затем **Ctrl-d**, чтобы выйти, если вы видите предыдущий вывод. Если нет, то Squid либо неправильно установлен, либо установлен на нестандартный порт.

Чтобы настроить свой контейнер на использование этого прокси-сервера, мы предоставили следующий пример файла Dockerfile. Имейте в виду, что IP-адрес хоста с точки зрения контейнера способен меняться от запуска к запуску. Может возникнуть желание преобразовать этот Dockerfile в сценарий, который будет запускаться из контейнера перед установкой нового программного обеспечения.

Листинг 8.4. Настройка образа CentOS для использования прокси-сервера yum



Если вы настроите два контейнера таким образом и установите один и тот же большой пакет на оба контейнера один за другим, вы должны заметить, что при второй установке необходимые компоненты загружаются гораздо быстрее, чем при первой.

ОБСУЖДЕНИЕ

Возможно, вы заметили, что можете запустить прокси-сервер Squid в контейнере, а не на хосте. Эта опция не была показана здесь для упрощения объяснения (в некоторых случаях требуется больше шагов, чтобы Squid работал внутри контейнера). Вы можете подробнее прочитать об этом, а также о том, как заставить контейнеры автоматически использовать прокси, на странице https://github.com/jpetazzo/squid-in-a-can.

МЕТОД 64 Headless Chrome в контейнере

Выполнение тестов является важной частью непрерывной интеграции, и большинство фреймворков модульных тестов будут работать в Docker без каких-либо проблем. Но иногда требуется более сложное тестирование, начиная с проверки правильности совместной работы нескольких микросервисов, заканчивая проверкой работоспособности интерфейса веб-сайта. Для посещения веб-интерфейса требуется какой-нибудь браузер, поэтому для решения этой проблемы нам нужен способ запустить браузер внутри контейнера, а затем управлять им программно.

ПРОБЛЕМА

Вы хотите протестировать браузер Chrome в контейнере без графического интерфейса пользователя.

РЕШЕНИЕ

Используйте библиотеку Puppeteer Node.js в образе для автоматизации действий Chrome.

Эта библиотека поддерживается группой разработчиков Google Chrome и позволяет писать сценарии для Chrome для тестирования. Этот браузер может работать в режиме headless. Это означает, что вам не нужен графический интерфейс пользователя для работы с ним.

> **ПРИМЕЧАНИЕ.** Этот образ также поддерживается нами на GitHub на странице https://github.com/docker-in-practice/docker-puppeteer и также доступен в виде образа Docker с помощью команды `docker pull dockerinpractice/docker-puppeteer`.

В следующем листинге показан файл Dockerfile, который создаст образ, содержащий все необходимое для начала работы с Puppeteer.

Листинг 8.5. Dockerfile Puppeteer

```
FROM ubuntu:16.04
 RUN apt-get update -y && apt-get install -y \
     npm python-software-properties curl git \
     libpangocairo-1.0-0 libx11-xcb1 \
     libxcomposite1 libxcursor1 libxdamage1 \
     libxi6 libxtst6 libnss3 libcups2 libxss1 \
     libxrandr2 libgconf-2-4 libasound2 \
     libatk1.0-0 libgtk-3-0 vim gconf-service \
     libappindicator1 libc6 libcairo2 libcups2 \
     libdbus-1-3 libexpat1 libfontconfig1 libgcc1 \
     libgdk-pixbuf2.0-0 libglib2.0-0 libnspr4 \
     libpango-1.0-0 libstdc++6 libx11-6 libxcb1 \
     libxext6 libxfixes3 libxrender1 libxtst6 \
     ca-certificates fonts-liberation lsb-release \
     xdg-utils wget
 RUN curl -sL https://deb.nodesource.com/setup_8.x | bash -
 RUN apt-get install -y nodejs
 RUN useradd -m puser
 USER puser
 RUN mkdir -p /home/puser/node_modules
 ENV NODE_PATH /home/puppeteer/node_modules
 WORKDIR /home/puser/node_modules
 RUN npm i webpack
 RUN git clone https://github.com/GoogleChrome/puppeteer
 WORKDIR /home/puser/node_modules/puppeteer
 RUN npm i .
 WORKDIR /home/puser/node_modules/puppeteer/examples
 RUN perl -p -i -e \
     "s/puppeteer.launch\(\)/puppeteer.launch({args: ['--no-sandbox']})/" *
 CMD echo 'eg: node pdf.js' && bash
```

Начинается с базового образа Ubuntu

Устанавливает все необходимое программное обеспечение. Это большинство библиотек отображения, необходимых, для того чтобы заставить Chrome работать в контейнере

Устанавливает последнюю версию nodejs

Устанавливает пакет Ubuntu nodejs

Создает пользователя без полномочий root «puser» (который нужен библиотеке для запуска)

Создает папку модулей узла

Устанавливает переменную среды NODE_PATH в папку node_module

Устанавливает текущий рабочий каталог в путь к модулю узла

Устанавливает webpack (зависимость Puppeteer)

Клонирует код модуля Puppeteer

Переходит в папку с кодом Puppeteer

Устанавливает библиотеку Puppeteer NodeJS

Переходит в папку с примерами Puppeteer

Добавляет аргумент no-sandbox в аргументы запуска Puppeteer, чтобы преодолеть настройки безопасности при работе в контейнере

Запускает контейнер с bash, добавляя полезную команду echo

Соберите и запустите этот Dockerfile с помощью команды:

```
$ docker build -t puppeteer .
```

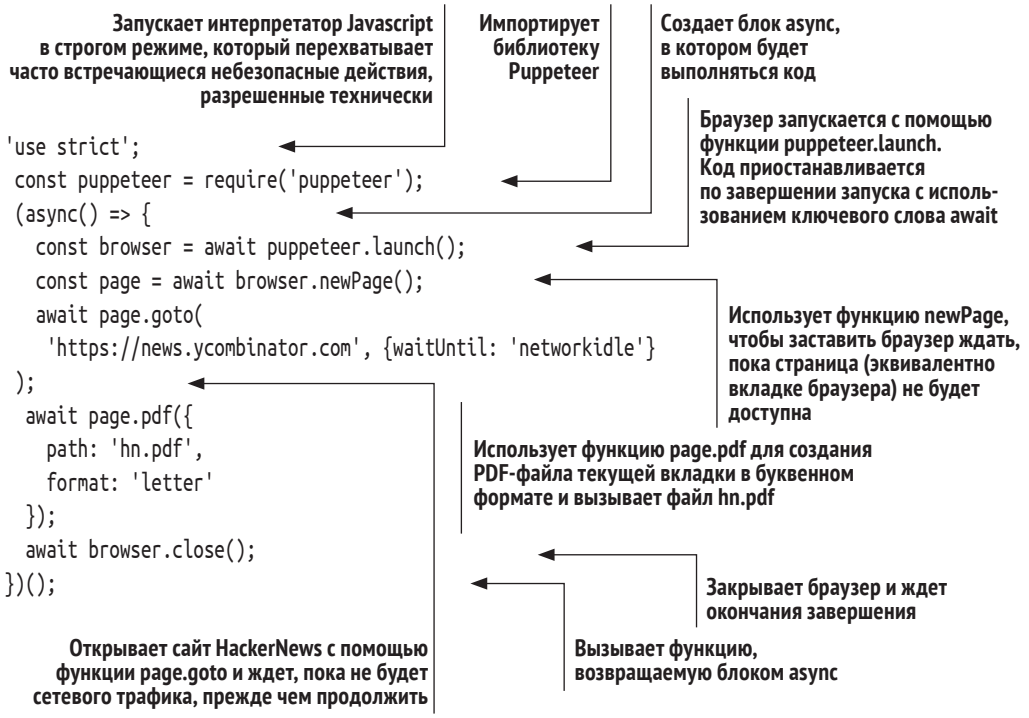
Затем выполните это:

```
$ docker run -ti puppeteer
eg: node pdf.js
puser@03b9be05e81d:~/node_modules/puppeteer/examples$
```

Вам будет представлен терминал и предложение запустить `node pdf.js`.

Файл pdf.js содержит простой сценарий, который служит примером того, что можно сделать с помощью библиотеки Puppeteer.

Листинг 8.6. pdf.js



За пределами этого простого примера пользователю Puppeteer доступно множество опций. Подробное описание API Puppeteer выходит за рамки данного метода. Если вы хотите более подробно изучить API и адаптировать этот метод, посмотрите документацию по API Puppeteer на GitHub: https://github.com/GoogleChrome/puppeteer/blob/master/docs/api.md.

ОБСУЖДЕНИЕ

Данный метод показывает, как можно использовать Docker для тестирования определенного браузера.

Следующий метод расширяет предыдущий двумя способами: с помощью Selenium, популярного инструмента тестирования, который может работать с несколькими браузерами, сочетая его с изучением системы X11, чтобы вы могли видеть браузер, работающий в графическом окне, а не в режиме headless, используемом в этом методе.

МЕТОД 65 Выполнение тестов Selenium внутри Docker

Один из вариантов использования Docker, который мы еще не рассматривали, – это запуск графических приложений. В главе 3 для подключения к контейнерам мы использовали VNC во время подхода к разработке под названием «сохранить игру» (метод 19), но этот способ может быть неудобен – окна содержатся в окне клиента VNC viewer, и взаимодействие с рабочим столом может быть несколько ограничено.

Рассмотрим альтернативу этому, продемонстрировав, как можно писать графические тесты с использованием Selenium. Мы также покажем вам, как этот образ используют для запуска тестов в рамках вашего рабочего процесса непрерывной интеграции.

ПРОБЛЕМА

Вы хотите иметь возможность запускать графические программы в процессе непрерывной интеграции, отображая те же графические программы на своем экране.

РЕШЕНИЕ

Используйте совместно сокет сервера X11 для просмотра программ на собственном экране и xvfb в своем процессе непрерывной интеграции.

Независимо от того, что нужно сделать для запуска вашего контейнера, у вас должен быть сокет Unix, который X11 использует для отображения ваших окон, смонтированных в виде тома внутри контейнера, и вам нужно указать, на каком экране должны отображаться ваши окна.

Вы можете дважды проверить, установлены ли эти две вещи в значения по умолчанию, выполнив следующие команды на своем хосте:

```
~ $ ls /tmp/.X11-unix/
X0
~ $ echo $DISPLAY
:0
```

Первая команда проверяет, работает ли сокет Unix сервера X11 в месте, допускаемом для оставшейся части этого метода. Вторая команда проверяет приложения переменных среды, чтобы найти сокет X11. Если ваш вывод для этих команд не совпадает с приведенным здесь выводом, может потребоваться изменить некоторые аргументы команд.

Теперь, когда вы проверили настройки своего компьютера, нужно, чтобы приложения, работающие внутри контейнера, незаметно отображались за пределами контейнера. Основная проблема, которую важно преодолеть, – это меры безопасности, вводимые вашим компьютером для предотвращения подключения к нему других лиц, захвата контроля над дисплеем и потенциальной записи нажатий клавиш. В методе 29 вы вкратце видели, как это сделать, но не говорили о том, как это работает, и не рассматривали какие-либо альтернативы.

У X11 есть несколько способов, чтобы аутентифицировать контейнер для использования вашего X-сокета. Сначала мы рассмотрим файл Xauthority – он должен присутствовать в вашем домашнем каталоге. Файл содержит имена хостов наряду с «секретным cookie», который каждый хост должен использовать для подключения. Давая своему контейнеру то же имя хоста, что и своему компьютеру, и взяв то же самое имя пользователя, что и вне контейнера, вы можете использовать существующий файл Xauthority.

Листинг 8.7. Запускаем контейнер, используя экран с поддержкой Xauthority

```
$ ls $HOME/.Xauthority
/home/myuser/.Xauthority
$ docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
    --hostname=$HOSTNAME -v $HOME/.Xauthority:$HOME/.Xauthority \
    -it -e EXTUSER=$USER ubuntu:16.04 bash -c 'useradd $USER && exec bash'
```

Второй метод, позволяющий Docker получить доступ к сокету, – гораздо более грубый инструмент, и у него есть проблемы с безопасностью, потому что он отключает всю защиту, которую дает вам X. Если никто не имеет доступа к вашему компьютеру, это может быть приемлемым решением, но вы всегда должны сначала пытаться использовать файл .Xauthority. Можно снова обезопасить себя после попытки выполнить следующие шаги, запустив команду `xhost -` (хотя это заблокирует ваш контейнер):

Листинг 8.8. Запускаем контейнер, используя экран с поддержкой xhost

```
$ xhost +
access control disabled, clients can connect from any host
$ docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
    -it ubuntu:16.04 bash
```

Первая строка в предыдущем листинге отключает весь контроль доступа к X, а вторая запускает контейнер. Обратите внимание, что вам не нужно устанавливать имя хоста или монтировать что-либо кроме X-сокета.

После того как вы запустили контейнер, пришло время проверить, работает ли он. Вы можете сделать это, выполнив следующие команды, если идете по маршруту.Xauthority:

```
root@myhost:/# apt-get update && apt-get install -y x11-apps
[...]
root@myhost:/# su - $EXTUSER -c "xeyes"
```

В качестве альтернативы можно использовать эти немного разные команды, если идете по маршруту xhost, потому что вам не нужно запускать команду от имени определенного пользователя:

```
root@ef351febcee4:/# apt-get update && apt-get install -y x11-apps
[...]
root@ef351febcee4:/# xeyes
```

Будет запущено классическое приложение xeyes, которое проверяет, работает ли X. Вы должны видеть, как глаза следуют за курсором, когда вы перемещаете его по экрану. Обратите внимание, что (в отличие от VNC) это приложение интегрировано в ваш рабочий стол, – если вы запустите xeyes несколько раз, то увидите много окон.

Пришло время приступить к работе с Selenium. Если вы никогда не использовали его ранее, это инструмент, обладающий способностью автоматизировать действия браузера, и обычно он применяется для тестирования кода веб-сайта – для работы браузера требуется графический дисплей. Хотя наиболее часто он используется с Java, мы возьмем Python для большей интерактивности.

В следующем листинге сначала устанавливаются Python, Firefox и менеджер пакетов Python, а затем используется менеджер пакетов Python для установки пакета Selenium Python.

Мы также скачиваем двоичный файл «драйвера», который Selenium использует для управления Firefox. Затем запускается Python REPL, а библиотека Selenium нужна для создания экземпляра Firefox.

Для простоты мы будем рассматривать только маршрут xhost – чтобы идти по маршруту Xauthority, вам потребуется создать домашний каталог для пользователя, чтобы у Firefox было место для сохранения настроек своего профиля.

Листинг 8.9. Установка требований Selenium и запуск браузера

```
root@myhost:/# apt-get install -y python2.7 python-pip firefox wget
[...]
root@myhost:/# pip install selenium
Collecting selenium
[...]
Successfully installed selenium-3.5.0
root@myhost:/# url=https://github.com/mozilla/geckodriver/releases/download
➥ /v0.18.0/geckodriver-v0.18.0-linux64.tar.gz
root@myhost:/# wget -qO- $url | tar -C /usr/bin -zxf -
root@myhost:/# python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> from selenium import webdriver
>>> b = webdriver.Firefox ()
```

Как вы могли заметить, Firefox запустился и появился на вашем экране.

Теперь можно поэкспериментировать с Selenium. Ниже приведен пример сеанса в GitHub – вам понадобится базовое понимание селекторов CSS, чтобы понять, что здесь происходит. Обратите внимание, что веб-сайты часто меняются, поэтому для корректной работы этого фрагмента может потребоваться изменение:

```
>>> b.get('https://github.com/search')
>>> searchselector = '#search_form input[type="text"]'
>>> searchbox = b.find_element_by_css_selector(searchselector)
>>> searchbox.send_keys('docker-in-practice')
>>> searchbox.submit()
>>> import time
>>> time.sleep(2) # wait for page JS to run
>>> usersxpath = '//nav//a[contains(text(), "Users")]'
>>> userslink = b.find_element_by_xpath(usersxpath)
>>> userslink.click()
>>> dlinkselector = '.user-list-info a'
>>> dlink = b.find_elements_by_css_selector(dlinkselector)[0]
>>> dlink.click()
>>> mlinkselector = '.meta-item a'
>>> mlink = b.find_element_by_css_selector(mlinkselector)
>>> mlink.click()
```

Детали здесь не важны, хотя вы можете получить представление о том, что происходит, переключившись на Firefox между командами, – мы переходим к организации docker-in-practice в GitHub и нажимаем ссылку организации. Основным выводом является то, что мы пишем команды на Python в контейнере и видим, как они вступают в силу в окне Firefox, работающем внутри контейнера, но видимом на рабочем столе.

Это отлично подходит для отладки тестов, которые вы пишете, но как бы вы интегрировали их в конвейер непрерывной интеграции с тем же образом Docker? Сервер непрерывной интеграции обычно не имеет графического дисплея, поэтому нужно сделать это без монтирования собственного сокета X-сервера. Но Firefox все еще нужен X-сервер для запуска.

Существует полезный инструмент, созданный для подобных ситуаций, под названием xvfb, который делает вид, что X-сервер работает для приложений, но ему не нужен монитор.

Чтобы увидеть, как это работает, мы установим xvfb, зафиксируем контейнер, пометим его как `selenium` и создадим тестовый сценарий:

Листинг 8.10. Создание тестового сценария Selenium

```
>>> exit()
root@myhost:/# apt-get install -y xvfb
[...]
root@myhost:/# exit
$ docker commit ef351febcee4 selenium
d1cbfbc76790cae5f4ae95805a8ca4fc4cd1353c72d7a90b90ccfb79de4f2f9b
$ cat > myscript.py << EOF
from selenium import webdriver
b = webdriver.Firefox ()
print 'Visiting github'
b.get('https://github.com/search')
print 'Performing search'
searchselector = '#search_form input[type="text"]'
searchbox = b.find_element_by_css_selector(searchselector)
searchbox.send_keys('docker-in-practice')
searchbox.submit()
print 'Switching to user search'
import time
time.sleep(2) # wait for page JS to run
usersxpath = '//nav//a[contains(text(), "Users")]'
userslink = b.find_element_by_xpath(usersxpath)
userslink.click()
print 'Opening docker in practice user page'
dlinkselector = '.user-list-info a'
dlink = b.find_elements_by_css_selector(dlinkselector)[99]
dlink.click()
print 'Visiting docker in practice site'
mlinkselector = '.meta-item a'
mlink = b.find_element_by_css_selector(mlinkselector)
mlink.click()
print 'Done!'
EOF
```

Обратите внимание на небольшую разницу в присваивании переменной `dlink` (при индексировании видна позиция 99, а не 0). Пытаясь получить сотый результат, содержащий текст «Docker in Practice», вы инициируете ошибку, из-за которой контейнер Docker завершит работу с ненулевым состоянием, и произойдет сбой в конвейере непрерывной интеграции.

Время опробовать это:

```
$ docker run --rm -v $(pwd):/mnt selenium sh -c \
"xvfb-run -s '-screen 0 1024x768x24 -extension RANDR'\
python /mnt/myscript.py"
```

```
Visiting github
Performing search
Switching to user search
Opening docker in practice user page
Traceback (most recent call last):
  File "myscript.py", line 15, in <module>
      dlink = b.find_elements_by_css_selector(dlinkselector)[99]
      IndexError: list index out of range
$ echo $?
1
```

Вы запустили самоудаляющийся контейнер, который выполняет тестовый сценарий Python, работающий на виртуальном X-сервере. Как и ожидалось, он завершился неудачей и вернул ненулевой код выхода.

> **ПРИМЕЧАНИЕ.** `sh -c "command string here"` – неудачный результат того, как Docker обрабатывает значения `CMD` по умолчанию. Если бы вы создали этот образ с помощью файла Dockerfile, вы бы могли удалить `sh -c` и сделать `xvfb-run -s '-screen 01024x768x24-extension RANDR'` точкой входа, что позволило бы вам передать команду тестирования в качестве аргументов образа.

Docker – гибкий инструмент, и его можно использовать в некоторых неожиданных целях (в данном случае в графических приложениях). Некоторые запускают *все* свои графические приложения в Docker, включая игры!

Мы не будем заходить так далеко (метод 40 рассматривает это как минимум для ваших инструментов разработчика), но мы обнаружили, что пересмотр предположений относительно Docker может привести к неожиданным вариантам использования. Например, в приложении A рассказывается о запуске графических приложений Linux в Windows после установки Docker для Windows.

8.3. Контейнеризация процесса непрерывной интеграции

Если у вас есть согласованный процесс разработки в разных командах, важно также иметь согласованный процесс сборки. Сборки, случайным образом заканчивающиеся неудачей, подрывают саму суть Docker.

В результате имеет смысл *контейнеризировать* весь процесс непрерывной интеграции. Это не только обеспечивает повторяемость сборок, но и позволяет перемещать процесс непрерывной интеграции куда угодно, не боясь оставить позади какой-либо жизненно важный элемент конфигурации (вероятно, позже обнаруженный с большим разочарованием).

В этих методах мы будем использовать Jenkins (так как это наиболее широко используемый инструмент), но те же методы следует применять и к другим инструментам непрерывной интеграции. Здесь не предполагается тесного знакомства с Jenkins, но мы не будем рассматривать настройку стандартных тестов и сборок.

Данная информация не важна для приведенных здесь методов.

МЕТОД 66 Запуск ведущего устройства Jenkins в контейнере Docker

Помещение ведущего устройства Jenkins в контейнер не имеет столько преимуществ, как в случае с ведомым устройством (см. следующий метод), но обеспечивает стандартный выигрыш для неизменяемых образов. Мы обнаружили, что возможность фиксации конфигураций хорошо известных ведущих устройств и плагинов значительно облегчает бремя экспериментов.

ПРОБЛЕМА

Вам нужен портативный сервер Jenkins.

РЕШЕНИЕ

Используйте официальный образ Docker для Jenkins для запуска своего сервера.

Запуск Jenkins в контейнере Docker дает вам некоторые преимущества по сравнению с простой установкой хоста. В нашем офисе уже были слышны крики: «Не трогайте мою конфигурацию сервера Jenkins!» (или, что еще хуже, «Кто трогал мой сервер Jenkins?»), и возможность клонировать состояние сервера Jenkins с помощью команды работающего контейнера `docker export` для экспериментов с обновлениями и изменениями помогает заглушить эти жалобы.

Аналогично резервное копирование и портирование становятся проще.

В этом методе мы возьмем официальный образ докера Jenkins и внесем несколько изменений, чтобы упростить некоторые более поздние методики, требующие возможности доступа к сокету Docker, например сборку Docker из Jenkins.

> **ПРИМЕЧАНИЕ.** Примеры из этой книги, относящиеся к Jenkins, доступны на GitHub: `git clone https://github.com/docker-in-practice/jenkins.git`.

> **ПРИМЕЧАНИЕ.** В этой книге образ Jenkins и его команда `run` будут использоваться в качестве сервера в методах, связанных с Jenkins.

Создание сервера

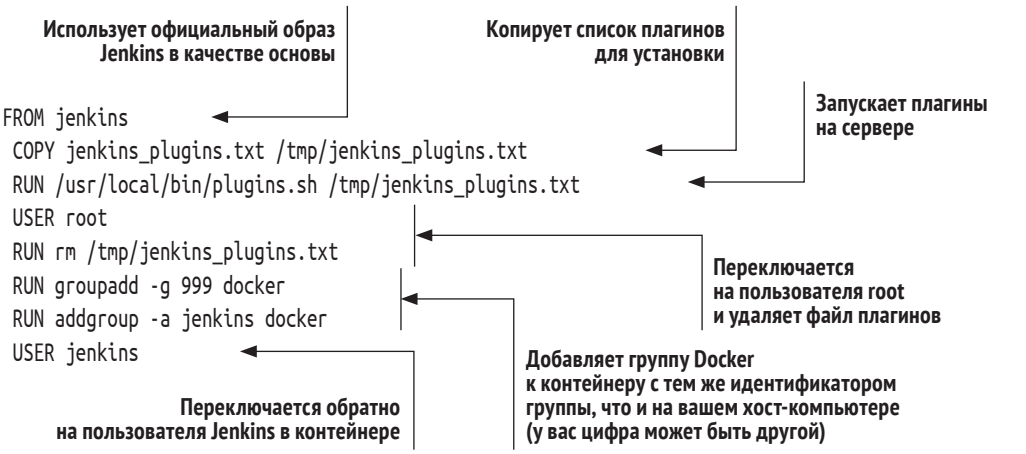
Сначала мы подготовим список необходимых нам плагинов для сервера и поместим его в файл с именем jenkins_plugins.txt:

```
swarm:3.4
```

Этот очень короткий список состоит из плагина Jenkins под названием Swarm (никакой связи с Docker Swarm), который мы будем использовать в последующем методе.

В листинге 8.11 показан файл Dockerfile для сборки сервера Jenkins.

Листинг 8.11. Сборка сервера Jenkins



Инструкции `CMD` или `ENTRYPOINT` не указаны, поскольку мы хотим наследовать команду запуска, определенную в официальном образе Jenkins.

Идентификатор группы для Docker на вашем хост-компьютере может быть другим. Чтобы узнать, какой у вас идентификатор, выполните эту команду:

```
$ grep -w ^docker /etc/group
docker:x:999:imiell
```

Замените значение, если оно отличается от 999.

> **ПРЕДУПРЕЖДЕНИЕ.** Идентификатор группы должен совпадать в среде сервера Jenkins и в вашей ведомой среде, если вы планируете запускать Docker из контейнера Jenkins Docker. Также возникнет потенциальная проблема переносимости, если вы решите переместить сервер (такая же проблема возникнет при установке собственного сервера).

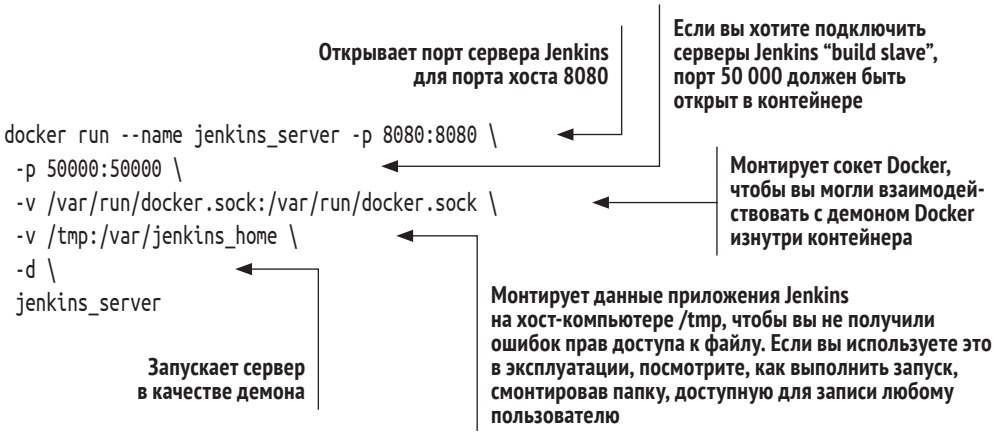
Переменные среды здесь не помогут сами по себе, так как группу нужно настраивать во время сборки, а не динамически.

Чтобы собрать образ в этом сценарии, выполните команду:

```
docker build -t jenkins_server .
```

Запуск сервера

Теперь можете запустить сервер для Docker с помощью этой команды:



Если вы перейдете по адресу http://localhost:8080, то увидите интерфейс конфигурации Jenkins, – следуйте этому процессу до связывания, возможно, используя команду `docker exec` (описанную в методе 12), чтобы получить пароль, который вам будет предложен при первом шаге.

По завершении ваш сервер Jenkins будет готов к работе с уже установленными плагинами (вместе с рядом других плагинов в зависимости от параметров, выбранных вами в процессе установки). Чтобы проверить это, перейдите в раздел **Manage Jenkins > Manage Plugins > Installed** (Управление Jenkins > Управление плагинами > Установлено) и найдите Swarm, чтобы убедиться, что он установлен.

ОБСУЖДЕНИЕ

Вы увидите, что мы смонтировали сокет Docker, используя ведущее устройство Jenkins, как мы это делали в методе 45, обеспечивая доступ к демону Docker. Это позволяет вам выполнять сборки Docker, используя встроенную модель «ведущий/ведомый», запуская контейнеры на хосте.

> **ПРИМЕЧАНИЕ.** Код для этого метода и связанных с ним методов доступен на GitHub по адресу https://github.com/docker-in-practice/jenkins.

МЕТОД 67 Содержание сложной среды разработки

Портативность и легкость Docker делают его очевидным выбором для ведомого устройства непрерывной интеграции (компьютера, к которому подключается ведущее устройство непрерывной интеграции для выполнения сборок). Ведомое устройство непрерывной интеграции Docker – это ощутимый переход от ведомого устройства виртуальной машины (и еще больший рывок по сравнению с компьютерами для сборки «на голом железе»). Это позволяет

выполнять сборки во множестве сред с одним хостом, чтобы быстро разрушать и создавать чистые среды для обеспечения незагрязненных сборок и использовать все знакомые инструменты Docker для управления средами сборки.

Возможность рассматривать ведомое устройство непрерывной интеграции как еще один контейнер Docker особенно интересна. На одном из ваших ведомых устройств непрерывной интеграции происходят таинственные сбои сборки?

Извлеките образ и опробуйте сборку самостоятельно.

ПРОБЛЕМА

Вы хотите масштабировать и модифицировать ведомое устройство Jenkins.

РЕШЕНИЕ

Используйте Docker для инкапсуляции конфигурации вашего ведомого устройства в образ Docker и выполните развертывание.

Многие организации устанавливают тяжелое ведомое устройство Jenkins (часто на том же хосте, что и сервер), поддерживаемое центральным ИТ-подразделением, которое некоторое время служит полезной цели. С течением времени команды расширяют свои кодовые базы и расходятся, требования растут, чтобы все больше и больше программ устанавливалось, обновлялось или изменялось так, чтобы выполнялись задания.

Рис. 8.3 ❖ Перегруженный сервер Jenkins

На рис. 8.3 показана упрощенная версия этого сценария. Представьте себе сотни пакетов программного обеспечения и множество новых запросов, которые вызывают головную боль у перегруженной работой команды обслуживания инфраструктуры.

> **ПРИМЕЧАНИЕ.** Данный метод был разработан, чтобы показать вам основы работы ведомого устройства Jenkins в контейнере. Это делает результат менее переносимым, но урок легче усвоить. Как только вы поймете все приемы, описанные в этой главе, вы сможете выполнить более портативную настройку.

Известно, что наступает патовая ситуация, потому что системные администраторы могут неохотно обновлять свои сценарии управления конфигурацией для одной группы людей, поскольку боятся нарушить сборку другой, и команды все больше разочаровываются медлительностью, с которой происходят изменения.

Docker (естественно) предлагает решение, позволяющее нескольким командам использовать базовый образ для своего личного ведомого устройства Jenkins, применяя то же оборудование, что и раньше. Вы можете создать образ с необходимыми общими инструментами и позволить командам изменять его в соответствии со своими потребностями.

Некоторые участники загрузили свои справочные ведомые устройства в Docker Hub; вы можете найти их, введя в поиск фразу «jenkins slave» на Docker Hub. Следующий список является минимальным файлом Dockerfile ведомого устройства Jenkins.

Листинг 8.12. Простой файл Dockerfile ведомого устройства Jenkins

```
FROM ubuntu:16.04
ENV DEBIAN_FRONTEND noninteractive
RUN groupadd -g 1000 jenkins_slave
 RUN useradd -d /home/jenkins_slave -s /bin/bash \
-m jenkins_slave -u 1000 -g jenkins_slave
 RUN echo jenkins_slave:jpass | chpasswd
 RUN apt-get update && apt-get install -y \
openssh-server openjdk-8-jre wget iproute2
 RUN mkdir -p /var/run/sshd
 CMD ip route | grep "default via" \
 | awk '{print $3}' && /usr/sbin/sshd -D
```

Устанавливает пароль пользователя Jenkins как «jpass». При более сложной настройке вы, вероятно, захотите использовать другие методы аутентификации

Создает пользователя и группу jenkins_slave

Устанавливает необходимое программное обеспечение для работы в качестве ведомого сервера Jenkins

При запуске выводит IP-адрес хост-компьютера с точки зрения контейнера и запускает SSH-сервер

Создайте образ ведомого устройства, пометив его как `jenkins_slave`:

```
$ docker build -t jenkins_slave .
```

Запустите его с помощью этой команды:

```
$ docker run --name jenkins_slave -ti -p 2222:22 jenkins_slave
172.17.0.1
```

Сервер Jenkins должен быть запущен

Если на вашем хосте еще не запущен сервер Jenkins, настройте его, используя предыдущий метод. Если вы спешите, выполните эту команду:

```
$ docker run --name jenkins_server -p 8080:8080 -p 50000:50000 \
dockerinpractice/jenkins:server
```

Она сделает сервер Jenkins доступным по адресу http://localhost:8080, если вы выполнили ее на своем локальном компьютере. Нужно будет пройти процесс установки, прежде чем использовать его.

Если вы перейдете к серверу Jenkins, откроется страница с приветствием, изображенная на рис. 8.4.

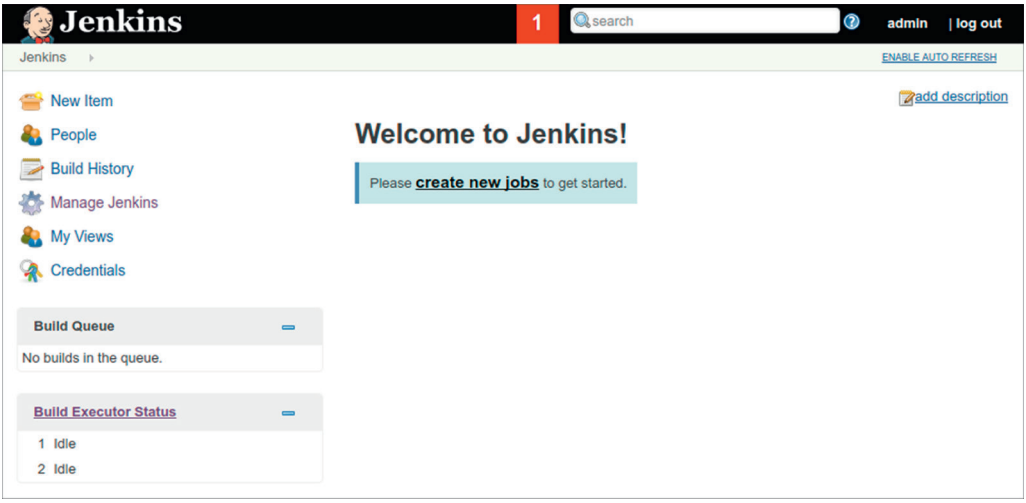


Рис. 8.4 ❖ Домашняя страница Jenkins

Вы можете добавить ведомое устройство, щелкнув **Build Executor Status > New Node** и добавив имя узла как Permanent Agent, как показано на рис. 8.5. Назовите его `mydockerslave`.

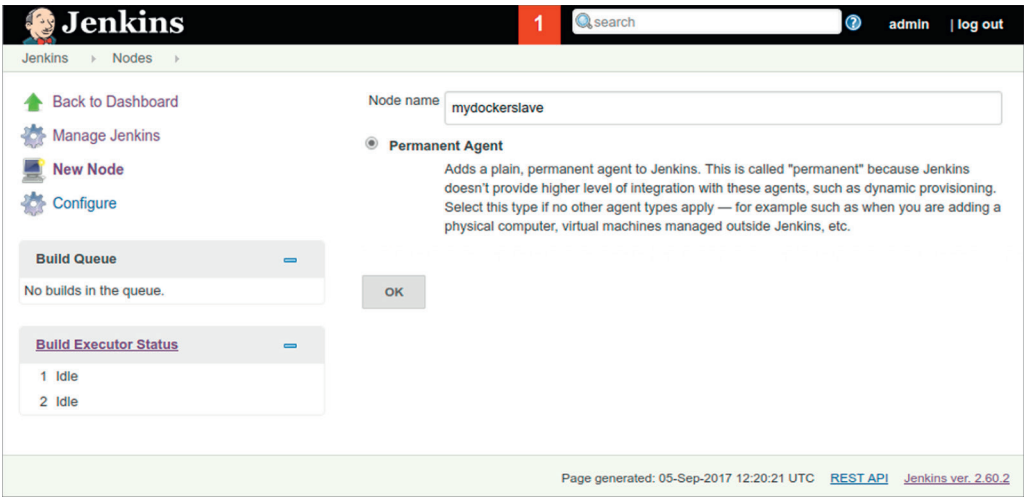


Рис. 8.5 ❖ Добавляем имя для нового узла

Нажмите **OK** и настройте его, как показано на рис. 8.6:

- установите для удаленного корневого каталога значение /home/jenkins_slave;
- присвойте ему ярлык «dockerslave»;
- убедитесь, что выбрана опция Launch Slave Agents Via SSH;
- установите для хоста IP-адрес маршрута, видимый из контейнера (вывод с помощью команды `docker run` ранее);
- нажмите кнопку **Add** (Добавить), чтобы добавить учетные данные, и установите для имени пользователя значение «jenkins_slave» и пароль «jpass». Теперь выберите эти учетные данные из выпадающего списка;
- задайте для поля Host Key Verification Strategy значение Manually Trusted Key Verification Strategy, при котором SSH-ключ будет принят при первом подключении, либо Non Verifying Verification Strategy, при котором проверка ключа выполняться не будет;
- нажмите **Advanced** (Дополнительно), чтобы открыть поле **Порт**, и установите для него значение 2222;
- нажмите **Save** (Сохранить).

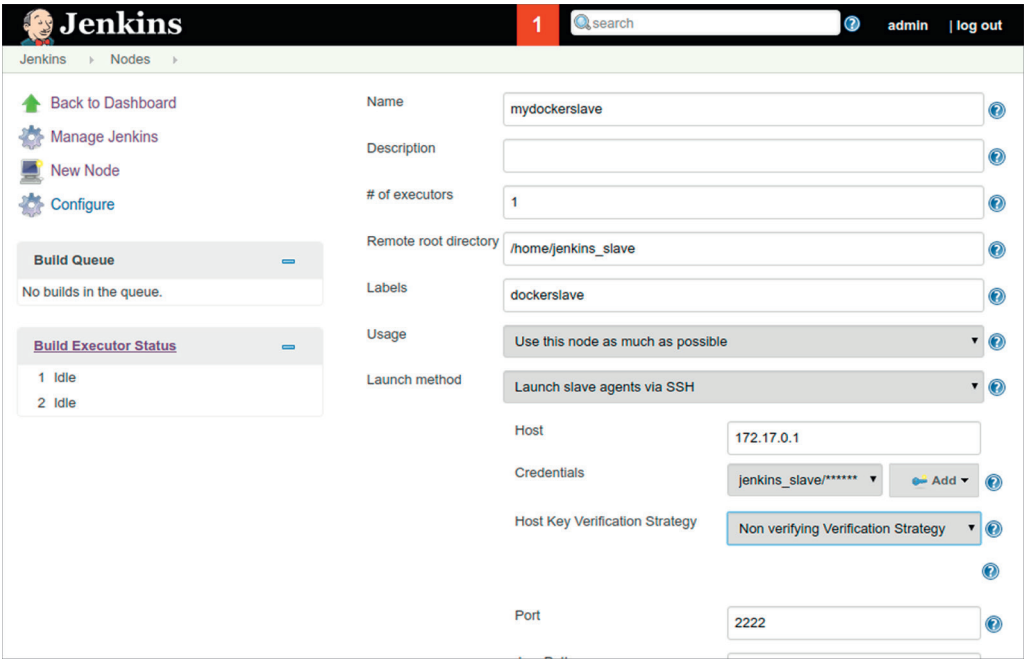


Рис. 8.6 ❖ Страница настроек узла Jenkins

Теперь нажмите на новое ведомое устройство и **Launch Slave Agent** (при условии, что это не произойдет автоматически). Через минуту вы увидите, что ведомый агент помечен как online.

Вернитесь на домашнюю страницу (кнопка **Jenkins** в левом верхнем углу) и нажмите **New Item** (Новый элемент). Создайте проект Freestyle под названием «test» и в разделе **Build** нажмите **Add Build Step > Execute Shell** с командой `echo done`. Прокрутите страницу вверх и выберите **Restrict Where Project Can Be Run** (Ограничить место выполнения проекта), введите выражение метки «dockerslave». Вы должны увидеть, что **Slaves In Label** имеет значение 1. Это означает, что задание теперь привязано к ведомому устройству Docker. Нажмите **Save** (Сохранить), чтобы создать задание.

Нажмите **Build Now** (Собрать сейчас), а затем – по ссылке build "# 1", которая появится ниже слева.

Затем нам нужна кнопка **Console Output**. Вы должны увидеть следующий вывод в главном окне:

```
Started by user admin
Building remotely on mydockerslave (dockerslave)
➥ in workspace /home/jenkins_slave/workspace/test
[test] $ /bin/sh -xe /tmp/jenkins5620917016462917386.sh
+ echo done
done
Finished: SUCCESS
```

Молодцы! Вы успешно создали собственное ведомое устройство Jenkins.

Теперь, если вы хотите создать свое уникальное ведомое устройство, все, что вам нужно сделать, – это изменить Dockerfile образа ведомого устройства на свой вкус и запустить его вместо примера.

> **ПРИМЕЧАНИЕ.** Код для этого метода и связанных с ним методов доступен на GitHub по адресу https://github.com/docker-in-practice/jenkins.

ОБСУЖДЕНИЕ

Этот метод поможет вам создать контейнер, который будет действовать как виртуальная машина. Он очень напоминает метод 12, но с дополнительной сложностью интеграции Jenkins.

Одна из особенно полезных стратегий – также смонтировать сокет Docker внутри контейнера и установить двоичный файл клиента Docker, чтобы вы могли выполнять сборки. Смотрите метод 45 для получения информации о монтировании сокета Docker (для другой цели) и приложение A, где приводятся подробности установки.

МЕТОД 68 Масштабирование процесса непрерывной интеграции с помощью плагина Swarm

Возможность воспроизводить среды – это большая победа, но ваши возможности сборки все еще ограничены числом выделенных компьютеров, которые имеются. Если вы хотите проводить эксперименты в разных средах, используя новооткрытую гибкость ведомых устройств Docker, вас может постигнуть разочарование. Производительность также может стать проблемой по более приземленным причинам – рост вашей команды!

ПРОБЛЕМА

Вы хотите масштабировать процесс непрерывной интеграции в соответствии с темпами разработки.

РЕШЕНИЕ

Используйте плагин Swarm и ведомое устройство Docker Swarm для динамической подготовки ведомых устройств Jenkins.

> **ПРИМЕЧАНИЕ.** Об этом уже упоминалось, но здесь стоит повторить: плагин Swarm не имеет никакого отношения к технологии Docker Swarm. Это две совершенно не связанные вещи, которые используют одно и то же слово. Тот факт, что они могут применяться вместе в данном случае, – чистое совпадение.

На многих предприятиях малого и среднего бизнеса имеется модель непрерывной интеграции, где для предоставления ресурсов, необходимых для выполнения заданий, выделяются один или несколько серверов. Это показано на рис. 8.7.

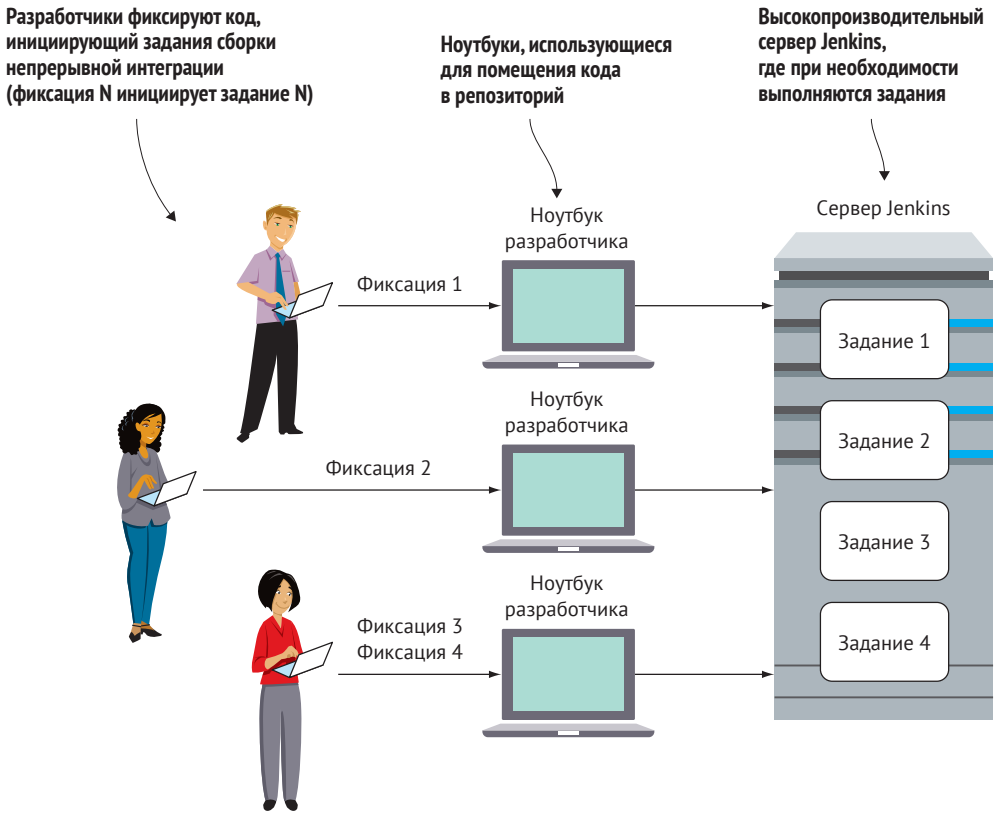


Рис. 8.7 ❖ До: Сервер Jenkins – в порядке, когда разработчик один, но масштабируемость отсутствует

Некоторое время это работает нормально, но, по мере того как процессы непрерывной интеграции становятся более встроенными, часто можно достичь предела производительности. Большинство рабочих нагрузок Jenkins инициируются при записи изменений в репозиторий, поэтому по мере того как все больше разработчиков вносят изменения, рабочая нагрузка увеличивается. Тогда количество жалоб в оперативную группу возрастает, поскольку занятые разработчики с нетерпением ждут результатов своих сборок.

Одно из правильных решений – иметь столько ведомых устройств Jenkins, сколько людей размещают файлы, как показано на рис. 8.8.

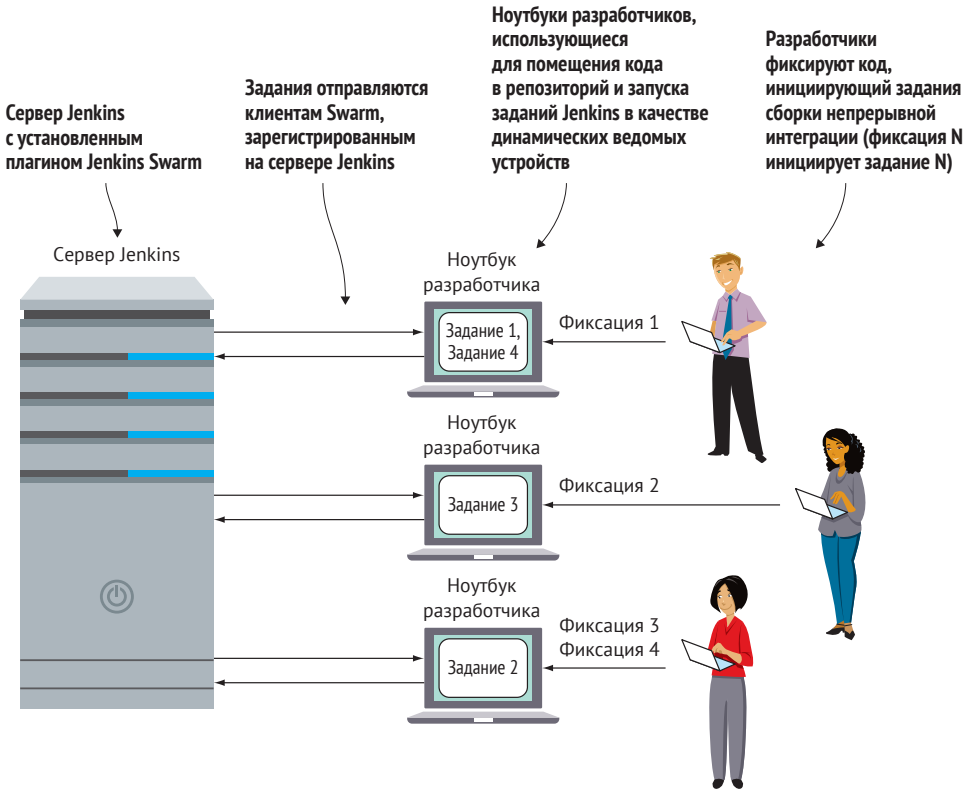


Рис. 8.8 ❖ После: масштабируемость при наличии команды разработчиков

Файл Dockerfile, показанный в листинге 8.13, создает образ с установленным плагином клиента Swarm, позволяя ведущему устройству Jenkins с соответствующим плагином сервера Swarm подключаться и запускать задания. Он начинается так же, как обычный Dockerfile ведомого устройства Jenkins, использованный в последнем методе.

Листинг 8.13. Файл Dockerfile

```
FROM ubuntu:16.04
ENV DEBIAN_FRONTEND noninteractive
RUN groupadd -g 1000 jenkins_slave
RUN useradd -d /home/jenkins_slave -s /bin/bash \
-m jenkins_slave -u 1000 -g jenkins_slave
RUN echo jenkins_slave:jpass | chpasswd
RUN apt-get update && apt-get install -y \
openssh-server openjdk-8-jre wget iproute2
RUN wget -O /home/jenkins_slave/swarm-client-3.4.jar \
 https://repo.jenkins-ci.org/releases/org/jenkins-ci/plugins/swarm-client
```

Получает плагин Swarm

```
➥ /3.4/swarm-client-3.4.jar
COPY startup.sh /usr/bin/startup.sh
 RUN chmod +x /usr/bin/startup.sh
 ENTRYPOINT ["/usr/bin/startup.sh"]
```

Копирует сценарий запуска в контейнер

Помечает сценарий запуска как исполняемый

Делает сценарий запуска командой по умолчанию

Следующий листинг – это сценарий запуска, скопированный в предыдущий файл Dockerfile.

Листинг 8.14. startup.sh

Использует IP-адрес хоста в качестве IP-адреса сервера Jenkins, если только JENKINS_IP не был установлен в среде вызова этого сценария

Определяет IP-адрес хоста

```
#!/bin/bash
export HOST_IP=$(ip route | grep ^default | awk '{print $3}')
 export JENKINS_IP=${JENKINS_IP:-$HOST_IP}
 export JENKINS_PORT=${JENKINS_PORT:-8080}
 export JENKINS_LABELS=${JENKINS_LABELS:-swarm}
 export JENKINS_HOME=${JENKINS_HOME:-$HOME}
 echo "Starting up swarm client with args:"
echo "$@"
echo "and env:"
echo "$(env)"
set -x
 java -jar \
 /home/jenkins_slave/swarm-client-3.4.jar \
  -sslFingerprints '[]' \
  -fsroot "$JENKINS_HOME" \
   -labels "$JENKINS_LABELS" \
   -master http://$JENKINS_IP:$JENKINS_PORT "$@"
```

Устанавливает порт Jenkins на 8080 по умолчанию

Устанавливает метку "swarm" для этого ведомого устройства

Регистрирует команды, которые запускаются отсюда как часть вывода сценария

Устанавливает домашний каталог Jenkins в домашний каталог пользователя jenkins_slave по умолчанию

Запускает клиент Jenkins Swarm

Устанавливает корневой каталог в домашний каталог Jenkins

Устанавливает метку для идентификации клиента для заданий

Устанавливает сервер Jenkins, на который будет указывать ведомое устройство

Большая часть предыдущего сценария устанавливает и выводит среду для вызова Java в конце. Вызов Java запускает клиент Swarm, который превращает компьютер, где он работает, в динамическое ведомое устройство Jenkins в каталоге, указанном во флаге `-fsroot`, и запускает задания, помеченные флагом `-labels` и указывающим на сервер Jenkins, обозначенный с помощью флага `-master`. Строки, содержащие `echo`, просто предоставляют некоторую отладочную информацию об аргументах и настройке среды.

Сборка и запуск контейнера – это простой вопрос запуска того, что должно быть привычным для вас шаблоном:

```
$ docker build -t jenkins_swarm_slave .
$ docker run -d --name \
jenkins_swarm_slave jenkins_swarm_slave \
-username admin -password adminpassword
```

Имя пользователя и пароль должны быть учетной записью в вашем экземпляре Jenkins с разрешением на создание ведомых устройств – учетная запись `admin` будет работать, но вы также можете создать еще одну учетную запись для этой цели.

Теперь, когда у вас на компьютере есть ведомое устройство, можете запускать на нем задания Jenkins.

Настройте задание как обычно, но добавьте `swam` в качестве выражения метки в разделе **Restrict Where This Project Can Be Run** (Ограничить место выполнения проекта) (см. метод 67).

> **ПРЕДУПРЕЖДЕНИЕ.** Задания Jenkins могут быть обременительными процессами, и вполне возможно, что их выполнение отрицательно скажется на лэптопе. Если задание интенсивное, вы можете соответствующим образом установить метки для заданий и клиентов Swarm. Например, можно установить для задания метку 4CPU8G и сопоставить ее с контейнерами Swarm, работающими на компьютерах с четырехъядерным процессором и 8 Гб памяти.

Этот метод дает некоторое представление о концепции Docker. Предсказуемая и переносимая среда может быть размещена на нескольких хостах, снижая нагрузку на дорогой сервер и сводя к минимуму необходимую конфигурацию.

Хотя это не тот метод, который можно применять без учета производительности, мы считаем, что здесь есть широкие возможности для превращения компьютерных ресурсов разработчиков в форму игры, повышающую эффективность в организации, занимающейся вопросами разработки без необходимости в новом дорогостоящем оборудовании.

ОБСУЖДЕНИЕ

Вы можете автоматизировать этот процесс, настроив его в качестве контролируемой системной службы на всех компьютерах, имеющихся в вашем распоряжении (см. метод 82).

> **ПРИМЕЧАНИЕ.** Код для этого метода и связанных с ним методов доступен на GitHub по адресу https://github.com/docker-in-practice/jenkins.

МЕТОД 69 Безопасное обновление контейнеризованного сервера Jenkins

Если вы какое-то время работали Jenkins, то наверняка знаете, что Jenkins часто публикует обновления для своего сервера для обеспечения безопасности и изменения функциональности.

На выделенном хосте, где не используется Docker, этим обычно управляет система управления пакетами. В случае с Docker может быть несколько сложнее рассуждать об обновлениях, поскольку вы, вероятно, отделили контекст сервера от его данных.

ПРОБЛЕМА

Вы хотите надежно обновить свой сервер Jenkins.

РЕШЕНИЕ

Запустите образ Jenkins updater, который будет обрабатывать обновление сервера Jenkins.

Этот метод поставляется в виде образа Docker, состоящего из нескольких частей.

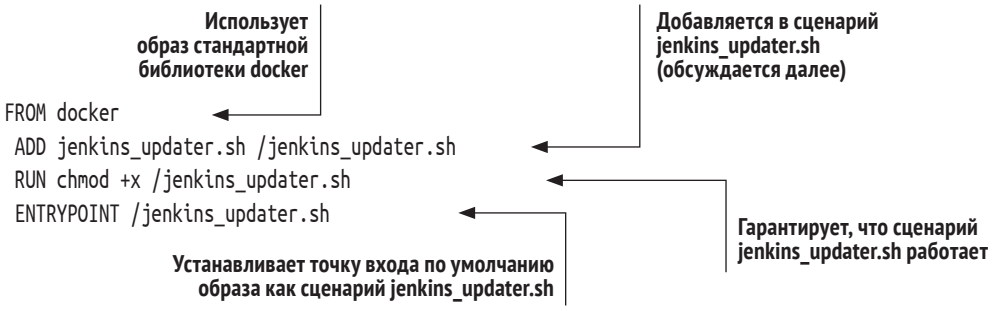
Сначала мы наметим файл Dockerfile, с помощью которого соберем образ. Этот файл извлекается из образа библиотеки (который содержит клиент Docker) и добавляет сценарий, управляющий обновлением.

Образ запускается в команде Docker, которая монтирует элементы Docker на хосте, давая ему возможность управлять любым необходимым обновлением Jenkins.

Dockerfile

Начнем с файла Dockerfile.

Листинг 8.15. Dockerfile для Jenkins updater



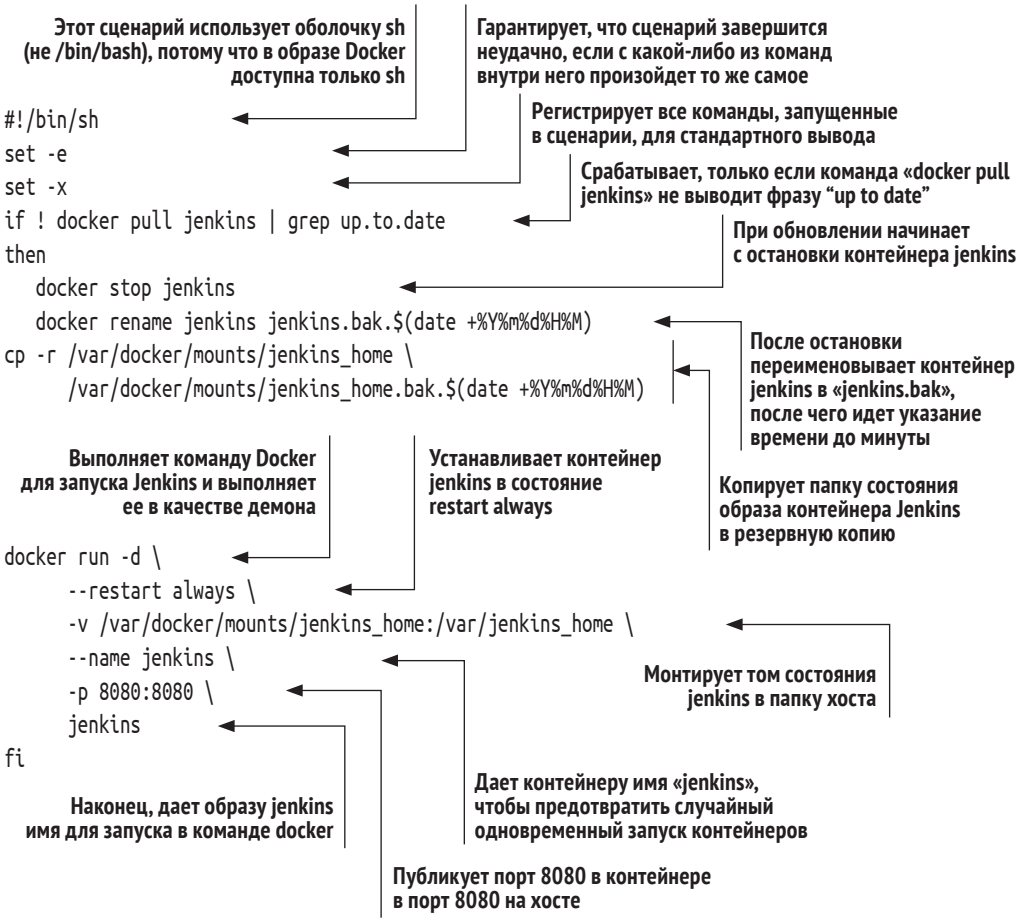
Предыдущий файл Dockerfile инкапсулирует требования для резервного копирования Jenkins в работающем образе Docker. Он использует образ стандартной библиотеки `docker` для запуска клиента Docker внутри контейнера. Этот контейнер будет запускать сценарий в листинге 8.16 для управления любым необходимым обновлением Jenkins на хосте.

> **ПРИМЕЧАНИЕ.** Если ваша версия демона Docker отличается от версии в образе `docker`, вы можете столкнуться с проблемами. Попробуйте использовать ту же версию.

jenkins_updater.sh

Это сценарий оболочки, который управляет обновлением внутри контейнера.

Листинг 8.16. Сценарий оболочки для резервного копирования и перезапуска Jenkins



Предыдущий сценарий пытается вытащить `jenkins` из Docker Hub с помощью команды `docker pull`. Если выходные данные содержат фразу «up to date», команда `docker pull | grep...` возвращает значение `true`. Но вы хотите обновляться только тогда, когда не видите этой фразы в выходных данных. Вот почему оператор `if` сопровождается символом `!` после `if`.

В результате код в блоке if запускается только в том случае, если вы скачали новую версию «последнего» образа Jenkins. Внутри этого блока работающий контейнер Jenkins останавливается и переименовывается. Вы переименовываете его, а не удаляете, в случае если обновление не работает и вам необходимо восстановить предыдущую версию. В дополнение к этой стратегии отката также выполняется резервное копирование папки монтирования на хосте, где находится состояние Jenkins.

Наконец, с помощью команды `docker run` запускается последний скачанный образ Jenkins.

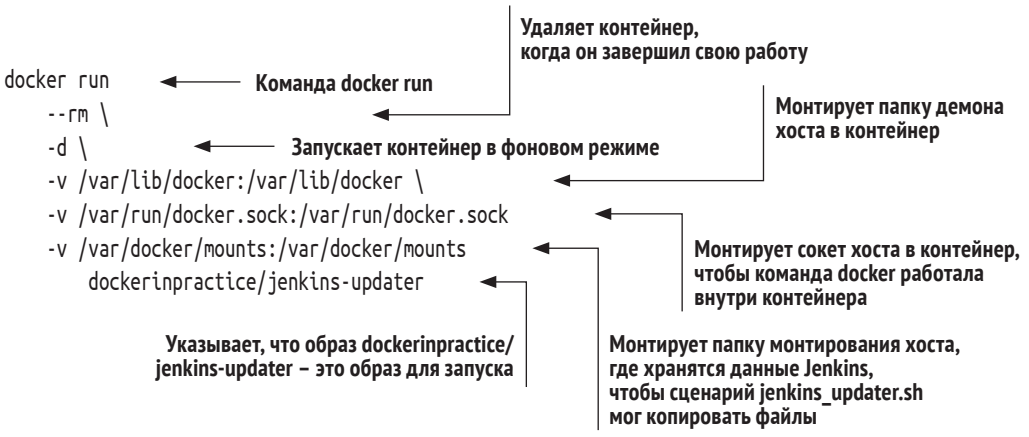
> **ПРИМЕЧАНИЕ.** Возможно, вы захотите изменить папку монтирования или имя работающего контейнера Jenkins на основе личных предпочтений.

Возможно, вам интересно, как этот образ Jenkins связывается с демоном хоста. Чтобы добиться этого, образ запускается с использованием способа, показанного в методе 66.

Вызов образа Jenkins-updater

Следующая команда выполнит обновление Jenkins, используя образ (с оболочкой сценария внутри него), который был создан ранее:

Листинг 8.17. Команда Docker для запуска Jenkins updater



Автоматизация обновления

Следующий однострочник облегчает запуск в crontab. Мы выполняем его на наших домашних серверах.

```
0 * * * * docker run --rm -d -v /var/lib/docker:/var/lib/docker -v
➥ /var/run/docker.sock:/var/run/docker.sock -v
➥ /var/docker/mounts:/var/docker/mounts dockerinpractice/jenkins-updater
```

> **ПРИМЕЧАНИЕ.** Предыдущая команда находятся целиком в одной строке, потому что crontab не игнорирует символы новой строки, если перед ними стоит обратная косая черта, как это делают сценарии оболочки.

Конечным результатом является то, что одна запись crontab может безопасно управлять обновлением вашего экземпляра Jenkins, не вызывая у вас беспокойства.

Задача по автоматизации очистки старых зарезервированных контейнеров и монтируемых томов оставлена для читателя в качестве упражнения.

ОБСУЖДЕНИЕ

Этот метод иллюстрирует ряд моментов, с которыми мы сталкиваемся на протяжении всей книги и которые могут применяться в аналогичных контекстах к ситуациям, отличным от Jenkins.

Во-первых, он использует основной образ `docker` для обмена данными с демоном Docker на хосте. Можно было бы написать другие переносимые сценарии, чтобы управлять демонами Docker иными способами. Например, сценарии для удаления старых томов или такие, чтобы сообщить об активности вашего демона.

Более конкретно шаблон блока if можно использовать для обновления и перезапуска других образов, когда доступен новый. Нередко образы обновляются по соображениям безопасности или для незначительных обновлений.

Если вы обеспокоены трудностями при обновлении версий, также стоит указать, что вам не нужно брать тег «последнего» образа (что сделано в этом методе). У многих образов разные теги, которые отслеживают разные номера версий. Например, у вашего образа `exampleimage` может быть тег `exampleimage:latest`, а также `exampleimage:v1.1` и `exampleimage: v1`. Любой из них может быть обновлен когда угодно, но тег `: v1.1` с меньшей вероятностью перейдет на новую версию, чем `latest`. Тег `latest` может перейти в ту же версию, что и новый тег: `v1.2` (для этого потребуется обновление) или даже тег: `v2.1`, где новая основная версия 2 указывает на изменение, которое с большей вероятностью может нарушить работу любого процесса обновления.

Этот метод также описывает стратегию отката для обновлений Docker. Разделение контейнера и данных (с использованием монтируемых томов) может создать напряженность в отношении стабильности любого обновления. Сохранив старый контейнер и копию старых данных в том месте, где работала служба, легче будет восстановиться после сбоя.

Обновление базы данных и Docker

Обновления базы данных представляют собой особый контекст, в котором актуальны проблемы стабильности. Если вы хотите обновить свою базу данных до новой версии, нужно подумать, требует ли обновление изменения структур данных и хранилища данных базы.

Недостаточно запустить образ новой версии в качестве контейнера и ожидать, что он будет работать. Становится немного сложнее, если база данных достаточно умна, чтобы знать, какую версию данных она просматривает, и может выполнить соответствующее обновление самостоятельно. В этих случаях, возможно, вам удобнее сделать обновление.

На стратегию обновления влияет множество факторов. Ваше приложение может мириться с оптимистическим подходом (как видно здесь в примере с Jenkins), который предполагает, что все будет хорошо, и готовится к неудаче, когда (не если) это происходит. С другой стороны, вы можете требовать 100 % безотказной работы и не мириться с какими бы то ни было неудачами. В таких случаях, как правило, желательно иметь полностью протестированный план обновления и более глубокое знание платформы вместо выполнения команды `docker pull` (с участием или без участия Docker).

Хотя Docker не устраняет проблему обновления, неизменность версионных образов может упростить процесс рассуждения о них. Docker также может помочь вам подготовиться к сбоям двумя способами: выполнить резервное копирование состояния на томах хоста и упростить тестирование предсказуемого состояния. Удар, который вы принимаете при управлении и понимании того, что делает Docker, может дать вам больше контроля и уверенности относительно процесса обновления.

Резюме

- Вы можете использовать рабочий процесс Docker Hub, чтобы автоматически инициировать запуск сборок при изменении кода.
- Сборки можно значительно ускорить, используя eatmydata и кеши пакетов.
- На ускорение сборок также влияет использование кешей прокси-серверов для внешних артефактов, таких как системные пакеты.
- Вы можете запускать тесты с применением графического интерфейса пользователя (например, Selenium) внутри Docker.
- Ваша платформа непрерывной интеграции (например, Jenkins) может быть запущена из контейнера.
- Ведомое устройство непрерывной интеграции Docker позволяет вам полностью контролировать свою среду.
- Вы можете передать процессы сборки всей команде, используя Docker и плагин Swarm.

Глава 9

Непрерывная доставка: идеальная совместимость с принципами Docker

О чем рассказывается в этой главе:

- контракт Docker между разработчиками программного обеспечения и ИТ-операторами;
- ручное управление доступностью сборки в разных средах;
- перемещение сборок между средами по соединениям с низкой пропускной способностью;
- централизованная настройка всех контейнеров в среде;
- достижение развертывания с нулевым временем простоя с помощью Docker.

Как только вы убедитесь, что все ваши сборки проверяются на качество с помощью согласованного процесса непрерывной интеграции, следующим логическим шагом будет начать рассмотрение развертывания каждой удачной сборки для ваших пользователей. Эта цель известна как непрерывная доставка.

В этой главе мы будем ссылаться на «конвейер непрерывной доставки» – процесс, который проходит ваша сборка, после того как выходит из «конвейера непрерывной интеграции». Разделительная линия иногда может быть размыта, но рассматривайте конвейер непрерывной доставки как конвейер, который начинается, когда у вас есть окончательный образ, прошедший начальные тесты в процессе сборки. На рис. 9.1 показано, как образ может продвигаться по конвейеру непрерывной доставки, пока он (надеюсь) не достигнет этапа эксплуатации.

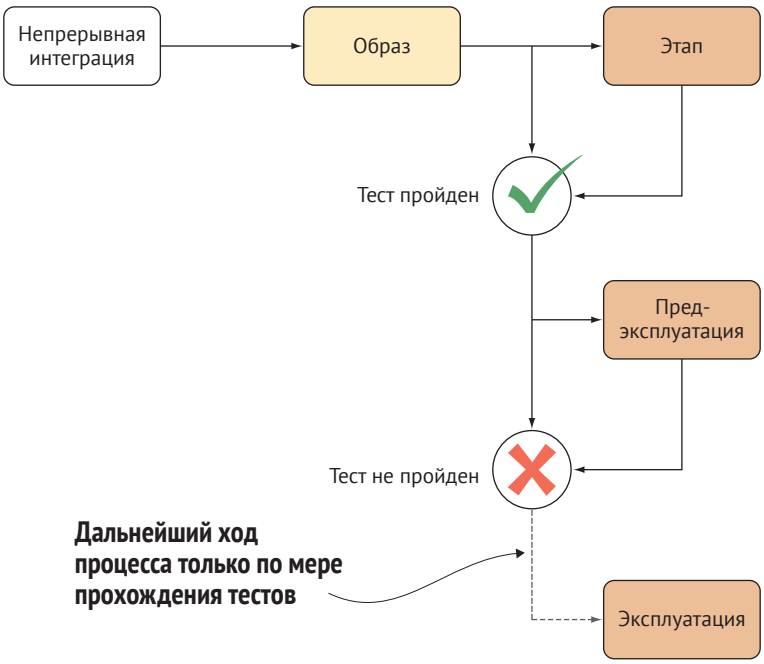


Рис. 9.1 ❖ Типичный конвейер непрерывной доставки

Стоит повторить последний момент – образ, получаемый из непрерывной интеграции, должен быть окончательным и неизмененным на протяжении всего процесса непрерывной доставки! Docker позволяет легко реализовать это благодаря неизменяемым образам и инкапсуляции состояния, поэтому, используя Docker, вы уже продвигаетесь на шаг вперед на пути к непрерывной доставке.

Закончив эту главу, вы полностью поймете, почему неизменность Docker делает его идеальным партнером для вашей стратегии непрерывной доставки. Таким образом, Docker может стать ключевым фактором для любой стратегии DevOps в любой организации.

9.1. Взаимодействие с другими командами в конвейере непрерывной доставки

Сначала мы сделаем небольшой шаг назад и посмотрим, как Docker меняет отношения между разработчиками программного обеспечения и ИТ-операторами.

Некоторые крупнейшие проблемы разработки программного обеспечения- это не технические проблемы: разделение людей на команды на основе их ролей и опыта является обычной практикой, однако это может привести к коммуникационным барьерам и изолированности. Наличие успешного конвейера непрерывной доставки требует участия команд на всех этапах процесса:

от разработки до тестирования и эксплуатации. Наличие единой контрольной точки для всех команд может способствовать этому взаимодействию, предоставляя структуру.

МЕТОД 70 Контракт Docker: устранение разногласий

Одна из целей Docker – обеспечить простое выражение входных и выходных данных, поскольку они относятся к контейнеру, содержащему одно приложение. Это может обеспечить ясность при работе с другими людьми – общение является важной частью сотрудничества, и понимание того, как Docker может упростить ситуацию, предоставив единую точку отсчета, может помочь вам победить скептиков Docker.

ПРОБЛЕМА

Вы хотите, чтобы результаты совместной работы были гладкими и однозначными и устранились разногласия в конвейере доставки.

РЕШЕНИЕ

Используйте *контракт* Docker, чтобы облегчить наличие гладких результатов между командами.

По мере того как компании меняются, они часто обнаруживают, что плоская, скудная организация, которая у них когда-то была, где ключевые лица «знали всю систему», уступает место более структурированной организации, в которой разные команды имеют разные обязанности и компетенции.

Мы непосредственно видели это в организациях, где работали.

Если не будут сделаны технические инвестиции, могут возникнуть разногласия, по мере того как растущие команды осуществляют поставки друг другу. Жалобы на увеличивающуюся сложность, «швыряние релиза в стену» и некорректно работающие обновления – все это становится знакомым. Все чаще будут раздаваться крики: «ну, это работает на нашей машине!», к разочарованию всех заинтересованных сторон. На рис. 9.2 приводится упрощенный, но репрезентативный вид этого сценария.

У рабочего процесса на рис. 9.2 имеется ряд проблем, которые могут показаться вам знакомыми. Все они сводятся к трудностям управления состоянием. Команда тестирования может протестировать что-либо на компьютере, что отличается от того, что было установлено операционной группой. Теоретически изменения во всех средах должны быть тщательно задокументированы, отменены, если видны проблемы, и оставаться неизменными. К сожалению, реалии коммерческого давления и поведения человека обычно вступают в сговор против этой цели, и наблюдается медленное перемещение среды.

Существующие решения этой проблемы включают в себя виртуальные машины и пакеты RPM. Виртуальные машины могут быть использованы для уменьшения площади опасного воздействия на среду, если дать другим командам сотрудников полное представление о них. Недостатком является

то, что виртуальные машины – относительно монолитные объекты, которыми командам сложно управлять эффективно. С другой стороны, пакеты RPM предлагают стандартный способ упаковки приложений, который помогает определять зависимости при развертывании программного обеспечения. Это, однако, не устраняет проблемы с управлением конфигурацией, а развертывание пакетов RPM, созданных коллегами, гораздо более подвержено ошибкам, чем использование RPM, которые были проверены в действии через интернет.

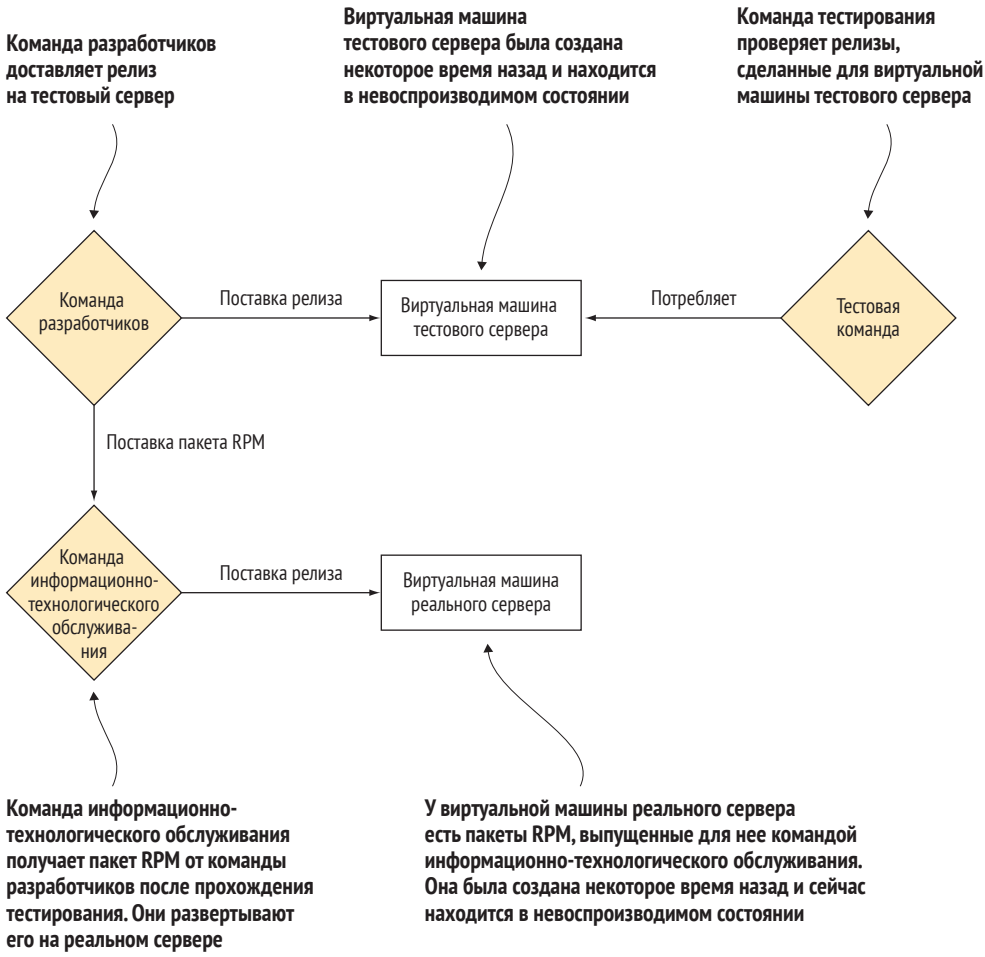


Рис. 9.2 ❖ Прежде: типичный рабочий процесс программного обеспечения

Контракт Docker

Docker может предоставить вам четкую линию разграничения между группами, где образ Docker является одновременно границей и единицей обмена. Мы называем это *контрактом Docker*. Он показан на рис. 9.3.

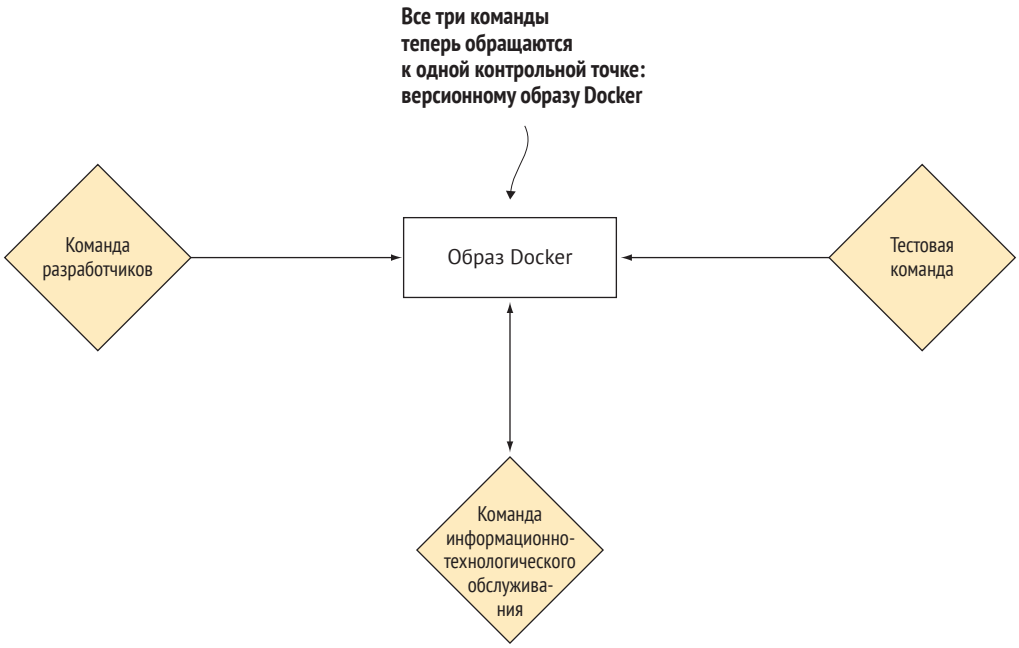


Рис. 9.3 ❖ После: контракт Docker

С использованием Docker ориентир для всех групп становится намного яснее. Вместо того чтобы иметь дело с растягивающимися монолитными виртуальными (или реальными) машинами в невоспроизводимых состояниях, все группы говорят об одном и том же коде, будь то тестирование, работа в реальном времени или разработка.

Кроме того, существует четкое разделение данных и кода, благодаря чему легче судить, вызваны ли проблемы изменениями в данных или коде.

Поскольку Docker использует удивительно стабильный API Linux в качестве своей среды, группы, поставляющие программное обеспечение, имеют гораздо больше свободы для создания программного обеспечения и служб любым удобным для них способом, будучи уверенными в том, что они станут работать предсказуемо в различных средах. Это не означает, что вы можете игнорировать контекст, в котором работает ПО, но это снижает риск возникновения проблем из-за различий в среде.

Различная эффективность достигается благодаря наличию этой единственной контрольной точки.

Воспроизведение ошибок становится намного проще, поскольку все команды могут описывать и воспроизводить проблемы из известной отправной точки. Обновления становятся обязанностью команды, поставляющей изменения. Говоря кратко, состояние управляется теми, кто вносит изменения. Все эти преимущества значительно уменьшают накладные расходы на связь и позволяют командам продолжить свою работу, что также может способствовать переходу к архитектуре микросервисов.

Это не просто теоретическое преимущество: мы видели улучшение воочию в компании, где работает более 500 разработчиков, и это часто обсуждается на технических собраниях Docker.

ОБСУЖДЕНИЕ

Данный метод обрисовывает в общих чертах стратегию, которую полезно помнить, когда вы продолжаете читать книгу, чтобы определить, как другие методы вписываются в этот новый мир. Например, метод 76 описывает способ запуска приложения на основе микросервисов в том же кросс-контейнерном режиме, что и в производственных системах, устраняя источник настройки файла конфигурации. Когда вы столкнетесь с внешними URL-адресами или другими неизменными факторами в разных средах, метод 85 предоставит информацию об обнаружении службы – хороший способ превратить разрастание файлов конфигурации в единый источник истины.

9.2. Облегчение развертывания образов Docker

Первая проблема при попытке реализовать непрерывную доставку – это перемещение результатов процесса сборки в соответствующее место. Если вы можете использовать один реестр для всех этапов конвейера непрерывной доставки, может показаться, что проблема решена. Но это решение не охватывает один ключевой аспект.

Одной из основных идей непрерывной доставки является *продвижение сборки (build promotion)*: каждый этап конвейера (пользовательские приемочные тесты, интеграционные тесты и тесты производительности) может инициировать следующий этап, только если предыдущий был успешным. При наличии нескольких реестров вы можете убедиться, что используются исключительно продвигаемые сборки, едва сделав их доступными в следующем реестре, когда пройдет этап сборки.

Мы рассмотрим несколько способов перемещения ваших образов между реестрами и даже способ совместного использования объектов Docker без реестра.

МЕТОД 71 Зеркальное отображение образов реестра вручную

Самый простой сценарий зеркального отображения образов – это когда у вас есть компьютер с высокоскоростным подключением к обоим реестрам. Это позволяет использовать обычные функции Docker для копирования образов.

ПРОБЛЕМА

Вы хотите скопировать образ из одного реестра в другой.

РЕШЕНИЕ

Вручную используйте стандартные команды извлечения и размещения для передачи образа.

Это решение включает в себя:

- извлечение образа из реестра;
- повторное присваивание метки образу;
- размещение повторно помеченного образа.

Если у вас есть образ на странице test-registry.company.com и вы хотите переместить его на stageregistry.company.com, это простой процесс.

Листинг 9.1. Перенос образа из теста в промежуточный реестр

```
$ IMAGE=mygroup/myimage:mytag
$ OLDREG=test-registry.company.com
$ NEWREG=stage-registry.company.com
$ docker pull $OLDREG/$MYIMAGE
[...]
$ docker tag -f $OLDREG/$MYIMAGE $NEWREG/$MYIMAGE
$ docker push $NEWREG/$MYIMAGE
$ docker rmi $OLDREG/$MYIMAGE
$ docker image prune -f
```

Здесь следует отметить три важных момента:

1. Новый образ был помечен принудительно. Это означает, что любой старый образ с тем же именем на компьютере (оставленный там для кеширования слоев) потеряет имя, поэтому новый может быть помечен желаемым именем.
2. Все повисшие образы были удалены. Хотя кеширование слоев чрезвычайно полезно для ускорения развертывания, если вы оставите слои неиспользуемых образов, это может привести к тому, что место на диске быстро будет занято. В целом старые слои с меньшей вероятностью будут использоваться с течением времени, и они становятся более устаревшими.
3. Вам может потребоваться войти в новый реестр с помощью команды `docker login`.

Образ теперь доступен в новом реестре для использования на последующих этапах вашего конвейера непрерывной доставки.

ОБСУЖДЕНИЕ

Этот метод иллюстрирует простой момент касательно тегирования в Docker: сам тег содержит информацию о реестре, которому он принадлежит.

Большую часть времени это скрыто от пользователей, потому что они обычно извлекают данные из реестра по умолчанию (Docker Hub на docker.io). Когда вы начинаете работать с реестрами, эта проблема выходит на первый план, потому что вам необходимо явно пометить реестр с помощью его местоположения, чтобы поместить его в правильную конечную точку.

МЕТОД 72 Доставка образов через ограниченные соединения

Даже при использовании концепции слоев извлечение и размещение образов Docker может стать процессом, которому не хватает пропускной способности. В мире бесплатных соединений с большой пропускной способностью это не будет проблемой, но иногда реальность заставляет нас иметь дело с соединениями с низкой пропускной способностью или дорогостоящим измерением показателей пропускной способности между центрами обработки данных. В этой ситуации вам нужно найти более эффективный способ передачи различий, иначе непрерывная доставка, идеально позволяющая запускать конвейер несколько раз в день, останется вне досягаемости.

Оптимальным решением является инструмент, который уменьшает средний размер образа, поэтому он даже меньше того, с чем могут управляться классические методы сжатия.

ПРОБЛЕМА

Вы хотите скопировать образ с одного компьютера на другой, между которыми существует соединение с низкой пропускной способностью.

РЕШЕНИЕ

Экспортируйте образ, разделите его, передайте фрагменты и импортируйте вновь объединенный образ на другом конце.

Чтобы сделать все это, мы должны сначала познакомиться с новым инструментом: bup. Он был создан как инструмент резервного копирования с чрезвычайно эффективной дедупликацией. *Дедупликация* – это способность распознавать, где данные используются повторно, и сохранять их только один раз. Особенно хорошо он работает с архивами, содержащими ряд похожих файлов, что является удобным форматом, в котором Docker позволяет экспортировать образы.

Для этого метода мы создали образ под названием dbup (сокращение от Docker bup), которое упрощает использование bup для дедупликации образов. Вы можете найти его код на странице https://github.com/docker-in-practice/dbup.

В качестве демонстрации давайте посмотрим, сколько пропускной способности мы могли бы сэкономить при обновлении образа ubuntu: 14.04.1 до ubuntu: 14.04.2. Имейте в виду, что на практике у вас будет несколько слоев поверх каждого из них, которые Docker захочет полностью перенести после смены нижнего слоя. С другой стороны, этот метод распознает существенные сходства и дает вам гораздо большую экономию по сравнению со следующим примером.

Первый шаг – извлечение обоих этих образов, чтобы мы могли увидеть, сколько передается по сети.

Листинг 9.2. Изучение и сохранение двух образов Ubuntu

```
$ docker pull ubuntu:14.04.1 && docker pull ubuntu:14.04.2
[...]
$ docker history ubuntu:14.04.1
IMAGE         CREATED       CREATED BY                                      SIZE
ab1bd63e0321  2 years ago   /bin/sh -c #(nop) CMD [/bin/bash]               0B
<missing>     2 years ago   /bin/sh -c sed -i 's/^#\s*\(deb.*universe\...   1.9kB
<missing>     2 years ago   /bin/sh -c echo '#!/bin/sh' > /usr/sbin/po...   195kB
<missing>     2 years ago   /bin/sh -c #(nop) ADD file:62400a49cced0d7...   188MB
<missing>     4 years ago                                                   0B
$ docker history ubuntu:14.04.2
IMAGE         CREATED       CREATED BY                                      SIZE
44ae5d2a191e  2 years ago   /bin/sh -c #(nop) CMD ["/bin/bash"]             0B
<missing>     2 years ago   /bin/sh -c sed -i 's/^#\s*\(deb.*universe\...   1.9kB
<missing>     2 years ago   /bin/sh -c echo '#!/bin/sh' > /usr/sbin/po...   195kB
<missing>     2 years ago   /bin/sh -c #(nop) ADD file:0a5fd3a659be172...   188MB
$ docker save ubuntu:14.04.1 | gzip | wc -c
65973497
$ docker save ubuntu:14.04.2 | gzip | wc -c
65994838
```

Нижний слой на каждом образе (ADD) составляет большую часть размера, и видно, что добавляемый файл отличается, поэтому вы можете рассматривать весь размер образа как величину, которая будет передана при заливке нового образа. Также обратите внимание на то, что реестр Docker использует сжатие gzip для передачи слоев, поэтому мы включили его в наши измерения (вместо того чтобы брать размер из `docker history`). При первоначальном и последующем развертывании передается около 65 Мб.

Для начала вам понадобятся две вещи: каталог для хранения пула данных, используемый bup в качестве хранилища, и образ dockerinpractice/dbup. После этого можете приступить и добавить свой образ в пул данных bup.

Листинг 9.3. Сохранение двух образов Ubuntu в пул данных bup

```
$ mkdir bup_pool
$ alias dbup="docker run --rm \
    -v $(pwd)/bup_pool:/pool -v /var/run/docker.sock:/var/run/docker.sock \
    dockerinpractice/dbup"
$ dbup save ubuntu:14.04.1
Saving image!
Done!
$ du -sh bup_pool
74M     bup_pool
$ dbup save ubuntu:14.04.2
```

```
Saving image!
Done!
$ du -sh bup_pool
96M     bup_pool
```

Добавление второго образа в пул данных bup увеличило размер примерно на 20 Мб. Предполагая, что вы синхронизировали папку с другим компьютером (возможно, с помощью rsync) после добавления ubuntu: 14.04.1, повторная синхронизация папки будет передавать только 20 Мб (в отличие от 65 Мб ранее).

Затем вам нужно загрузить образ на другом конце.

Листинг 9.4. Загрузка образа из пула данных bup

```
$ dbup load ubuntu:14.04.1
Loading image!
Done!
```

Процесс переноса между реестрами будет выглядеть примерно так:

1. `docker pull` на хосте 1.
2. `dbup save` на хосте 1.
3. `rsync` из хоста 1 в хоста 2.
4. `dbup load` на хосте 2.
5. `docker push` на хосте 2.

Этот метод открывает ряд возможностей, которые раньше были нереальны. Например, теперь вы можете переставлять и объединять слои, не беспокоясь о том, сколько времени потребуется для передачи всех новых слоев по соединению с низкой пропускной способностью.

ОБСУЖДЕНИЕ

Даже если следовать последним рекомендациям и добавить код приложения в качестве последнего этапа, bup может помочь – он распознает, что большая часть кода не изменилась, и добавит разницу только в пул данных.

Пулы данных могут быть очень большими, как, например, файлы базы данных, и bup, вероятно, будет работать очень хорошо (что полезно, если вы решили использовать метод 77 с базой данных внутри контейнера, а это означает отсутствие тома). Это на самом деле несколько необычно – экспорт и резервное копирование баз данных обычно очень эффективны для постепенной передачи, но фактическое хранение баз данных на диске может значительно различаться и иногда приводит к поражению таких утилит, как rsync. Кроме того, dbup предоставляет вам полную историю образов – не нужно хранить три полных копии образов для отката. Вы можете извлечь их из пула на досуге. К сожалению, в настоящее время нет способа очистить пул от образов, которые вам больше не нужны, поэтому, вероятно, придется очищать пул время от времени.

Хотя вы, возможно, не видите немедленной необходимости в использовании dbup, имейте ее в виду на случай, если ваши счета за пропускную способность начнут расти.

МЕТОД 73 Совместное использование объектов Docker в виде TAR-файлов

Файлы TAR – это классический метод перемещения файлов в Linux. Docker позволяет создавать эти файлы и отправлять их вручную, когда нет доступного реестра и нет возможности его создать. Здесь мы покажем вам все тонкости этих команд.

ПРОБЛЕМА

Вы хотите делиться образами и контейнерами с другими, не имея доступного реестра.

РЕШЕНИЕ

Используйте команды `docker export` или `docker save` для создания артефактов файлов TAR, а затем примените их с командами `docker import` или `docker load` через протокол SSH.

Различия между командами может быть трудно понять, если вы используете их небрежно, поэтому давайте на минутку быстро рассмотрим, что они делают. Посмотрите на табл. 9.1.

Таблица 9.1 ❖ Сравнение команд export и import с командами save и load

Команда	Создает	С чем работает команда	Откуда
`export`	TAR-файл	Контейнерная файловая система	Контейнер
`import`	Образ Docker	Плоская файловая система	TAR-файл
`save`	TAR-файл	Образ Docker (с историей)	Образ
`load`	Образ Docker	Образ Docker (с историей)	TAR-файл

Первые две команды работают с плоскими файловыми системами. Команда `docker export` выводит TAR-файл из файлов, которые составляют состояние контейнера. Как обычно в случае с Docker, состояние запущенных процессов не сохраняется – только файлы. Команда `docker import` создает образ Docker – без истории или метаданных – из TAR-файла.

Данные команды не являются симметричными – нельзя создать контейнер из уже существующего, используя только `import` и `export`. Эта асимметрия может быть полезна, поскольку позволяет экспортировать образ в TAR-файл с помощью команды `docker export`, а затем импортирует его с помощью `docker import`, чтобы «избавиться» ото всей истории слоев и метаданных. Это подход сращивания образов, описанный в методе 52.

Если вы экспортируете или сохраняете в TAR-файл, по умолчанию файл отправляется в `stdout`, поэтому обязательно сохраните его в файл, подобный этому:

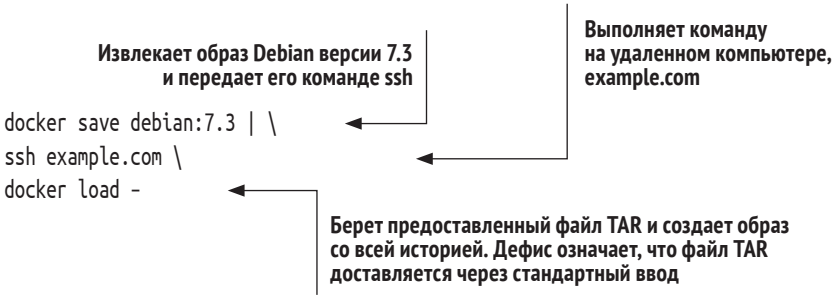
```
docker pull debian:7:3
[...]
docker save debian:7.3 > debian7_3.tar
```

TAR-файл, как тот, что был только что создан, можно безопасно перебросить по сети (хотя вы, возможно, захотите сначала сжать его с помощью gzip), и другие люди способны использовать его для импорта образов без изменений. Они могут быть отправлены по электронной почте или с помощью команды `scp`, если у вас есть доступ:

```
$ scp debian7_3.tar example.com:/tmp/debian7_3.tar
```

Вы можете сделать еще один шаг вперед и напрямую доставлять образы демонам Docker других пользователей – при условии, что у вас есть права.

Листинг 9.5. Отправка образа напрямую через SSH



Если вы хотите отказаться от истории образа, то можете использовать команду `import` вместо `load`.

Листинг 9.6. Передача образа Docker напрямую через SSH, отбрасывая слои

```
docker export $(docker run -d debian:7.3 true) | \
    ssh example.com docker import
```

> **ПРИМЕЧАНИЕ.** В отличие от `docker import`, `docker export` не требует тире в конце, чтобы указать, что TAR-файл доставляется через стандартный ввод.

ОБСУЖДЕНИЕ

Возможно, вы помните процесс экспорта и импорта из метода 52, где видели, как сращивание образов можно использовать для удаления секретов,

которые могут быть скрыты в нижних слоях. Тот факт, что секреты бывают доступны на нижних слоях, стоит иметь в виду, если вы передаете образы другим людям, – осознание того, что вы удалили свой открытый ключ на верхнем слое образа, но он доступен ниже, может стать настоящей проблемой, поскольку тогда вам придется рассматривать его как взломанный и менять его везде.

Если вы обнаружите, что в этом методе часто выполняете передачу образов, возможно, стоит потратить немного времени на метод 9, чтобы настроить собственный реестр и сделать вещи менее специализированными.

9.3. Настройка ваших образов для среды

Как упоминалось во введении к этой главе, одним из ключевых моментов непрерывной доставки является концепция «делать везде одно и то же». Без Docker это означало бы создание артефакта развертывания один раз и использование одного и того же артефакта повсюду. В мире Docker это означает использование одного и того же образа везде.

Но среды не все одинаковые – например, для внешних служб могут быть разные URL-адреса. Для «обычных» приложений вы можете использовать переменные среды для решения этой проблемы (с учетом того, что их нелегко применять на многочисленных компьютерах). То же решение может работать и для Docker (явная передача переменных), но есть способ сделать это лучше с помощью Docker, и у этого способа имеются некоторые дополнительные преимущества.

МЕТОД 74 Информирование контейнеров с помощью etcd

Образы Docker спроектированы таким образом, чтобы их можно было развернуть где угодно, но после развертывания часто требуется добавлять дополнительную информацию, чтобы влиять на поведение приложения во время его работы. Кроме того, может потребоваться оставить неизменными компьютеры, работающие под управлением Docker, поэтому вам понадобится внешний источник информации (создание переменных среды менее подходит).

ПРОБЛЕМА

Вам нужен внешний источник конфигурации при запуске контейнеров.

РЕШЕНИЕ

Установите etcd, распределенное хранилище типа «ключ/значение» для хранения конфигурации контейнера.

etcd содержит фрагменты информации и может быть частью многоузлового кластера для обеспечения устойчивости. В этом методе вы создадите кластер etcd для хранения вашей конфигурации и будете использовать прокси-сервер etcd для доступа к нему.

> **ПРИМЕЧАНИЕ.** Каждое значение, хранящееся в etcd, должно быть маленьким. Хорошее правило – менее 512 Кб. После этого вам следует подумать о сопоставительном анализе, чтобы убедиться, что etcd все еще работает так, как вы ожидаете. Этот предел не уникален для etcd. Вы должны иметь это в виду при работе с другими хранилищами типа «ключ/значение», такими как Zookeeper и Consul.

Поскольку узлы кластера etcd должны общаться друг с другом, первым шагом является определение вашего внешнего IP-адреса. Если вы собираетесь запускать узлы на разных компьютерах, потребуется внешний IP для каждого из них.

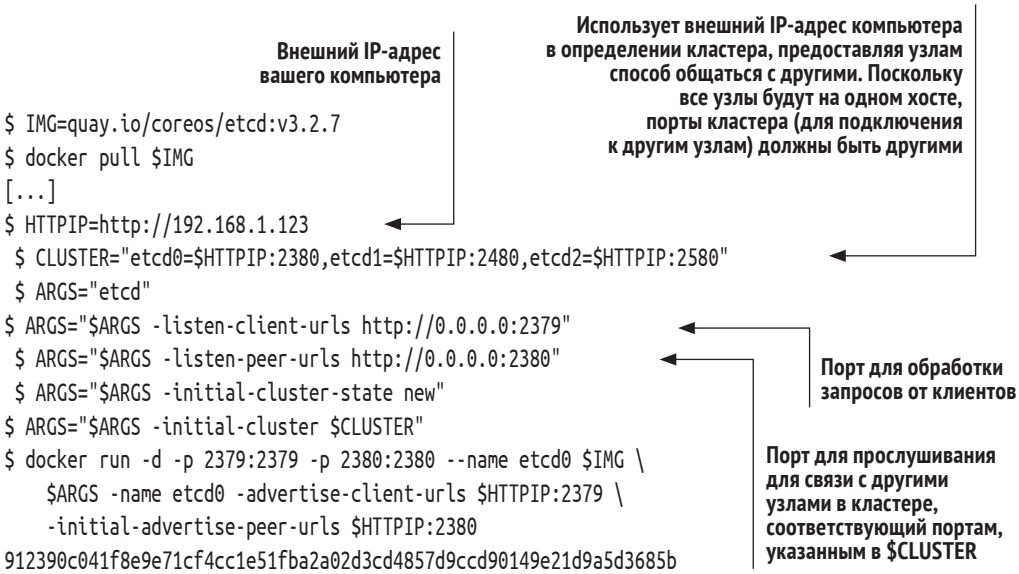
Листинг 9.7. Идентификация IP-адресов локальной машины

```
$ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$'
    inet 192.168.1.123/24 brd 192.168.1.255 scope global dynamic wlp3s0
    inet 172.18.0.1/16 scope global br-0c3386c9db5b
```

Здесь мы рассмотрели все интерфейсы IPv4 и исключили LoopBack и Docker. Верхняя строка (первый IP-адрес в этой строке) – это строка, которая вам нужна. Она представляет компьютер в локальной сети – попробуйте пропинговать ее с другого компьютера, если вы не уверены.

Теперь мы можем начать работу с трехузловым кластером, целиком работающим на том же компьютере. Будьте осторожны со следующими аргументами – порты, которые открываются и объявляются, меняются в каждой строке, так же, как и имена узлов и контейнеров кластера.

Листинг 9.8. Настройка трехузлового кластера etcd



```
$ docker run -d -p 2479:2379 -p 2480:2380 --name etcd1 $IMG \
    $ARGS -name etcd1 -advertise-client-urls $HTTPIP:2479 \
    -initial-advertise-peer-urls $HTTPIP:2480
446b7584a4ec747e960fe2555a9aaa2b3e2c7870097b5babe65d65cffa175dec
$ docker run -d -p 2579:2379 -p 2580:2380 --name etcd2 $IMG \
    $ARGS -name etcd2 -advertise-client-urls $HTTPIP:2579 \
    -initial-advertise-peer-urls $HTTPIP:2580
3089063b6b2ba0868e0f903a3d5b22e617a240cec22ad080dd1b497ddf4736be
$ curl -L $HTTPIP:2579/version
{"etcdserver":"3.2.7","etcdcluster":"3.2.0"}
$ curl -sSL $HTTPIP:2579/v2/members | python -m json.tool | grep etcd
            "name": "etcd0",
            "name": "etcd1",
            "name": "etcd2",
```

Подключенные в настоящее время узлы в кластере

Вы запустили кластер и получили ответ от одного узла. В предыдущих командах все, что относится к «peer», контролирует, как узлы etcd находят друг друга и общаются, а все, что относится к «client», определяет, как другие приложения могут подключаться к etcd.

Давайте посмотрим на распределенную природу etcd в действии.

Листинг 9.9. Тестирование устойчивости кластера etcd

```
$ curl -L $HTTPIP:2579/v2/keys/mykey -XPUT -d value="test key"
{"action":"set","node": >
{"key":"/mykey","value":"test key","modifiedIndex":7,"createdIndex":7}}
$ sleep 5
$ docker kill etcd2
etcd2
$ curl -L $HTTPIP:2579/v2/keys/mykey
curl: (7) couldn't connect to host
$ curl -L $HTTPIP:2379/v2/keys/mykey
{"action":"get","node": >
{"key":"/mykey","value":"test key","modifiedIndex":7,"createdIndex":7}}
```

В предыдущем коде вы добавляете ключ к узлу etcd2, а затем уничтожаете его. Но etcd автоматически реплицировал информацию на другие узлы и в любом случае может предоставить вам эту информацию. Хотя выполнение предыдущего кода приостановилось на пять секунд, обычно etcd будет копироваться менее чем за секунду (даже на разных компьютерах). Не стесняйтесь сейчас выполнить команду `docker start etcd2`, чтобы снова сделать его доступным, – все внесенные вами изменения будут реплицироваться обратно на него.

Видно, что данные все еще доступны, но приходится вручную выбирать другой узел для подключения, что несколько неудобно. К счастью, у etcd есть

решение для этого – вы можете запустить узел в режиме «прокси», что означает, что он не реплицирует никаких данных. Скорее, пересылает запросы на другие узлы.

Листинг 9.10. Использование прокси-сервера etcd

```
$ docker run -d -p 8080:8080 --restart always --name etcd-proxy $IMG \
    etcd -proxy on -listen-client-urls http://0.0.0.0:8080 \
    -initial-cluster $CLUSTER
037c3c3dba04826a76c1d4506c922267885edbfa690e3de6188ac6b6380717ef
$ curl -L $HTTPIP:8080/v2/keys/mykey2 -XPUT -d value="t"
{"action":"set","node": >
{"key":"/mykey2","value":"t","modifiedIndex":12,"createdIndex":12}}
$ docker kill etcd1 etcd2
$ curl -L $HTTPIP:8080/v2/keys/mykey2
{"action":"get","node": >
{"key":"/mykey2","value":"t","modifiedIndex":12,"createdIndex":12}}
```

Это теперь дает вам свободу экспериментировать с тем, как ведет себя etcd, когда более половины узлов находится в автономном режиме.

Листинг 9.11. Использование etcd, когда более чем половина узлов в автономном режиме

```
$ curl -L $HTTPIP:8080/v2/keys/mykey3 -XPUT -d value="t"
{"errorCode":300,"message":"Raft Internal Error", >
"cause":"etcdserver: request timed out","index":0}
$ docker start etcd2
etcd2
$ curl -L $HTTPIP:8080/v2/keys/mykey3 -XPUT -d value="t"
{"action":"set","node": >
{"key":"/mykey3","value":"t","modifiedIndex":16,"createdIndex":16}}
```

Etcd разрешает чтение, но запрещает запись, когда половина или более узлов недоступны.

Теперь вы видите, что можно было бы запускать прокси-сервер etcd на каждом узле кластера, чтобы он действовал как контейнер-посол для получения централизованной конфигурации.

Листинг 9.12. Использование прокси-сервера etcd внутри контейнера-посла

```
$ docker run -it --rm --link etcd-proxy:etcd ubuntu:14.04.2 bash
root@8df11eaae71e:/# apt-get install -y wget
root@8df11eaae71e:/# wget -q -O- http://etcd:8080/v2/keys/mykey3
```

```
{"action":"get","node": >
{"key":"/mykey3","value":"t","modifiedIndex":16,"createdIndex":16}}
```

> **ПОДСКАЗКА.** Посол – это так называемый шаблон Docker, обладающий популярностью среди пользователей Docker. Он помещается между вашим приложением и внешним сервисом и обрабатывает запрос. Он похож на прокси-сервер, но в него встроен некоторый интеллект, чтобы справляться с конкретными требованиями ситуации, – во многом как настоящий посол.

Когда у вас есть etcd, работающий во всех средах, создание компьютера в среде – это всего лишь вопрос его запуска со ссылкой на контейнер etcd-proxy – все сборки непрерывной доставки на компьютере будут использовать правильную конфигурацию среды. Следующий метод показывает, как использовать предоставленную etcd конфигурацию для обновления без простоев.

ОБСУЖДЕНИЕ

Контейнер-посол, показанный в предыдущем разделе, использует флаг ссылки, с которым мы познакомились в методе 8. Как уже отмечалось, в мире Docker связывание несколько утратило популярность, и теперь более своеобразный способ добиться того же – это использовать именованные контейнеры в виртуальной сети, о которых рассказывается в методе 80.

Наличие кластера серверов типа «ключ/значение», обеспечивающих согласованное представление о мире, – это большой шаг вперед по сравнению с управлением файлами конфигурации на множестве компьютеров, и это помогает вам продвигаться к выполнению контракта Docker, описанного в методе 70.

9.4. Обновление запущенных контейнеров

Чтобы каждый день достигать идеала множественного развертывания для эксплуатации в производственной среде, важно сократить время простоя на последнем этапе процесса развертывания – выключения старых приложений и запуске новых. Нет смысла выполнять развертывание четыре раза в день, если каждый раз переключение выполняется в течение часа!

Поскольку контейнеры обеспечивают изолированную среду, ряд проблем уже сглажен. Например, вам не нужно беспокоиться о двух версиях приложения, использующих один и тот же рабочий каталог и конфликтующих друг с другом, или о перечитывании некоторых файлов конфигурации и получении новых значений без перезапуска, используя новый код.

К сожалению, в этом есть некоторые недостатки – изменение файлов на месте уже не просто, поэтому программного перезапуска (необходимого для получения изменений в файле конфигурации) труднее достичь. В результате мы обнаружили, что рекомендуется всегда выполнять один и тот же процесс обновления независимо от того, изменяете ли вы несколько файлов конфигурации или тысячи строк кода.

Давайте рассмотрим процесс обновления, который позволит достичь золотого стандарта развертывания без простоев для веб-приложений.

МЕТОД 75 Использование confd для включения переключения без простоя

Поскольку на хосте контейнеры могут существовать бок о бок, простой подход к удалению контейнера и запуску нового может быть выполнен всего за несколько секунд (и дает возможность для столь же быстрого отката).

Для большинства приложений это будет достаточно быстро, но для приложений с длительным временем запуска или требованиями высокой доступности нужен альтернативный подход. Иногда это неизбежно сложный процесс, требующий специальной обработки с самим приложением, но у веб-приложений есть опция, которую вы, возможно, захотите рассмотреть в первую очередь.

ПРОБЛЕМА

Вы должны иметь возможность обновлять веб-приложения с нулевым временем простоя.

РЕШЕНИЕ

Используйте confd с nginx на вашем хосте, чтобы выполнить двухэтапное переключение.

Nginx – чрезвычайно популярный веб-сервер с критически важной встроенной возможностью – он может перезагружать файлы конфигурации, не прерывая соединения с сервером. В сочетании с confd, инструментом, который может извлекать информацию из центрального хранилища данных (например, etcd) и соответствующим образом изменять конфигурационные файлы, вы можете обновить etcd с последними настройками и наблюдать, как все остальное будет обработано за вас.

> **ПРИМЕЧАНИЕ.** HTTP-сервер Apache и HAProxy также предлагают перезагрузку без простоев и могут использоваться вместо nginx, если у вас есть опыт конфигурирования.

Первым шагом является запуск приложения. Оно будет служить старым приложением, которое вы в конечном итоге обновите. Python поставляется с Ubuntu и имеет встроенный веб-сервер, поэтому мы будем использовать его в качестве примера.

Листинг 9.13. Запуск простого файлового сервера в контейнере

```
$ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$'
    inet 10.194.12.221/20 brd 10.194.15.255 scope global eth0
$ HTTPIP=http://10.194.12.221
```

```
$ docker run -d --name py1 -p 80 ubuntu:14.04.2 \
  sh -c 'cd / && python3 -m http.server 80'
e6b769ec3efa563a959ce771164de8337140d910de67e1df54d4960fdff74544
$ docker inspect -f '{{.NetworkSettings.Ports}}' py1
map[80/tcp:[{0.0.0.0 32768}]]
$ curl -s localhost:32768 | tail | head -n 5
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
```

HTTP-сервер успешно запущен, и мы использовали параметр фильтра команды `inspect`, чтобы извлечь информацию о том, какой порт на хосте перенаправляется с указанием внутри контейнера.

Теперь убедитесь, запущен ли etcd, – этот метод предполагает, что вы все еще находитесь в той же рабочей среде, что и в предыдущем методе. На этот раз используйте etcdctl (сокращение от etcd controller) для взаимодействия с etcd (вместо обращения к etcd с помощью утилиты curl напрямую) для простоты.

Листинг 9.14. Скачайте и используйте образ etcdctl

```
$ IMG=dockerinpractice/etcdctl
$ docker pull dockerinpractice/etcdctl
[...]
$ alias etcdctl="docker run --rm $IMG -C \"$HTTPIP:8080\""
$ etcdctl set /test value
value
$ etcdctl ls
/test
```

Мы скачали образ etcdctl, который подготовили, и настроили псевдоним, чтобы всегда подключать кластер etcd, настроенный ранее. Теперь запустите nginx.

Листинг 9.15. Запустите контейнер nginx + confd

```
$ IMG=dockerinpractice/confd-nginx
$ docker pull $IMG
[...]
$ docker run -d --name nginx -p 8000:80 $IMG $HTTPIP:8080
ebdf3faa1979f729327fa3e00d2c8158b35a49acdc4f764f0492032fa5241b29
```

Это образ, который мы подготовили ранее. Он использует confd для получения информации из etcd и автоматического обновления файлов

конфигурации. Параметр, который мы передаем, сообщает контейнеру, где он может подключиться к кластеру etcd. К сожалению, мы еще не сообщили ему, где можно найти наши приложения, поэтому журналы заполнены ошибками.

Давайте добавим соответствующую информацию в etcd.

Листинг 9.16. Демонстрация автоконфигурации контейнера nginx

```
$ docker logs nginx
Using http://10.194.12.221:8080 as backend
2015-05-18T13:09:56Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:06Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
$ echo $HTTPIP
http://10.194.12.221
$ etcdctl set /app/upstream/py110.194.12.221:32768
10.194.12.221:32768
$ sleep 10
$ docker logs nginx
Using http://10.194.12.221:8080 as backend
2015-05-18T13:09:56Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:06Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:16Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:26Z ebdf3faa1979 confd[14]: >
INFO Target config /etc/nginx/conf.d/app.conf out of sync
2015-05-18T13:10:26Z ebdf3faa1979 confd[14]: >
INFO Target config /etc/nginx/conf.d/app.conf has been updated
$ curl -s localhost:8000 | tail | head -n5
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
```

Обновление etcd было прочитано confd и применено к файлу конфигурации nginx, что позволяет вам посетить простой файловый сервер. Команда `sleep` входит в состав, потому что confd был настроен на проверку обновлений каждые 10 секунд. За кулисами демон confd, работающий в контейнере confd-nginx, проводит опрос на предмет изменений в кластере etcd, используя шаблон внутри контейнера для регенерации конфигурации nginx только при обнаружении изменений.

Допустим, мы решили, что хотим обслуживать /etc, а не /. Теперь запустим наше второе приложение и добавим его в etcd. Поскольку у нас тогда будет два бэкэнда, мы получим ответы от каждого из них.

Листинг 9.17. Использование confd для настройки серверных веб-служб обоих бэкэндов для nginx

```
$ docker run -d --name py2 -p 80 ubuntu:14.04.2 \
  sh -c 'cd /etc && python3 -m http.server 80'
9b5355b9b188427abaf367a51a88c1afa2186e6179ab46830715a20eacc33660
$ docker inspect -f '{{.NetworkSettings.Ports}}' py2
map[80/tcp:[{0.0.0.0 32769}]]
$ curl -s $HTTPIP:32769 | tail | head -n 5
<li><a href="udev/">udev/</a></li>
<li><a href="update-motd.d/">update-motd.d/</a></li>
<li><a href="upstart-xsessions">upstart-xsessions</a></li>
<li><a href="vim/">vim/</a></li>
<li><a href="vtrgb">vtrgb@</a></li>
$ echo $HTTPIP
http://10.194.12.221
$ etcdctl set /app/upstream/py2 10.194.12.221:32769
10.194.12.221:32769
$ etcdctl ls /app/upstream
/app/upstream/py1
/app/upstream/py2
$ curl -s localhost:8000 | tail | head -n 5
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
$ curl -s localhost:8000 | tail | head -n 5
<li><a href="udev/">udev/</a></li>
<li><a href="update-motd.d/">update-motd.d/</a></li>
<li><a href="upstart-xsessions">upstart-xsessions</a></li>
<li><a href="vim/">vim/</a></li>
<li><a href="vtrgb">vtrgb@</a></li>
```

В предыдущем процессе мы проверяли, правильно ли появился новый контейнер, прежде чем добавить его в etcd (см. рис. 9.4). Мы могли бы выполнить этот процесс за один шаг, переписав ключ `/app/upstream/py1` в etcd – это также полезно, если вам нужно, чтобы одновременно был доступен только один бэкэнд.

При двухэтапном переключении второй этап заключается в удалении старого бэкэнда и контейнера.

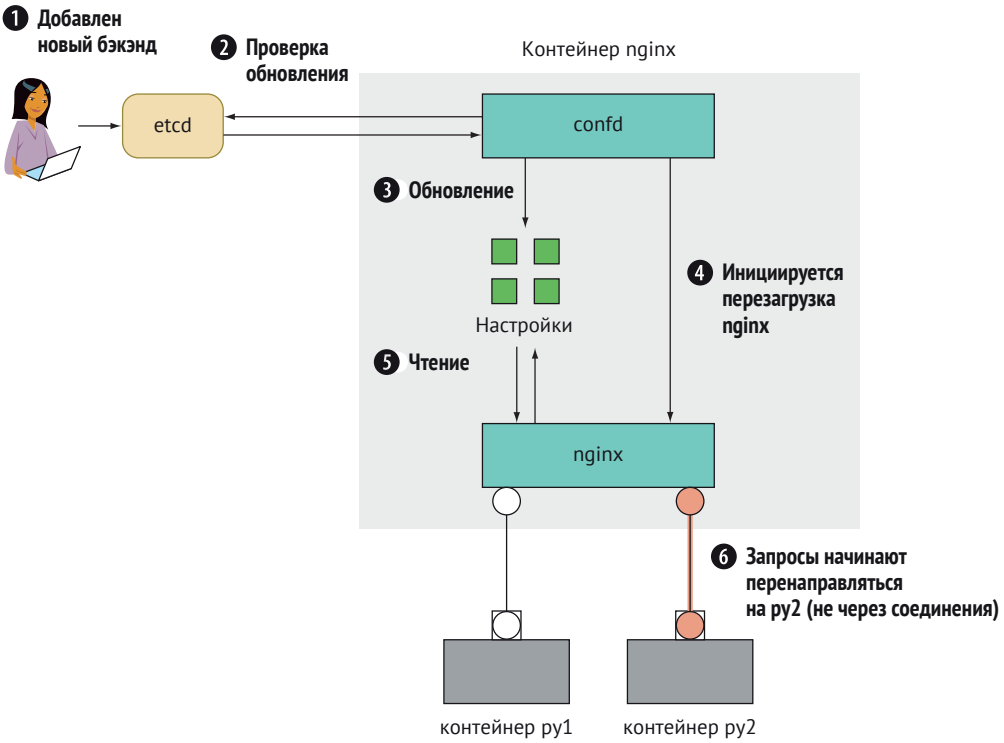


Рис. 9.4 ❖ Добавление контейнера py2 в etcd

Листинг 9.18. Удаление старого адреса

```
$ etcdctl rm /app/upstream/py1
PrevNode.Value: 192.168.1.123:32768
$ etcdctl ls /app/upstream
/app/upstream/py2
$ docker rm -f py1
py1
```

Новое приложение работает само по себе! Никоим образом приложение не было недоступно для пользователей, и не было необходимости вручную подключаться к компьютерам веб-сервера для перезагрузки nginx.

ОБСУЖДЕНИЕ

Использование confd распространяется не только на настройку веб-серверов: если у вас есть какой-либо файл, содержащий текст, который необходимо обновить на основе внешних значений, в действие вступает confd – полезное связующее звено между файлами конфигурации, хранящимися на диске, и кластером etcd с единым источником достоверных данных.

Как было отмечено в предыдущем методе, etcd не предназначен для хранения больших значений.

Также нет причины, по которой вы должны использовать etcd с confd – для самых популярных хранилищ типа «ключ/значение» доступен ряд интеграций, поэтому вам может не понадобиться добавлять другую движущуюся часть, если уже есть что-то, что вам подходит.

Позже, в методе 86, когда мы будем рассматривать использование Docker при эксплуатации, вы увидите метод, который вообще избегает необходимости вручную изменять etcd, если вы хотите обновить backend-серверы для службы.

Резюме

- Docker является отличной основой для заключения контракта между командами разработчиков и операторов.
- Перемещение образов между реестрами может быть хорошим способом контроля за ходом сборки через конвейер непрерывной доставки.
- Bup хорошо сжимает образы и передает даже больше, чем слои.
- Образы Docker можно перемещать и использовать совместно в виде TAR-файлов.
- etcd может действовать как центральное хранилище конфигурации для среды.
- Развертывание без простоя может быть достигнуто путем объединения etcd, confd и nginx.

Глава 10

Сетевое моделирование: Безболезненное реалистичное тестирование среды

О чем рассказывается в этой главе:

- вплотную беремся за Docker Compose;
- тестирование приложений в проблемных сетях;
- первый взгляд на сетевые драйверы Docker;
- создание физической сети для бесперебойного обмена данными между хостами Docker.

В рамках рабочего процесса DevOps вам, вероятно, потребуется каким-то образом использовать сеть. Независимо от того, пытаетесь ли вы выяснить, где находится локальный контейнер memcache, подключаетесь к внешнему миру или подключаете вместе контейнеры Docker, работающие на разных хостах, вы, скорее всего, рано или поздно захотите обратиться к более широкой сети.

После прочтения этой главы вы узнаете, как управлять контейнерами как единым целым с помощью Docker Compose, а также моделировать и управлять сетями с помощью инструментов виртуальной сети Docker.

10.1. Обмен данными между контейнерами: за пределами ручного соединения

В методе 8 вы видели, как связывать контейнеры между собой с помощью соединений, и мы упомянули преимущества, обеспечиваемые четкой формулировкой зависимостей контейнера. К сожалению, у соединений есть

ряд недостатков. Соединения должны быть указаны вручную при запуске каждого контейнера, контейнеры запущены в правильном порядке, циклы в соединениях запрещены, и нет никакой возможности заменить соединение (если контейнер умирает, каждый зависимый контейнер должен быть перезапущен, чтобы воссоздать соединения). Помимо всего прочего, они устарели!

В настоящее время Docker Compose является самой популярной заменой всему тому, что ранее требовало сложной настройки соединений, и сейчас мы на него взглянем.

МЕТОД 76 Простой кластер Docker Compose

Docker Compose начинал свое существование как *Fig*. Это ныне устаревшая независимая попытка облегчить задачу запуска нескольких контейнеров с соответствующими аргументами для соединений, томов и портов. Docker Inc. это так понравилось, что они приобрели его, переделали и выпустили под новым именем.

Этот метод знакомит вас с Docker Compose, используя простой пример оркестровки контейнера Docker.

ПРОБЛЕМА

Вы хотите координировать связанные контейнеры на вашем хост-компьютере.

РЕШЕНИЕ

Используйте Docker Compose, инструмент для определения и запуска многоконтейнерных приложений Docker.

Основная идея заключается в том, что вместо соединения команд запуска контейнера со сложными сценариями оболочки или файлами Makefile вы объявляете конфигурацию запуска приложения, а затем запускаете приложение с помощью одной простой команды.

> **ПРИМЕЧАНИЕ.** Мы предполагаем, что у вас установлен Docker Compose – обратитесь к официальным инструкциям (http://docs.docker.com/compose/install) для получения последних рекомендаций.

В этом методе мы максимально упростим работу с эхо-сервером и клиентом. Клиент отправляет знакомое сообщение «Hello, World!» каждые пять секунд на эхо-сервер, а затем получает сообщение обратно.

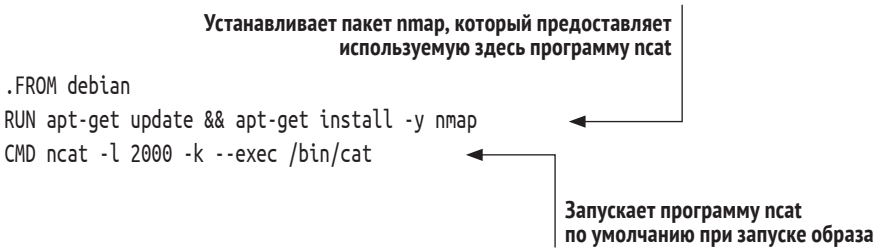
> **ПОДСКАЗКА.** Исходный код этого метода доступен на странице https://github.com/docker-in-practice/docker-compose-echo.

Приведенные ниже команды используются для создания каталога, в котором вы будете работать при создании образа сервера:

```
$ mkdir server
$ cd server
```

Создайте Dockerfile сервера с кодом, показанным в следующем листинге.

Листинг 10.1. Dockerfile – простой эхо-сервер



Аргументы `-l 2000` инструктируют `ncat` прослушивать порт 2000, а `-k` сообщает ему одновременно принимать несколько клиентских подключений и продолжать работу, после того как подключения будут закрыты, чтобы больше клиентов могло подключиться. Последние аргументы, `--exec/bin/cat`, заставят `ncat` запустить `/bin/cat` для всех входящих подключений и перенаправить все данные, поступающие через подключение, в работающую программу.

Затем создайте файл Dockerfile с помощью этой команды:

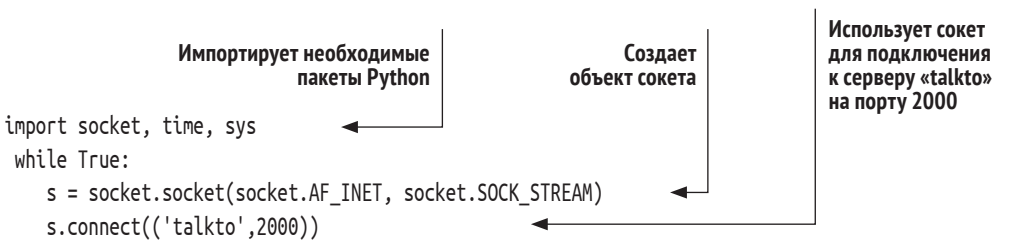
```
$ docker build -t server .
```

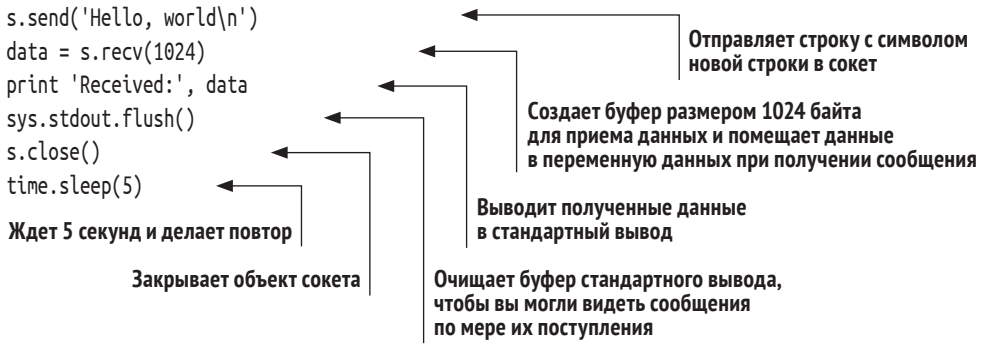
Теперь можете настроить образ клиента, который отправляет сообщения на сервер. Создайте новый каталог и поместите туда файл client.py и Dockerfile:

```
$ cd ..
$ mkdir client
$ cd client
```

В следующем листинге мы будем использовать простую программу, написанную на Python в качестве клиента эхо-сервера.

Листинг 10.2. client.py – простой эхо-клиент





Dockerfile для клиента прост. Он устанавливает Python, добавляет файл client.py и затем по умолчанию запускает его при запуске, как показано в следующем листинге.

Листинг 10.3. Простой эхо-клиент

```
FROM debian
RUN apt-get update && apt-get install -y python
ADD client.py /client.py
CMD ["/usr/bin/python","/client.py"]
```

Создайте клиента с помощью этой команды:

```
docker build -t client .
```

Чтобы продемонстрировать ценность Docker Compose, мы сначала запустим эти контейнеры вручную:

```
docker run --name echo-server -d server
docker run --name client --link echo-server:talkto client
```

Когда вы закончите, нажмите сочетание клавиш **Ctrl-C** на клиенте и удалите контейнеры:

```
docker rm -f client echo-server
```

Многое может пойти не так даже в этом тривиальном примере: первый запуск клиента приведет к невозможности запуска приложения; если вы забудете удалить контейнеры, возникнут проблемы при попытке перезагрузки; а неправильное присвоение имен контейнерам вызовет сбой. Подобные проблемы с оркестровкой будут только увеличиваться, по мере того как ваши контейнеры и их архитектура будет становиться более сложной.

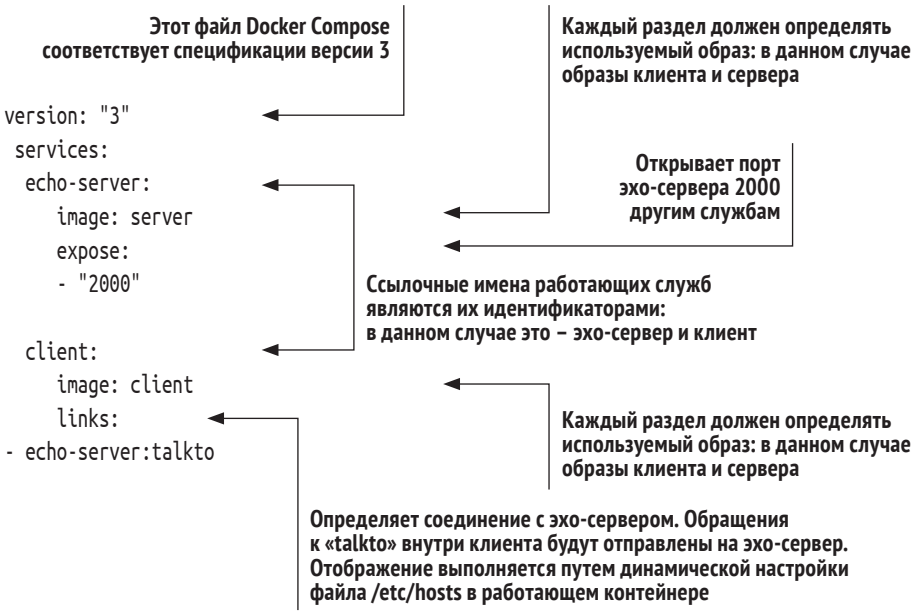
Compose помогает в этом, инкапсулируя оркестровку запуска и конфигурации этих контейнеров в простом текстовом файле, управляя практическими основами команд запуска и выключения.

Compose использует файл в формате YAML. Создайте его в новом каталоге:

```
cd ..
mkdir docker-compose
cd docker-compose
```

Содержимое файла YAML показано в следующем листинге.

Листинг 10.4. docker-compose.yml – эхо-сервер Docker Compose и клиентский YAML-файл



Синтаксис файла docker-compose.yml довольно прост для понимания: каждая служба именуется ключом `services`, а конфигурация указана в разделе с отступом внизу. Каждый элемент конфигурации имеет двоеточие после своего имени, а атрибуты этих элементов указываются либо в той же строке, либо в следующих, начиная с черточек с одинаковым уровнем отступа.

Ключевой элемент конфигурации для понимания здесь – это `links` в определении клиента. Он создается так же, как команда `docker run` устанавливает ссылки, за исключением того, что Compose обрабатывает порядок запуска за вас. Фактически большинство аргументов командной строки Docker имеют прямые аналоги в синтаксисе docker-compose.yml.

Мы использовали оператор `image:` в этом примере, чтобы определить образ, применяемый для каждой службы, но вы также можете получить docker-compose, чтобы динамически выполнить повторную сборку требуемого образа, определив путь к Dockerfile в операторе `build:`. Docker Compose выполнит эту сборку за вас.

> **ПОДСКАЗКА.** Файл YAML – это текстовый файл конфигурации с простым синтаксисом. Вы можете прочитать больше о нем на странице http://yaml.org.

Теперь, когда вся инфраструктура настроена, запустить приложение легко:

```
$ docker-compose up
Creating dockercompose_server_1...
Creating dockercompose_client_1...
Attaching to dockercompose_server_1, dockercompose_client_1
client_1 | Received: Hello, world
client_1 |
client_1 | Received: Hello, world
client_1 |
```

> **ПОДСКАЗКА.** Если при запуске docker-compose выдается сообщение об ошибке («Couldn't connect to Docker daemon at http+unix://var/run/docker.sock—is it running?»), проблема может быть в том, что вам нужно выполнять запуск с помощью sudo.

Когда вы увидите достаточно, нажмите сочетание клавиш **Ctrl-C** несколько раз, чтобы выйти из приложения. Можете вызвать его снова по желанию с помощью той же команды, не беспокоясь об удалении контейнеров. Обратите внимание, что при повторном запуске он выведет «Recreating», а не «Creating».

ОБСУЖДЕНИЕ

Мы упомянули о возможной необходимости использования команды `sudo` в предыдущем разделе – возможно, вы захотите снова просмотреть метод 41, если она применим к вашей ситуации, поскольку она значительно упрощает использование инструментов, взаимодействующих с демоном Docker.

Docker Inc. объявляет о том, что Docker Compose готов к использованию в эксплуатации либо на одном компьютере, как показано здесь, либо на нескольких машинах в режиме swarm, – вы увидите, как это сделать, в методе 87.

Теперь, когда вы освоили Docker Compose, мы перейдем к более сложному и реальному сценарию для docker-compose: использование socat, томов и замена ссылок для добавления серверной функциональности в экземпляр SQLite, работающий на хост-компьютере.

МЕТОД 77 SQLite-сервер, использующий Docker Compose

SQLite по умолчанию не имеет какой-либо концепции TCP-сервера. Основываясь на предыдущих методах, этот метод предоставляет вам средства для достижения функциональности TCP-сервера с помощью Docker Compose.

В частности, он основан на ранее рассмотренных инструментах и концепциях:

- тома;
- проксирование с помощью socat;
- Docker Compose.

Мы также познакомим вас с заменой соединениям – сетями.

> **ПРИМЕЧАНИЕ.** Для этого метода на вашем хосте должна быть установлена SQLite версии 3. Мы также рекомендуем установить rlwrap, чтобы сделать редактирование строк более удобным при взаимодействии с вашим сервером SQLite (это необязательно). Эти пакеты свободно доступны из стандартных менеджеров пакетов.

Код для этого метода доступен для скачивания на странице https://github.com/docker-in-practice/docker-compose-sqlite.

ПРОБЛЕМА

Вы хотите эффективно разработать сложное приложение, обращающееся к внешним данным на вашем хосте, используя Docker.

РЕШЕНИЕ

Используйте Docker Compose.

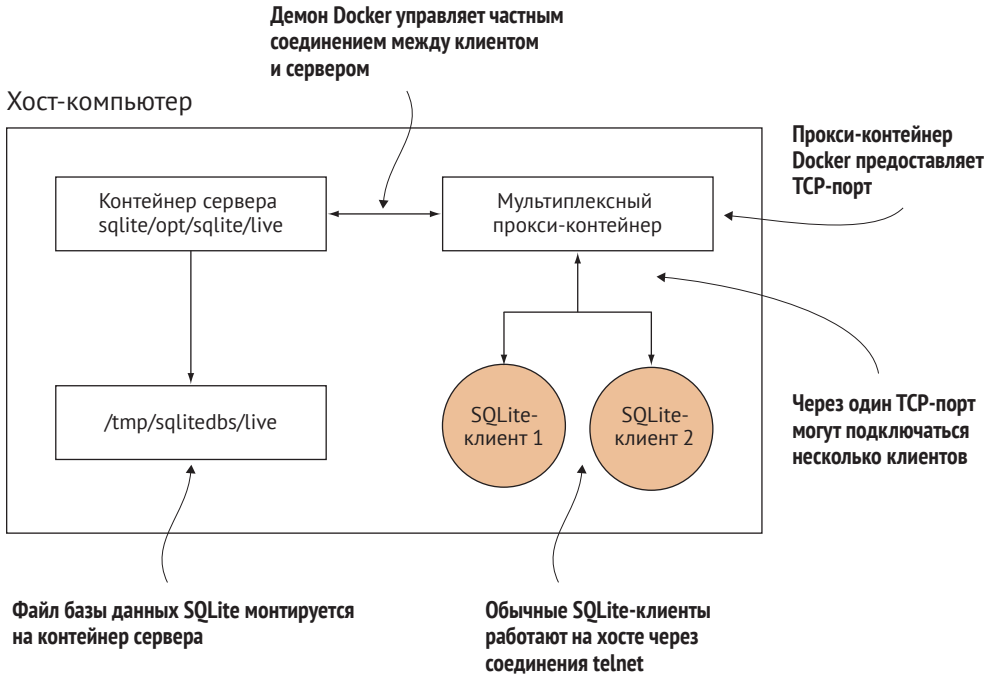


Рис. 10.1 ❖ Как работает сервер SQLite

На рис. 10.1 представлен обзор архитектуры этого метода. На высоком уровне работают два контейнера Docker: один отвечает за выполнение клиентов SQLite, а второй – за проксирование отдельных TCP-подключений к этим клиентам. Обратите внимание, что контейнер, выполняющий SQLite, не предоставляется хосту – это делает прокси-контейнер. Такое разделение ответственности на отдельные единицы является общей чертой архитектуры микросервисов.

Мы будем использовать один и тот же образ для всех наших узлов. Настройте Dockerfile в следующем листинге.

Листинг 10.5. Многофункциональный сервер SQLite, клиент и файл Dockerfile прокси-сервера

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y rlwrap sqlite3 socat
 EXPOSE 12345
```

Устанавливает необходимые приложения

Открывает порт 12345, чтобы узлы могли обмениваться данными через демон Docker

В следующем листинге показан файл docker-compose.yml, который определяет способ запуска контейнеров.

Листинг 10.6. Сервер SQLite и прокси-докер-compose.yml

Серверный и прокси-контейнер определены в этом разделе

Создает прокси-сервер socat, чтобы соединить вывод вызова SQLite с TCP-портом

Собирает образ при запуске из файла Dockerfile в том же каталоге

```
version: "3"
services:
  server:
     command: socat TCP-L:12345,fork,reuseaddr >
EXEC:'sqlite3 /opt/sqlite/db',pty
     build: .
     volumes:
     - /tmp/sqlitedbs/test:/opt/sqlite/db
     networks:
     - sqlnet

  proxy:
     command: socat TCP-L:12346,fork,reuseaddr TCP:server:12345
     build: .
     ports:
     - 12346:12346
```

Монтирует тестовый файл базы данных SQLite в /opt/sqlite/db внутри контейнера

Обе службы станут частью сети sqlnet

Серверный и прокси-контейнер определены в этом разделе

Создает прокси-сервер socat для передачи данных из порта 12346 в порт 12345 контейнера сервера

Публикует порт 12346 на хост

```
    networks:
    - sqlnet
networks:
    sqlnet:
    driver: bridge
```

Обе службы станут частью сети sqlnet

Определяет список сетей, к которым могут присоединиться контейнеры в этом файле Docker Compose

Процесс socat в контейнере сервера будет прослушивать порт 12345 и разрешать несколько соединений, как указано в аргументе `TCP-L: 12345, fork, reuseaddr`. Часть, следующая за `EXEC:`, сообщает `socat` запускать SQLite в файле /opt/sqlite/db для каждого подключения, присваивая псевдотерминал процессу. Процесс socat в клиентском контейнере имеет то же поведение при прослушивании, что и контейнер сервера (за исключением другого порта), но вместо запуска чего-либо в ответ на входящее соединение он установит TCP-соединение с сервером SQLite.

Одним заметным отличием от предыдущего метода является использование сетей, а не соединений – сети представляют собой способ создания новых виртуальных сетей внутри Docker.

Docker Compose всегда будет использовать новую «мостовую» виртуальную сеть по умолчанию; она только что была явно указана в предыдущей конфигурации Compose. Поскольку любая новая мостовая сеть разрешает доступ к контейнерам с использованием имен их служб, нет необходимости применять соединения (хотя вы все равно можете это сделать, если хотите использовать псевдонимы для служб).

Хотя эта функциональность может быть достигнута в одном контейнере, настройка сервер-/прокси-контейнера позволяет архитектуре этой системы расти легче, поскольку каждый контейнер отвечает за одну работу. Сервер отвечает за открытие соединений SQLite, а прокси-сервер отвечает за предоставление службы хост-компьютеру.

В следующем листинге (упрощенный вариант оригинала в репозитории, https://github.com/docker-in-practice/docker-compose-sqlite) на вашем хост-компьютере создаются две минимальные базы данных SQLite: тестовая и настоящая.

Листинг 10.7. setup_dbs.sh

```
#!/bin/bash
echo "Creating directory"
SQLITEDIR=/tmp/sqlitedbs
rm -rf $SQLITEDIR
 if [ -a $SQLITEDIR ]
 then
    echo "Failed to remove $SQLITEDIR"
```

Удаляет любой каталог из предыдущего запуска

Выдает ошибку, если каталог все еще существует

```
    exit 1
fi
mkdir -p $SQLITEDIR
cd $SQLITEDIR
echo "Creating DBs"
echo 'create table t1(c1 text);' | sqlite3 test
echo 'create table t1(c1 text);' | sqlite3 live
echo "Inserting data"
echo 'insert into t1 values ("test");' | sqlite3 test
echo 'insert into t1 values ("live");' | sqlite3 live
cd - > /dev/null 2>&1
echo "All done OK"
```

Создает тестовую БД с одной таблицей

Создает реальную БД с одной таблицей

Вставляет одну строку со строкой «test» в таблицу

Вставляет одну строку со строкой «live» в таблицу

Возвращается в предыдущий каталог

Чтобы запустить этот пример, настройте базы данных и вызовите docker-compose, как показано в следующем листинге.

Листинг 10.8. Запуск кластера Docker Compose

```
$ chmod +x setup_dbs.sh
$ ./setup_dbs.sh
$ docker-compose up
Creating network "tmpnwxqlnjvdn_sqlnet" with driver "bridge"
Building proxy
Step 1/3 : FROM ubuntu:14.04
14.04: Pulling from library/ubuntu
[...]
Successfully built bb347070723c
Successfully tagged tmpnwxqlnjvdn_proxy:latest
[...]
Successfully tagged tmpnwxqlnjvdn_server:latest
[...]
Creating tmpnwxqlnjvdn_server_1
Creating tmpnwxqlnjvdn_proxy_1 ... done
Attaching to tmpnwxqlnjvdn_server_1, tmpnwxqlnjvdn_proxy_1
```

Затем в одном или нескольких других терминалах вы можете запустить Telnet для создания множества сеансов с одной базой данных SQLite.

Листинг 10.9. Подключение к серверу SQLite

Создает соединение с прокси-сервером с помощью Telnet, обернутое в rlwrap, чтобы получить функции редактирования и истории командной строки

```
$ rlwrap telnet localhost 12346
```

```
 Trying 127.0.0.1...
 Connected to localhost.
 Escape character is '^]'.
 SQLite version 3.7.17
 Enter ".help" for instructions
sqlite> select * from t1;
 select * from t1;
test
sqlite>
```

Вывод соединения Telnet

Соединяется с SQLite здесь

Выполняет команду SQL для приглашения sqlite

Если вы хотите переключить сервер в рабочее состояние, можете исправить конфигурацию, изменив строку `volumes` в файле docker-compose.yml с

```
- /tmp/sqlitedbs/test:/opt/sqlite/db
```

на:

```
- /tmp/sqlitedbs/live:/opt/sqlite/db
```

Затем повторно выполните команду:

```
$ docker-compose up
```

> **ПРЕДОСТЕРЕЖЕНИЕ.** Хотя мы провели несколько базовых тестов с мультиплексированием SQLite-клиентов, мы не даем никаких гарантий относительно целостности данных или производительности этого сервера при любом виде нагрузки. SQLite-клиент не был задуман для такой работы. Цель этого метода – продемонстрировать общий подход к представлению двоичного файла таким образом.

Данный метод демонстрирует, как Docker Compose может взять что-то относительно хитрое и сложное и сделать его надежным и простым. Здесь мы взяли SQLite и наделили его дополнительной серверной функциональностью, подключив контейнеры к прокси-вызовам SQLite данных на хосте. Управление сложностью контейнера стало значительно проще с помощью конфигурации YAML в Docker Compose, которая превращает сложную задачу правильной организации контейнеров из ручного, подверженного ошибкам процесса в более безопасный, автоматизированный процесс, который может быть поставлен под контроль исходного кода. Это начало нашего путешествия в оркестровку, о которой вы услышите гораздо больше в четвертой части книги.

ОБСУЖДЕНИЕ

Использование сетей с функцией Docker Compose `depends_on` позволяет эффективно эмулировать функциональность соединений, контролируя порядок запуска. Для полной обработки всех возможных вариантов, доступных в Docker Compose, мы рекомендуем вам прочитать официальную документацию по адресу https://docs.docker.com/compose/compose-file/.

Чтобы узнать больше о виртуальных сетях Docker, взгляните на метод 80 – в нем подробно рассказывается о том, что Docker Compose делает за кулисами для настройки ваших виртуальных сетей.

10.2. Использование Docker для симуляции реальной сетевой среды

Большинство людей, которые пользуются интернетом, рассматривают его как черный ящик, который каким-то образом извлекает информацию из других мест по всему миру и размещает ее на своих экранах. Иногда они сталкиваются с медленным соединением или его обрывом, и в результате мы нередко слышим проклятья в адрес интернет-провайдера.

Когда вы создаете образы, содержащие приложения, которые необходимо подключить, вы, скорее всего, лучше понимаете, какие компоненты куда должны подключаться и как выглядит общая установка. Но одно остается неизменным: вы все еще можете сталкиваться с медленным соединением или его обрывом. Даже крупные компании с центрами обработки данных, которыми они владеют и управляют, наблюдали ненадежную сетевую среду и проблемы, которые она вызывает в приложениях.

Мы рассмотрим несколько способов, с помощью которых вы можете поэкспериментировать с нестабильными сетями, чтобы определить, с какими проблемами можно столкнуться в реальном мире.

МЕТОД 78 Имитация проблемных сетей с помощью Comcast

Как бы нам ни хотелось иметь идеальные условия при работе в сети, когда мы распределяем приложения по множеству компьютеров, реальность гораздо страшнее. Это рассказы о потере пакетов, обрыве соединения и сетевых разделах, особенно когда речь идет о поставщиках обычных облачных сервисов.

Целесообразно протестировать свой стек до того, как он столкнется с этими ситуациями в реальном мире, чтобы увидеть, как он себя ведет, – приложение, разработанное для высокой доступности, не должно останавливаться, если внешняя служба начинает испытывать значительную дополнительную задержку.

ПРОБЛЕМА

Вы хотите иметь возможность применять различные условия сети к отдельным контейнерам.

РЕШЕНИЕ

Используйте Comcast (это утилита, а не интернет-провайдер).

Comcast (https://github.com/tylertreat/Comcast) – программа с забавным названием для изменения сетевых интерфейсов на вашем компьютере с Linux,

чтобы применять к ним необычные (или, если вам не повезло, типичные!) условия.

Всякий раз, когда Docker создает контейнер, он также создает интерфейсы виртуальных сетей – так все ваши контейнеры имеют разные IP-адреса и могут проверять связь друг с другом.

Поскольку это стандартные сетевые интерфейсы, вы можете использовать для них Comcast, если можете найти имя сетевого интерфейса. Легче сказать, чем сделать.

В следующем листинге показан образ Docker, содержащий Comcast, все его предпосылки и настройки.

Листинг 10.10. Подготовка к запуску образа comcast

```
$ IMG=dockerinpractice/comcast
$ docker pull $IMG
latest: Pulling from dockerinpractice/comcast
[...]
Status: Downloaded newer image for dockerinpractice/comcast:latest
$ alias comcast="docker run --rm --pid=host --privileged \
-v /var/run/docker.sock:/var/run/docker.sock $IMG"
$ comcast -help
Usage of comcast:
  -cont string
        Container ID or name to get virtual interface of
  -default-bw int
        Default bandwidth limit in kbit/s (fast-lane) (default -1)
  -device string
        Interface (device) to use (defaults to eth0 where applicable)
  -dry-run
        Specifies whether or not to actually commit the rule changes
  -latency int
        Latency to add in ms (default -1)
  -packet-loss string
        Packet loss percentage (e. g. 0.1%)
  -stop
        Stop packet controls
  -target-addr string
        Target addresses, (e. g. 10.0.0.1 or 10.0.0.0/24 or >
10.0.0.1,192.168.0.0/24 or 2001:db8:a::123)
  -target-bw int
        Target bandwidth limit in kbit/s (slow-lane) (default -1)
  -target-port string
        Target port(s) (e. g. 80 or 1:65535 or 22,80,443,1000:1010)
  -target-proto string
```

```
        Target protocol TCP/UDP (e. g. tcp or tcp,udp or icmp) (default >
"tcp,udp,icmp")
  -version
        Print Comcast's version
```

Добавленные здесь твики предоставляют опцию `-cont`, которая позволяет вам обращаться к контейнеру, вместо того чтобы искать имя виртуального интерфейса. Обратите внимание, нам пришлось добавить некоторые специальные флаги в команду `docker run`, чтобы предоставить контейнеру больше полномочий, – так Comcast может свободно проверять и применять изменения к сетевым интерфейсам.

Чтобы увидеть разницу, которую может принести Comcast, для начала выясним, как выглядит обычное сетевое соединение. Откройте новый терминал и выполните следующие команды, чтобы установить свои ожидания относительно базовой производительности сети:

Связь между этим компьютером и www.example.com кажется надежной, без потери пакетов

```
$ docker run -it --name c1 ubuntu:14.04.2 bash
root@0749a2e74a68:/# apt-get update && apt-get install -y wget
[...]
root@0749a2e74a68:/# ping -q -c 5 www.example.com
PING www.example.com (93.184.216.34) 56(84) bytes of data.

--- www.example.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, >
time 4006ms
rtt min/avg/max/mdev = 86.397/86.804/88.229/0.805 ms
root@0749a2e74a68:/# time wget -o /dev/null https://www.example.com

real    0m0.379s
user    0m0.008s
sys     0m0.008s
root@0749a2e74a68:/#
```

Среднее время прохождения сигнала туда и обратно для www.example.com составляет около 100 мс

Общее время, необходимое для скачивания HTML-страницы www.example.com, составляет около 0,7 с

После того как вы это сделали, оставьте контейнер включенным и можете применить к нему условия сети:

```
$ comcast -cont c1 -default-bw 50 -latency 100 -packet-loss 20%
Found interface veth62cc8bf for container 'c1'
sudo tc qdisc show | grep "netem"
sudo tc qdisc add dev veth62cc8bf handle 10: root htb default 1
sudo tc class add dev veth62cc8bf parent 10: classid 10:1 htb rate 50kbit
sudo tc class add dev veth62cc8bf parent 10: classid 10:10 htb rate 1000000kb
➥ it
sudo tc qdisc add dev veth62cc8bf parent 10:10 handle 100: netem delay 100ms
```

```
➥ loss 20.00%
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p tcp
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p udp
sudo iptables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p icmp
sudo ip6tables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p tcp
sudo ip6tables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p udp
sudo ip6tables -A POSTROUTING -t mangle -j CLASSIFY --set-class 10:10 -p icmp
Packet rules setup...
Run `sudo tc -s qdisc` to double check
Run `comcast --device veth62cc8bf --stop` to reset
```

Предыдущая команда применяет три разных условия: ограничение пропускной способности 50 Кбит/с для всех адресатов (чтобы освежить воспоминания о коммутируемом соединении), дополнительная задержка в 100 мс (поверх любой внутренней задержки) и процент потери пакетов, равный 20 %.

Comcast сначала идентифицирует соответствующий виртуальный сетевой интерфейс для контейнера, а затем вызывает ряд стандартных сетевых утилит командной строки Linux для применения правил трафика, перечисляя, что он делает по мере продвижения. Давайте посмотрим, как наш контейнер реагирует на это:

```
root@0749a2e74a68:/# ping -q -c 5 www.example.com
PING www.example.com (93.184.216.34) 56(84) bytes of data.

--- www.example.com ping statistics ---
5 packets transmitted, 2 received, 60% packet loss, time 4001ms
rtt min/avg/max/mdev = 186.425/189.429/195.008/3.509 ms
root@0749a2e74a68:/# time wget -o /dev/null https://www.example.com

real    0m1.993s
user    0m0.011s
sys     0m0.011s
```

Все прошло успешно! Мы видим сообщение о дополнительной задержке в 100 мс, а синхронизация от `wget` показывает замедление чуть больше, чем в 5 раз, примерно, как и ожидалось (ограничение пропускной способности, добавление задержки и потеря пакетов будут влиять на это время). Но есть нечто странное в потере пакетов – кажется, что она в три раза больше, чем ожидалось. Важно иметь в виду, что отправляется несколько пакетов, и что потеря пакетов не является точным счетчиком «один из пяти» – если вы увеличите количество пакетов до 50, то обнаружите, что полученная потеря намного больше ближе к тому, что ожидается.

Обратите внимание, что эти правила применяются ко *всем* сетевым подключениям через этот сетевой интерфейс. Сюда входят подключения к хосту и другим контейнерам.

Давайте теперь поручим Comcast удалить правила. К сожалению, Comcast пока не может добавлять и удалять отдельные условия, поэтому изменение чего-либо в сетевом интерфейсе означает полное удаление и повторное добавление правил в интерфейсе. Вам также необходимо удалить правила, если вы хотите восстановить обычную работу контейнерной сети.

Не беспокойтесь об их устранении, если вы завершаете работу контейнера, – они будут автоматически удалены, когда Docker удалит интерфейс виртуальной сети.

```
$ comcast -cont c1 -stop
Found interface veth62cc8bf for container 'c1'
[...]
Packet rules stopped...
Run `sudo tc -s qdisc` to double check
Run `comcast` to start
```

Если вы хотите перейти к практике, то можете погрузиться в инструменты управления трафиком Linux, возможно, используя Comcast с `-dry-run` для генерации примеров наборов команд. Полное рассмотрение этих возможностей выходит за рамки метода, но помните: если вы помещаете его в контейнер и он обращается к сети, можете поиграть с ним.

ОБСУЖДЕНИЕ

При наличии незначительных усилий по реализации нет никаких причин, по которым вы не можете использовать Comcast для чего-то большего, чем просто ручное управление пропускной способностью контейнера. Например, представьте, что вы используете такой инструмент, как btsync (метод 35), но хотите ограничить доступную пропускную способность, чтобы он не насыщал ваше соединение, – скачайте Comcast, поместите его в контейнер и используйте `ENTRYPOINT` (метод 49) для настройки ограничения пропускной способности как часть запуска контейнера.

Для этого вам нужно установить зависимости Comcast (они перечислены для образа `alpine` в нашем файле Dockerfile по адресу https://github.com/docker-in-practice/docker-comcast/blob/master/Dockerfile) и, вероятно, предоставить контейнеру как минимум возможности сетевого администратора – вы можете прочитать больше о возможностях в методе 93.

МЕТОД 79 Имитация проблемных сетей с помощью Blockade

Comcast – превосходный инструмент для ряда приложений, но есть важный вариант использования, который он не решает, – как применять условия сети к контейнерам в массовом порядке? Запуск Comcast для десятков контейнеров вручную был бы проблемным, а для сотни – немыслимым! Это особенно актуальная проблема для контейнеров, потому что их запуск очень дешев – если вы пытаетесь запустить симуляцию большой сети на одном компьютере

с сотнями виртуальных машин, а не с контейнерами, вы можете столкнуться с большими проблемами, такими как нехватка памяти!

Что касается симуляции сети при большом количестве компьютеров, то при таком масштабе возникает интересный сбой сети – нарушение связности сети, когда группа сетевых компьютеров разделяется на две или более частей, так что все компьютеры в одной и той же части могут общаться друг с другом, но разные части общаться не могут.

Исследования показывают, что это происходит чаще, чем вы думаете, особенно в облаках на уровне конечного пользователя!

Переход на классический маршрут микросервисов Docker сильно облегчает эти проблемы, и наличие инструментов для проведения экспериментов имеет решающее значение для понимания того, как ваша служба справится с этим.

ПРОБЛЕМА

Вы хотите организовать сетевые условия для большого количества контейнеров, включая создание нарушений связности сети.

РЕШЕНИЕ

Используйте Blockade (https://github.com/worstcase/blockade) – часть программного обеспечения с открытым исходным кодом, разработанная командой Dell и созданная для «тестирования сетевых сбоев и нарушений связности».

Blockade работает, читая файл конфигурации (blockade.yml) в вашем текущем каталоге, который определяет, как запускать контейнеры и какие условия применять к ним. Чтобы применить условия, он может скачивать другие образы с установленными необходимыми утилитами. Полная информация о конфигурации доступна в документации Blockade, поэтому мы рассмотрим только самое необходимое.

Для начала вам нужно создать файл blockade.yml.

Листинг 10.11. Файл blockade.yml

```
containers:
  server:
    container_name: server
    image: ubuntu:14.04.2
    command: /bin/sleep infinity

  client1:
    image: ubuntu:14.04.2
    command: sh -c "sleep 5 && ping server"

  client2:
    image: ubuntu:14.04.2
    command: sh -c "sleep 5 && ping server"
```

```
network:
  flaky: 50%
  slow: 100ms
  driver: udn
```

Контейнеры в предыдущей конфигурации настроены для представления сервера, к которому подключаются два клиента. На практике это может быть что-то вроде сервера базы данных с клиентскими приложениями, и нет никакой внутренней причины, по которой вам нужно ограничивать количество компонентов, которые вы хотите моделировать. Скорее всего, если вы можете представить его в файле compose.yml (см. метод 76), вы, вероятно, сможете смоделировать его в Blockade.

Здесь мы указали сетевой драйвер как `udn` – это позволяет Blockade имитировать поведение Docker Compose, как в методе 77, создавая новую виртуальную сеть, чтобы контейнеры могли проверять связь друг с другом по имени контейнера. Для этого нам пришлось явно указать имя контейнера для сервера, так как по умолчанию Blockade генерирует его самостоятельно.

Команды `sleep 5` должны убедиться, что сервер запущен перед запуском клиентов, – если вы предпочитаете использовать соединения с Blockade, они обеспечат запуск контейнеров в правильном порядке. Пока не беспокойтесь о разделе `network`; мы вернемся к нему в ближайшее время.

Как обычно, первым шагом при использовании Blockade является извлечение образа:

```
$ IMG=dockerinpractice/blockade
$ docker pull $IMG
latest: Pulling from dockerinpractice/blockade
[...]
Status: Downloaded newer image for dockerinpractice/blockade:latest
$ alias blockade="docker run --rm -v \$PWD:/blockade \
-v /var/run/docker.sock:/var/run/docker.sock $IMG"
```

Вы наверное заметили, что мы упускаем несколько аргументов для команды `docker run` по сравнению с предыдущим методом (`--privileged` и `--pid = host`). Blockade использует другие контейнеры для выполнения сетевых манипуляций, поэтому ему самому не нужны права. Также обратите внимание на аргумент для монтирования текущего каталога в контейнер, чтобы Blockade имел возможность получить доступ к файлу blockade.yml и сохранить состояние в скрытой папке.

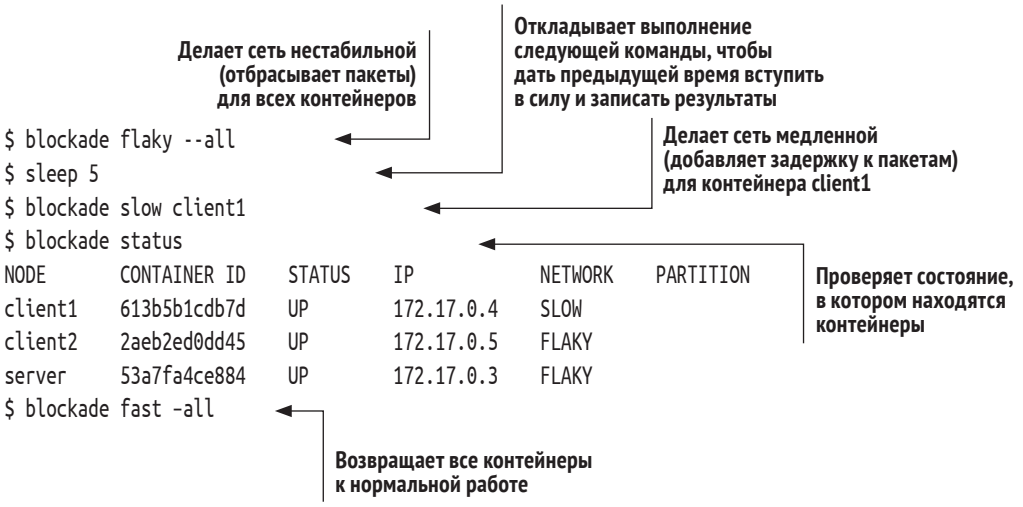
> **ПРИМЕЧАНИЕ.** Если вы работаете в сетевой файловой системе, вы можете столкнуться со странными проблемами, связанными с правами доступа при первом запуске Blockade, – вероятно, потому, что Docker пытается создать скрытую папку состояний в качестве пользователя root, но сетевая файловая система не взаимодействует. Решение состоит в том, чтобы использовать локальный диск.

Наконец-то мы подошли к моменту истины – к запуску Blockade. Убедитесь, что вы находитесь в каталоге, в котором вы сохранили blockade.yml:

```
$ blockade up
NODE        CONTAINER ID    STATUS    IP            NETWORK    PARTITION
client1     613b5b1cdb7d    UP        172.17.0.4    NORMAL
client2     2aeb2ed0dd45    UP        172.17.0.5    NORMAL
server      53a7fa4ce884    UP        172.17.0.3    NORMAL
```

> **ПРИМЕЧАНИЕ.** При запуске Blockade может иногда выдавать загадочные ошибки о файлах в /proc, которых не существует. Первое, что нужно проверить, – сразу ли контейнер завершил работу при запуске, не позволяя Blockade проверить состояние сети. Кроме того, попытайтесь противостоять любому искушению использовать опцию Blockade -c для указания пользовательского пути к файлу конфигурации – внутри контейнера доступны только подкаталоги текущего каталога.

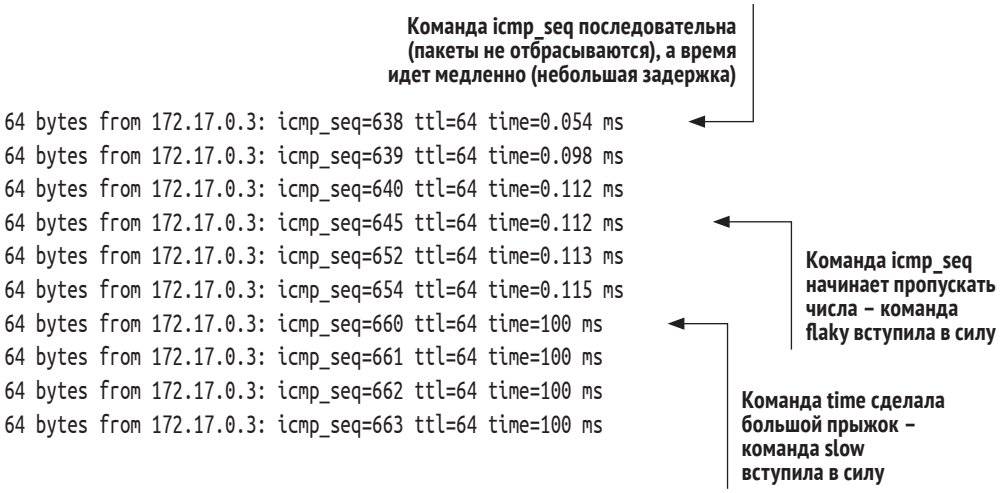
Все контейнеры, определенные в нашем конфигурационном файле, были запущены, и мы получили о них кучу полезной информации. Давайте теперь применим некоторые основные условия сети. Выполните мониторинг файлов журналов client1 в новом терминале (с помощью команды `docker logs -f 613b5b1cdb7d`), чтобы увидеть, что происходит, когда вы выполняете изменения:



Команды `flaky` и `slow` используют значения, определенные в разделе `network` из предыдущего файла конфигурации (листинг 10.11), – нет никакого способа указать ограничение в командной строке. При желании можно отредактировать файл blockade.yml во время работы контейнеров, а затем выборочно применить новые ограничения к контейнерам. Имейте в виду, что контейнер может быть в медленной (slow) *или* нестабильной (flaky) сети, но не в обеих.

Помимо этих ограничений, удобство работы с сотнями контейнеров довольно значительно.

Если вы посмотрите на свои журналы из `client1`, то теперь сможете увидеть, когда различные команды вступили в силу:



Все это полезно, но мы уже ничего не можем сделать с некоторыми (вероятно, болезненными) сценариями поверх Comcast, поэтому давайте взглянем на убойную особенность Blockade – нарушения связности сети:

```
$ blockade partition server client1,client2
$ blockade status
NODE       CONTAINER ID    STATUS    IP            NETWORK    PARTITION
client1    613b5b1cdb7d    UP        172.17.0.4    NORMAL     2
client2    2aeb2ed0dd45    UP        172.17.0.5    NORMAL     2
server     53a7fa4ce884    UP        172.17.0.3    NORMAL     1
```

Благодаря этому наши три узла помещаются в два блока – сервер в один, а клиенты в другой, – никакого обмена данными между ними. Вы увидите, что журнал client1 прекратил делать что-либо из-за потери всех пакетов.

Однако клиенты по-прежнему могут общаться друг с другом, и вы можете проверить это, отправив несколько пакетов:

```
$ docker exec 613b5b1cdb7d ping -qc 3172.17.0.5
PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.

--- 172.17.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2030ms
rtt min/avg/max/mdev = 0.109/0.124/0.150/0.018 ms
```

Потери пакетов нет, низкая задержка … выглядит как хорошее соединение. Разделы и другие сетевые условия работают независимо, поэтому вы можете поэкспериментировать с потерей пакетов, пока ваши приложения разбиты

на разделы. Количество определяемых вами разделов, не ограничено, так что можете играть со сложными сценариями в свое удовольствие.

ОБСУЖДЕНИЕ

Если вам нужно мощности больше, чем могут дать Blockade и Comcast по отдельности, можете объединить их. Blockade отлично подходит для создания разделов и выполнения тяжелых работ по запуску контейнеров; добавление Comcast дает вам детальный контроль над сетевыми подключениями каждого контейнера.

Также стоит посмотреть полную справку по Blockade – она предлагает другие вещи, которые могут оказаться полезными, например функцию «хаоса» для воздействия на случайные контейнеры с различными условиями и аргумент для команд `--random`, чтобы вы могли (например) увидеть, как ваше приложение реагирует на случайное уничтожение контейнеров. Если вы слышали о Chaos Monkey от Netflix, то это способ имитировать его в меньших масштабах.

10.3. Docker и виртуальные сети

Основная функциональность Docker заключается в изоляции. В предыдущих главах были показаны некоторые преимущества изоляции процессов и файловых систем, а здесь вы увидели изоляцию сети.

Можно рассматривать два аспекта изоляции сети:

- *отдельная песочница* – каждый контейнер имеет свой собственный IP-адрес и набор портов для прослушивания, не наступая на пальцы другим контейнерам (или хосту);
- *групповая песочница* – это логическое расширение отдельной песочницы – все изолированные контейнеры сгруппированы в закрытой сети, что позволяет вам экспериментировать, не вмешиваясь в сеть, в которой работает ваш компьютер (и не навлекая на себя гнев сетевого администратора вашей компании!).

Предыдущие методы предоставляют практические примеры этих двух аспектов изоляции сети: Comcast манипулировал отдельными песочницами, чтобы применять правила к каждому контейнеру, тогда как разбиение на разделы в Blockade основывалось на возможности иметь полный надзор за частной контейнерной сетью, чтобы разделить ее на фрагменты. За кулисами это выглядит, как показано на рис. 10.2.

Точные детали того, как работает мост, не важны. Достаточно сказать, что мост создает плоскую сеть между контейнерами (он позволяет прямой обмен данными без промежуточных шагов) и перенаправляет запросы во внешний мир на ваше внешнее соединение.

Затем Docker Inc. изменила эту модель на основе отзывов пользователей, чтобы позволить создавать свои собственные виртуальные сети с помощью *сетевых драйверов*, системы плагинов, расширяющей сетевые возможности Docker. Эти плагины либо встроены, либо предоставлены третьей стороной и должны сделать всю необходимую работу, чтобы подключить сеть, позволяя вам продолжать использовать ее.

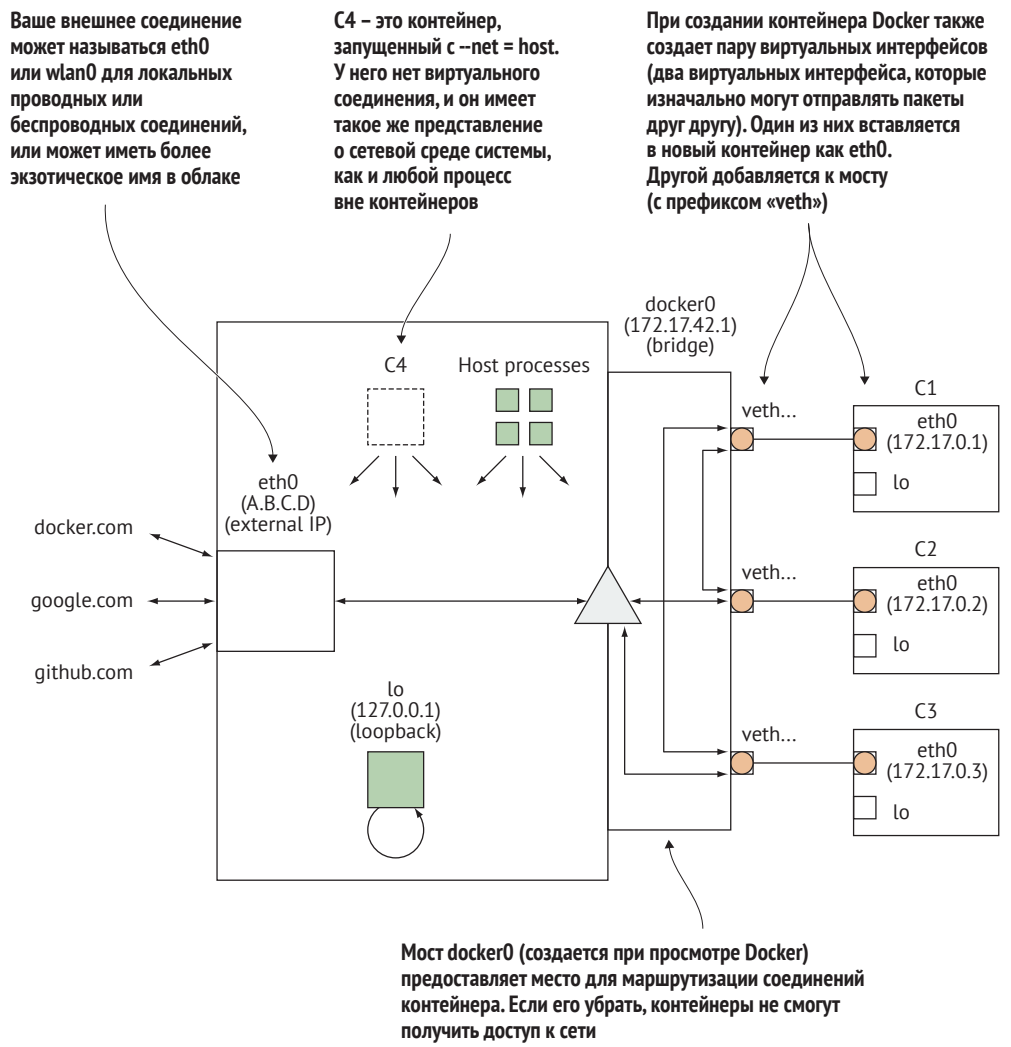


Рис. 10.2 ❖ Внутренняя сетевая среда Docker на хост-компьютере

Новые сети, которые вы создаете, могут рассматриваться как дополнительные групповые песочницы, обычно обеспечивающие доступ в рамках песочницы, но не позволяющие обмен данными между песочницами (хотя точные детали поведения сети зависят от драйвера).

МЕТОД 80 Создание еще одной виртуальной сети Docker

Когда люди впервые узнают о возможности создания собственных виртуальных сетей, одна из распространенных ответных реакций – спросить, как можно создать копию моста Docker по умолчанию, чтобы позволить наборам

контейнеров обмениваться данными, но при этом быть изолированными от других контейнеров.

Docker Inc. поняла, что это будет популярный запрос, поэтому он был реализован как одна из первых функций виртуальных сетей в первоначальном экспериментальном выпуске.

ПРОБЛЕМА

Вам нужно решение, поддерживаемое Docker Inc. для создания виртуальных сетей.

РЕШЕНИЕ

Используйте набор подкоманд Docker, вложенных в `docker network`, чтобы создать собственную виртуальную сеть.

Встроенный драйвер «моста», вероятно, является наиболее часто используемым драйвером – он официально поддерживается и позволяет создавать свежие копии встроенного моста по умолчанию.

Однако есть одно важное отличие, которое мы рассмотрим позже в этом методе, – в мостах не по умолчанию вы можете пинговать контейнеры по имени.

Можно посмотреть список встроенных сетей с помощью команды `docker network ls`:

```
$docker network ls
NETWORK ID          NAME          DRIVER        SCOPE
100ce06cd9a8        bridge        bridge        local
d53919a3bfa1        host          host          local
2d7fcd86306c        none          null          local
```

Здесь вы видите три сети, которые всегда доступны для подключения контейнеров на моем компьютере. Сеть `bridge` – это место, где контейнеры заканчиваются по умолчанию с возможностью общаться с другими контейнерами на мосту. Сеть `host` указывает, что происходит, когда вы используете `--net = host` при запуске контейнера (он видит сеть так же, как любая обычная программа, работающая на вашем компьютере), а `none` соответствует `--net = none`, контейнеру, у которого есть только интерфейс loopback.

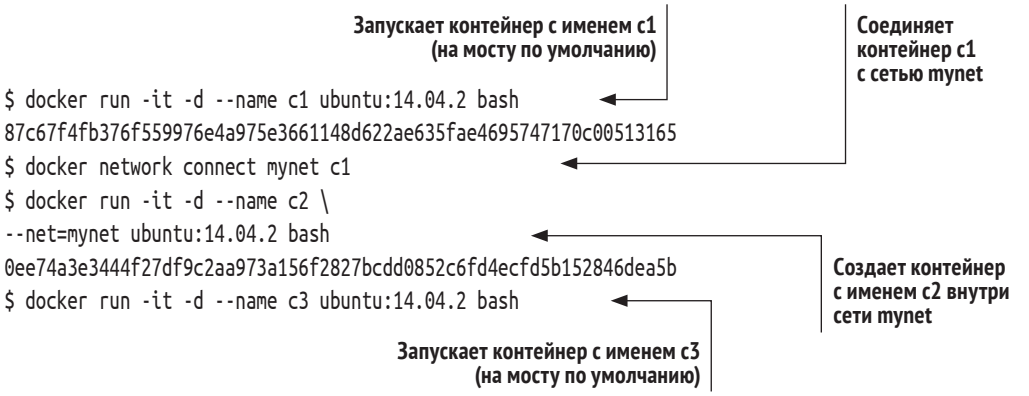
Давайте добавим новую сеть `bridge`, предоставив контейнерам новую плоскую сеть для свободного обмена данными:

```
$ docker network create --driver=bridge mynet
770ffbc81166d54811ecf9839331ab10c586329e72cea2eb53a0229e53e8a37f
$ docker network ls | grep mynet
770ffbc81166        mynet               bridge              local
$ ip addr | grep br-
522: br-91b29e0d29d5: <NOCARRIER,
     BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group
➥ default
```

```
    inet 172.18.0.1/16 scope global br-91b29e0d29d5
$ ip addr | grep docker
5: docker0: <NOCARRIER,
    BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group
➥ default
    inet 172.17.0.1/16 scope global docker0
```

Мы создали новый сетевой интерфейс, который будет использовать другой диапазон IP-адресов, отличный от обычного моста Docker. Для мостов имя нового сетевого интерфейса в настоящее время начинается с `br-`, но в будущем оно может поменяться.

Теперь давайте запустим два контейнера, подключенных к сети:

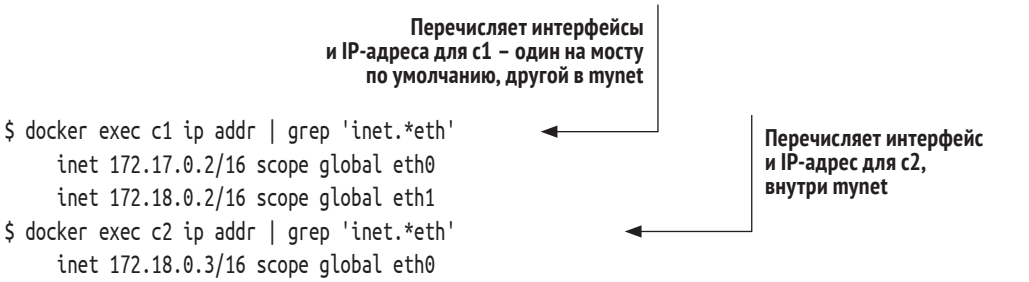


Предыдущие команды демонстрируют два разных способа подключения контейнера к сети: запуск контейнера, затем подключение службы, а также создание и подключение за один шаг.

Между этими двумя способами есть разница. В первом случае мы подключимся к сети по умолчанию при запуске (обычно это мост Docker, но можно настроить с помощью аргумента демона Docker), а затем добавим новый интерфейс, чтобы он также мог получить доступ к mynet.

Во втором случае мы *просто* присоединимся к mynet – ни один контейнер на обычном мосту Docker не сможет получить к нему доступ.

Давайте выполним проверку подключения. Сначала нужно взглянуть на IP-адреса нашего контейнера:



```
$ docker exec c3 ip addr | grep 'inet.*eth'
     inet 172.17.0.3/16 scope global eth0
```

Перечисляет интерфейс и IP-адрес для c3 на мосту по умолчанию

Теперь мы можем выполнить несколько тестов подключения:

Попытка пропинговать имя контейнера 1 из контейнера 2 (успешно)

```
$ docker exec c2 ping -qc1 c1
PING c1 (172.18.0.2) 56(84) bytes of data.

--- c1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.041/0.041/0.041/0.000 ms
$ docker exec c2 ping -qc1 c3
ping: unknown host c3
$ docker exec c2 ping -qc1172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.

--- 172.17.0.3 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
$ docker exec c1 ping -qc1 c2
PING c2 (172.18.0.3) 56(84) bytes of data.

--- c2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.047/0.047/0.047/0.000 ms
$ docker exec c1 ping -qc1 c3
ping: unknown host c3
$ docker exec c1 ping -qc1172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.

--- 172.17.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.095/0.095/0.095/0.000 ms
```

Попытка пропинговать имя и IP-адрес контейнера 3 из контейнера 2 (неудачно)

Попытка пропинговать имя контейнера 2 из контейнера 1 (успешно)

Попытка пропинговать имя и IP-адрес контейнера 3 из контейнера 1 (неудачно, успешно)

Много чего тут происходит! Вот ключевые выводы:

- на новом мосту контейнеры могут пинговать друг друга по IP-адресу и имени;
- на мосту по умолчанию контейнеры могут пинговать друг друга только по IP-адресу;
- контейнеры, расположенные между несколькими мостами, могут получить доступ к контейнерам из любой сети, членами которой они являются;
- контейнеры не могут получить доступ друг к другу через мосты, даже через IP-адрес.

ОБСУЖДЕНИЕ

Эта новая возможность создания моста использовалась в методе 77 с Docker Compose и методе 79 с Blockade, чтобы предоставить контейнерам возможность пинговать друг друга по имени. Но вы также видели, что это очень гибкий элемент функциональности с возможностью моделирования достаточно сложных сетей.

Например, вы можете поэкспериментировать с *узлом-бастионом*. Это специально отведенный компьютер, который обеспечивает доступ к другой более значимой сети. Поместив службы приложений в новый мост, а затем открывая сервисы только через контейнер, подключенный и к мосту по умолчанию, и к новому мосту, вы можете приступить к выполнению реалистичного тестирования на проникновение, оставаясь изолированными на собственном компьютере.

МЕТОД 81 Настройка физической сети с помощью Weave

Физическая сеть – это сетевой слой программного уровня, построенный поверх другой сети. По сути, вы получаете сеть, которая выглядит локальной, но внутри она обменивается данными через другие сети. Это означает, что с точки зрения производительности эта сеть будет вести себя менее надежно, чем локальная сеть, но с точки зрения удобства использования может быть очень удобна: вы будете общаться с узлами в совершенно разных местах, как если бы они находились в одной комнате.

Это особенно интересно в случае с контейнерами Docker – они могут быть незаметно соединены между хостами так же, как и хостами между сетями. Это устраняет необходимость срочного планирования количества контейнеров, которые можно разместить на одном хосте.

ПРОБЛЕМА

Вы хотите легко обмениваться данными между контейнерами через хосты.

РЕШЕНИЕ

Используйте Weave Net (в оставшейся части этого метода мы называем его просто Weave), чтобы настроить сеть, которая позволяет контейнерам общаться друг с другом, как если бы они находились в локальной сети вместе.

Мы продемонстрируем принцип физической сети с помощью Weave (https://www.weave.works/oss/net/), утилиты, разработанной для этой цели. На рис. 10.3 показан обзор типичной сети Weave.

На рис. 10.3 хост 1 не имеет доступа к хосту 3, но они могут общаться друг с другом через сеть Weave, поскольку были подключены локально. Сеть Weave закрыта для посторонних. Это делает разработку, тестирование и развертывание кода в разных средах относительно простым, поскольку топология сети в каждом случае может быть одинаковой.

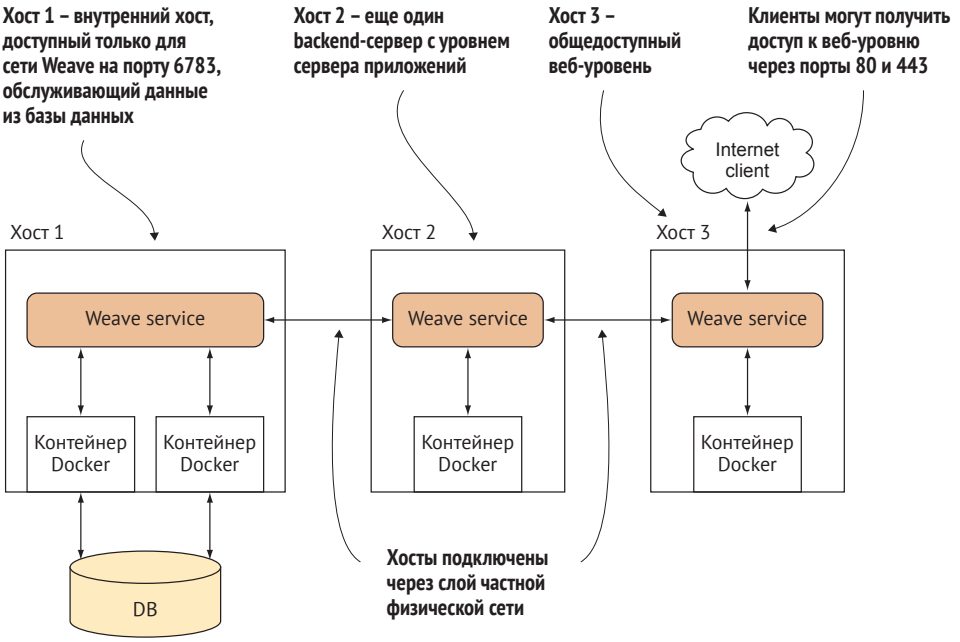


Рис. 10.3 ❖ Типичная сеть Weave

Установка Weave

Weave – это один двоичный файл. Вы можете найти инструкции по его установке по адресу https://www.weave.works/docs/net/latest/install/install-weave/.

Инструкции из предыдущей ссылки (и перечисленные ниже, что было удобнее) нам подошли. Weave необходимо установить на каждом хосте, который вы хотите включить в свою сеть Weave:

```
$ sudo curl -L git.io/weave -o /usr/local/bin/weave
$ sudo chmod +x /usr/local/bin/weave
```

> **ПРЕДУПРЕЖДЕНИЕ.** Если возникли проблемы с этим методом, возможно, на вашем компьютере уже есть двоичный файл Weave, который является частью другого программного пакета.

Настройка Weave

Чтобы следовать этому примеру, вам понадобятся два хоста. Назовем их `host1` и `host2`. Убедитесь, что они могут общаться друг с другом с помощью отправителя пакетов. Вам понадобится IP-адрес первого хоста, на котором вы запускаете Weave.

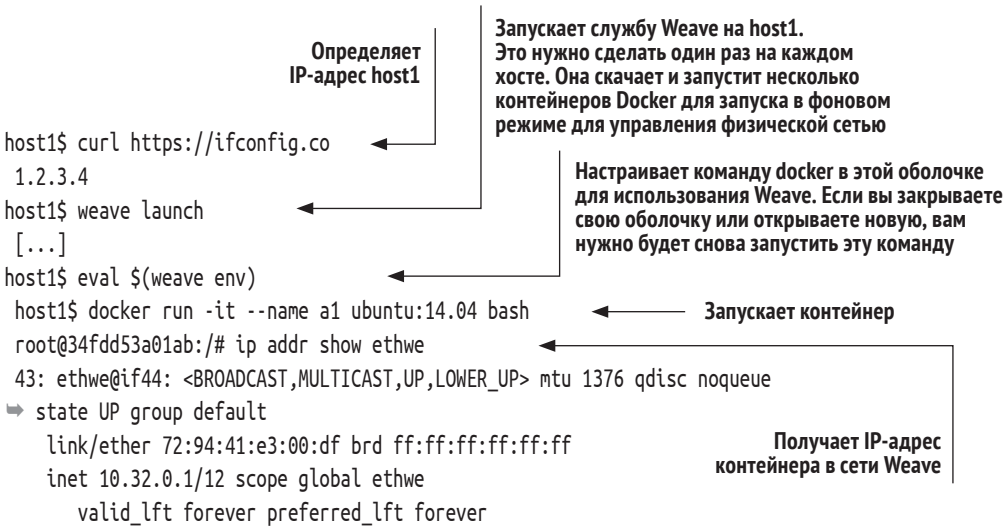
Быстрый способ получить публичный IP-адрес хоста – зайти на сайт https://ifconfig.co или выполнить команду `curl https://ifconfig.co`, но имейте в виду, что, вероятно, нужно будет открыть брандмауэры для обоих хостов, чтобы подключиться через открытый интернет.

Вы также можете запустить Weave в локальной сети, если выберете правильный IP-адрес.

> **СОВЕТ.** Если у вас возникли проблемы с этим методом, вполне вероятно, что сеть каким-то образом была запрещена брандмауэром. Если вы не уверены, обратитесь к администратору сети.

В частности, нужно открыть порт 6783 для TCP и UDP и 6784 для UDP.

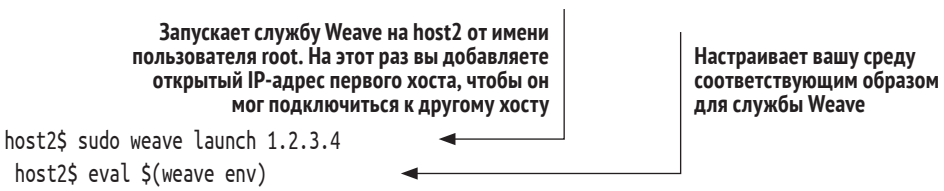
На первом хосте вы можете запустить первый маршрутизатор Weave:




```
host1$ curl https://ifconfig.co
 1.2.3.4
host1$ weave launch
 [...]
host1$ eval $(weave env)
 host1$ docker run -it --name a1 ubuntu:14.04 bash
 root@34fdd53a01ab:/# ip addr show ethwe
 43: ethwe@if44: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue
➥ state UP group default
    link/ether 72:94:41:e3:00:df brd ff:ff:ff:ff:ff:ff
    inet 10.32.0.1/12 scope global ethwe
       valid_lft forever preferred_lft forever
```

Weave заботится о включении в контейнер дополнительного интерфейса `ethwe`, который предоставляет IP-адрес в сети Weave.

Вы можете выполнить аналогичные шаги для второго хоста, но сообщите Weave о расположении host1:



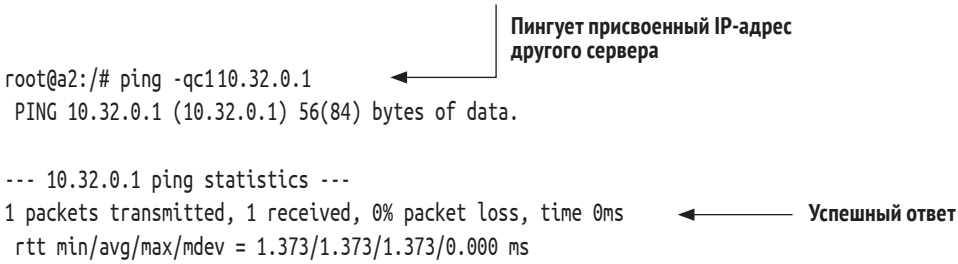

```
host2$ sudo weave launch 1.2.3.4
 host2$ eval $(weave env)
 host2$ docker run -it --name a2 ubuntu:14.04 bash
 root@a2:/# ip addr show ethwe
 553: ethwe@if554: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue
➥ state UP group default
 link/ether fe:39:ca:74:8a:ca brd ff:ff:ff:ff:ff:ff
 inet 10.44.0.0/12 scope global ethwe
    valid_lft forever preferred_lft forever
```

Единственное отличие `host2` заключается в том, что вы сообщаете Weave, что он должен взаимодействовать с Weave на `host1` (указанном с помощью IP-

адреса или имени хоста и (необязательно) :port, через который host2 может связаться с ним).

Проверка соединения

Теперь, когда у вас все настроено, можете проверить, могут ли ваши контейнеры общаться друг с другом. Возьмем контейнер на host2:



Если вы получаете успешный ответ, у вас есть проверенная связь в собственной частной сети, охватывающей два хоста. Также можете проверять соединение по имени контейнера, как в случае с пользовательским мостом.

> **СОВЕТ.** Возможно, это не сработает из-за того, что сообщения протокола ICMP (используемые ping) блокируются брандмауэром. Если это не работает, попробуйте получить доступ к порту 6783 через протокол Telnet на другом хосте, чтобы проверить, можно ли устанавливать соединения.

ОБСУЖДЕНИЕ

Физическая сеть – это мощный инструмент для наведения порядка в хаотическом мире сетей и брандмауэров. Weave даже претендует на интеллектуальную маршрутизацию вашего трафика через частично разделенные сети, где некий хост B может видеть A и C, но A и C не могут общаться – это может быть знакомо из метода 80. Тем не менее имейте в виду, что иногда эти сложные сетевые установки существуют по определенной причине – весь смысл узлов-бастионов – изоляция в целях безопасности.

Вся эта мощность обходится дорого: есть сообщения о том, что сети Weave иногда работают значительно медленнее, чем «сырые» сети, и приходится запускать дополнительные механизмы управления в фоновом режиме (поскольку модель плагинов для сетей не охватывает все случаи использования).

Сеть Weave имеет много дополнительных функциональных возможностей, от визуализации до интеграции с Kubernetes (познакомимся с Kubernetes, используемым в качестве системы оркестровки в методе 88). Мы рекомендуем вам посмотреть обзор Weave Net, чтобы узнать больше и получить максимальную отдачу от вашей сети, – https://www.weave.works/docs/net/latest/overview. Мы не рассмотрели встроенный плагин для оверлейной сети. В зависимости от вашего варианта использования, вероятно, стоит провести исследования

в качестве возможной замены Weave, хотя для этого требуется либо использовать режим Swarm (метод 87), либо создать глобально доступное хранилище «ключ/значение» (возможно, etcd, из метода 74).

Резюме

- Docker Compose можно использовать для настройки кластеров контейнеров.
- Comcast и Blockade являются полезными инструментами для тестирования контейнеров в плохих сетях.
- Виртуальные сети Docker являются альтернативой связыванию.
- Вы можете вручную моделировать сети в Docker с помощью виртуальных сетей.
- Weave Net полезен для объединения контейнеров в хостах.

Часть 4

Оркестровка от одного компьютера до облака

Часть 4 охватывает важную часть оркестровки. Как только вы запустите любое количество контейнеров в одной и той же среде, нужно будет подумать о том, как управлять ими последовательным и надежным способом, поэтому мы рассмотрим некоторые из самых популярных инструментов, доступных в настоящее время.

В главе 11 объясняется важность оркестровки. Она начинается с рассказа об управлении Docker на основе одного хоста с помощью systemd и завершается использованием обнаружения сервисов в сети с помощью Consul и Registrator.

Глава 12 посвящена кластерной среде Docker, где мы кратко расскажем о Docker Swarm, прежде чем перейти к Kubernetes, самой популярной системе оркестровки в мире. Затем изменим ситуацию и покажем вам, как использовать Docker для локального моделирования сервисов AWS. Наконец, мы расскажем о создании Docker-фреймворка на Mesos.

Глава 13 – это подробное обсуждение факторов, которые могут учитываться при выборе платформы на основе Docker. Варианты выбора могут сбить вас с толку, но это поможет вам структурировать свои мысли и принять более правильное решение, если нужно.

Глава 11

Основы оркестровки контейнеров

О чем рассказывается в этой главе:

- управление простыми сервисами Docker с помощью systemd;
- управление сервисами Docker с несколькими хостами с помощью Helios;
- использование Consul от компании Hashicorp для обнаружения сервисов;
- регистрация сервисов с использованием Registrator.

Технология, на которой построен Docker, существует некоторое время в разных формах, но Docker – это решение, которому удалось привлечь интерес технологической индустрии. Это ставит Docker в завидное положение – разум Docker выполнил начальную работу по запуску экосистемы инструментов, которая превратилась в нескончаемый людской цикл. Люди втягиваются в экосистему и вносят свой вклад в нее.

Это особенно очевидно, когда дело доходит до оркестровки. Увидев список названий компаний с предложениями в этом пространстве, вы будете прощены за то, что думаете, что у каждого свое мнение о том, как что-то делать, и он разработал свой собственный инструмент.

Хотя экосистема и является огромной силой Docker (и именно поэтому мы столько черпаем из нее в этой книге), огромное количество всевозможных инструментов оркестровки может ошеломить как новичков, так и опытных пользователей. В этой главе мы рассмотрим некоторые из наиболее заметных инструментов и дадим вам представление о предложениях высокого уровня, чтобы вы были лучше осведомлены, когда речь заходит о том, что вы хотитеот фреймворка.

Существует много способов организации генеалогических деревьев инструментов оркестровки. На рис. 11.1 показаны инструменты, с которыми мы

знакомы. В корне дерева находится `docker run`, наиболее распространенный способ запуска контейнера. Все, на что вдохновил Docker, является следствием этой команды. С левой стороны расположены инструменты, которые рассматривают группы контейнеров как единый объект. Посередине показаны инструменты, предназначенные для управления контейнерами в рамках systemd и служебных файлов. Наконец, правая сторона рассматривает отдельные контейнеры как они есть. По мере продвижения вниз по веткам инструменты в конечном итоге будут делать для вас больше, будь то работа на нескольких хостах или избавление от утомительного ручного развертывания контейнера.

На диаграмме вы заметите две, казалось бы, изолированные области – Mesos и группу, где указаны Consul, etcd и Zookeeper. Mesos – интересный случай – он существовал еще до появления Docker, и поддержка Docker, которой он располагает, – это дополнительная функция, а не основная функциональность. Тем не менее он работает очень хорошо и должен тщательно оцениваться, чтобы увидеть, какие его функции вам могут понадобиться в других инструментах. Consul, etcd и Zookeeper, напротив, вообще не являются инструментами оркестровки. Вместо этого они обеспечивают важное дополнение к оркестровке: обнаружение служб.

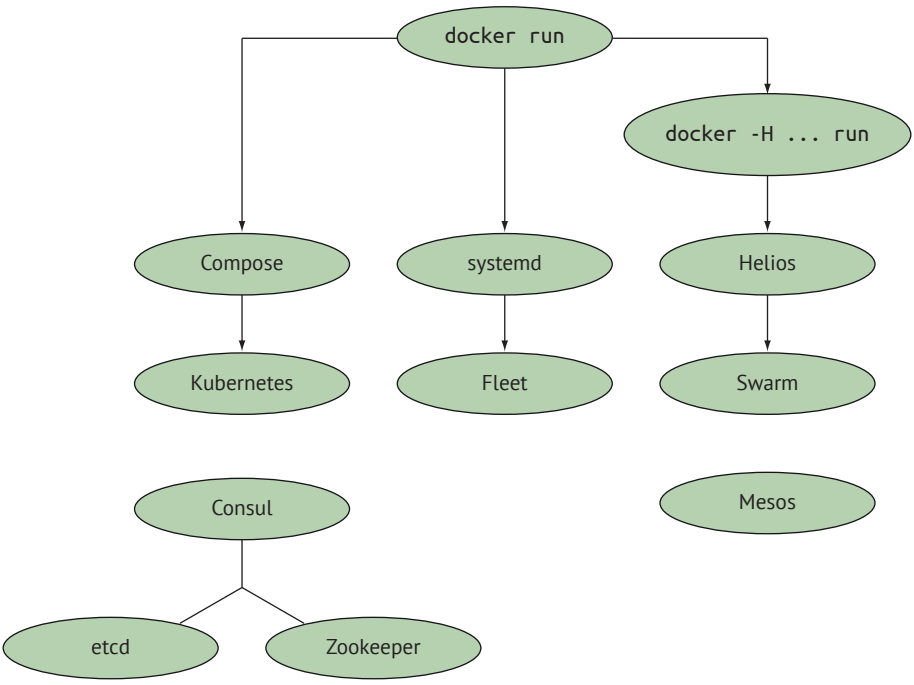


Рис. 11.1 ❖ Инструментальные средства оркестровки в экосистеме Docker

В данной и последующих главах будет рассказано об этой экосистеме оркестровки. Здесь мы познакомимся с инструментами, которые дают вам более детальный контроль, и переход от управления контейнерами вручную может

показаться не таким значительным. Мы рассмотрим управление контейнерами Docker на одном и нескольких хостах, а затем посмотрим, как сохранять и получать информацию о том, где были развернуты контейнеры. После этого в следующей главе изучим более полные решения, которые абстрагируются от многих деталей.

При прочтении этих двух глав может оказаться полезным сделать шаг назад, когда вы подходите к каждому инструменту оркестровки и пытаетесь придумать сценарий, где этот инструмент был бы полезен. Это поможет уточнить, подходит ли вам конкретный инструмент. Мы приведем несколько примеров, которые помогут вам начать работу.

Мы начнем не спеша, устремляя взор внутрь отдельного компьютера.

11.1. Простой Docker с одним хостом

Управление контейнерами на вашем локальном компьютере может быть болезненным. Функции, предоставляемые Docker для управления длительно работающими контейнерами, относительно примитивны, и запуск контейнеров с помощью соединений и общих томов может быть удручающе ручным процессом.

В главе 10 мы рассмотрели использование Docker Compose, чтобы упростить управление соединениями, поэтому сейчас разберемся с другим болевым моментом и увидим, как можно сделать управление длительно работающими контейнерами на одном компьютере более надежным.

МЕТОД 82 Управление контейнерами на вашем хосте с помощью systemd

В этом методе мы проведем вас через настройку простой службы Docker с помощью systemd.

Если вы уже знакомы с systemd, эта глава будет относительно легкой для понимания, но мы не предполагаем предварительных знаний этого инструмента.

Использование systemd для управления Docker может быть полезно для зрелой компании с операционной командой, предпочитающей придерживаться проверенных технологий, которые они уже знают и для которых у них имеются инструменты.

ПРОБЛЕМА

Вы хотите управлять запуском служб контейнера Docker на своем хосте.

РЕШЕНИЕ

Используйте systemd для управления своими службами.

systemd – это демон управления системой, который некоторое время назад заменил сценарии инициализации SysV в Fedora. Он управляет службами в вашей системе – от точек монтирования до процессов и одноразовых

сценариев – как отдельными «юнитами». Его популярность растет, поскольку он распространяется на другие дистрибутивы и операционные системы, хотя в некоторых системах (на момент написания этой главы, например, в Gentoo) могут возникать проблемы с его установкой и включением. Стоит посмотреть на опыт с systemd, полученный другими пользователями при настройке, аналогичной вашей.

В этом методе мы продемонстрируем, как systemd может управлять запуском ваших контейнеров, запустив приложение из главы 1.

Установка systemd

Если в вашей хост-системе нет systemd (можно проверить это, выполнив `systemctl status`, и посмотреть, получаете ли вы последовательный ответ), вы можете установить его непосредственно в операционной системе хоста с помощью стандартного менеджера пакетов.

Если вам неудобно вмешиваться в вашу хост-систему таким образом, рекомендуемый способ поэкспериментировать с ней – использовать Vagrant для подготовки виртуальной машины, готовой к работе с системой, как показано в следующем листинге. Мы кратко расскажем об этом здесь, но см. приложение C для получения дополнительных советов по установке Vagrant.

Листинг 11.1. Настройка Vagrant

```
$ mkdir centos7_docker
$ cd centos7_docker
$ vagrant init jdiprizio/centos-docker-io
$ vagrant up
$ vagrant ssh
```

Создает новую папку и входит в нее

Инициализирует папку для использования в качестве среды Vagrant, указывая образ Vagrant

Запускает виртуальную машину

Подключается к виртуальной машине по SSH

ПРИМЕЧАНИЕ. На момент написания этой главы jdiprizio/centos-docker-io является подходящим и доступным образом виртуальной машины. Если он больше не доступен, когда вы читаете это, можете заменить эту строку из предыдущего листинга другим именем образа. Вы можете найти его на странице компании HashiCorp “Discover Vagrant Boxes”: https://app.vagrantup.com/boxes/search (box – это термин, который Vagrant использует для обозначения образа виртуальной машины). Чтобы найти этот образ, мы ввели в строку поиска «docker centos». Возможно, вам понадобится найти справку по команде `add vagrant box`, чтобы выяснить как скачать новую виртуальную машину, прежде чем попытаться ее запустить.

Настройка простого приложения Docker для systemd

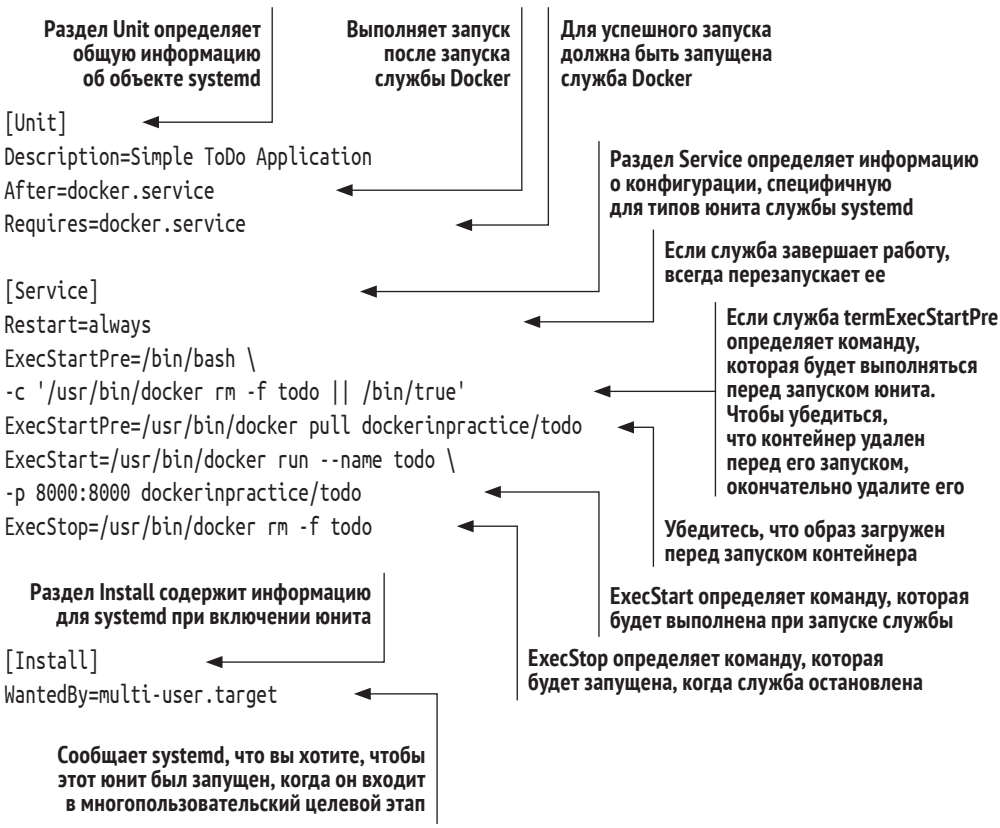
Теперь, когда у вас есть машина с systemd и Docker, мы воспользуемся ею для запуска приложения из главы 1.

Systemd работает, читая файлы конфигурации в простом формате INI.

> **ПОДСКАЗКА.** Файлы INI – это простые текстовые файлы с базовой структурой, состоящей из разделов, свойств и значений.

Сначала вам нужно создать файл службы в качестве пользователя root в /etc/systemd/system/todo.service, как показано в следующем листинге. В этом файле вы сообщаете systemd запустить контейнер Docker с именем «todo» на порту 8000 на этом хосте.

Листинг 11.2. /etc/systemd/system/todo.service



Этот файл конфигурации должен прояснить, что systemd предлагает простую декларативную схему для управления процессами, оставляя детали управления зависимостями службе systemd. Это не значит, что можно игнорировать

детали, но в вашем распоряжении появляется множество инструментов для управления процессами Docker (и другими процессами).

> **ПРИМЕЧАНИЕ.** Docker не устанавливает политики перезапуска контейнеров по умолчанию, но имейте в виду, что любые установленные вами политики будут конфликтовать с большинством менеджеров процессов. Не устанавливайте политики перезапуска, если вы используете диспетчер процессов.

Чтобы включить новый юнит, просто вызовите команду `systemctl enable`. Если хотите, чтобы этот юнит запускался автоматически при загрузке системы, вы также можете создать символическую ссылку в каталоге multi-user.target.wants. После этого запустите его с помощью команды `systemctl start`.

```
$ systemctl enable /etc/systemd/system/todo.service
$ ln -s '/etc/systemd/system/todo.service' \
'/etc/systemd/system/multi-user.target.wants/todo.service'
$ systemctl start todo.service
```

Потом просто подождите начала. Если появится проблема, вы будете проинформированы.

Чтобы проверить, что все в порядке, используйте команду `systemctl status`. Она выведет общую информацию о юните, например, как долго он работал, и идентификатор процесса, а затем ряд строк журнала процесса. В этом случае мы видим слова: `Swarm server started port 8000`. Это хороший знак:

```
[root@centos system]# systemctl status todo.service
todo.service - Simple ToDo Application
   Loaded: loaded (/etc/systemd/system/todo.service; enabled)
   Active: active (running) since Wed 2015-03-04 19:57:19 UTC; 2min 13s ago
  Process: 21266 ExecStartPre=/usr/bin/docker pull dockerinpractice/todo \
(code=exited, status=0/SUCCESS)
  Process: 21255 ExecStartPre=/bin/bash -c /usr/bin/docker rm -f todo || \
/bin/true (code=exited, status=0/SUCCESS)
  Process: 21246 ExecStartPre=/bin/bash -c /usr/bin/docker kill todo || \
/bin/true (code=exited, status=0/SUCCESS)
 Main PID: 21275 (docker)
   CGroup: /system.slice/todo.service
??21275 /usr/bin/docker run --name todo
➥ -p 8000:8000 dockerinpractice/todo
Mar 04 19:57:24 centos docker[21275]: TodoApp.js:117: \
// TODO scroll into view
Mar 04 19:57:24 centos docker[21275]: TodoApp.js:176: \
if (i>=list.length()) { i=list.length()-1; } // TODO .length
Mar 04 19:57:24 centos docker[21275]: local.html:30: \
<!-- TODO 2-split, 3-split -->
```

```
Mar 04 19:57:24 centos docker[21275]: model/TodoList.js:29: \
// TODO one op - repeated spec? long spec?
Mar 04 19:57:24 centos docker[21275]: view/Footer.jsx:61: \
// TODO: show the entry's metadata
Mar 04 19:57:24 centos docker[21275]: view/Footer.jsx:80: \
todoList.addObject(new TodoItem()); // TODO create default
Mar 04 19:57:24 centos docker[21275]: view/Header.jsx:25: \
// TODO list some meaningful header (apart from the id)
Mar 04 19:57:24 centos docker[21275]: > todomvc-swarm@0.0.1 start /todo
Mar 04 19:57:24 centos docker[21275]: > node TodoAppServer.js
Mar 04 19:57:25 centos docker[21275]: Swarm server started port 8000
```

Теперь вы можете посетить сервер на порту 8000.

ОБСУЖДЕНИЕ

Принципы этого метода могут быть применены не только к systemd – большинство менеджеров процессов, включая другие системы инициализации, могут быть настроены аналогичным образом. Если вам интересно, можете использовать systemd для замены существующих служб, работающих в вашей системе (возможно, базы данных PostgreSQL), на Docker'изированные.

В следующем методе мы пойдем дальше, внедрив в systemd сервер SQLite, созданный нами в методе 77.

МЕТОД 83 Оркестровка запуска контейнеров на вашем хосте

В отличие от docker-compose (на момент написания этой главы), systemd представляет собой зрелую технологию, готовую к эксплуатации. В этом методе мы покажем вам, как добиться функциональности локальной оркестровки, аналогичной docker-compose, используя systemd.

> **ПРИМЕЧАНИЕ.** Если возникли проблемы, вам может потребоваться обновить версию Docker. Версия 1.7.0 или выше должна работать нормально.

ПРОБЛЕМА

Вы хотите управлять более сложной оркестровкой контейнеров на одном хосте при эксплуатации.

РЕШЕНИЕ

Используйте systemd с зависимыми службами для управления контейнерами.

Чтобы продемонстрировать использование systemd для более сложного сценария, мы переопределим пример из метода 77, где используется TCP-сервер SQLite, в systemd. На рис. 11.2 показаны зависимости для нашей запланированной конфигурации юнита службы systemd.

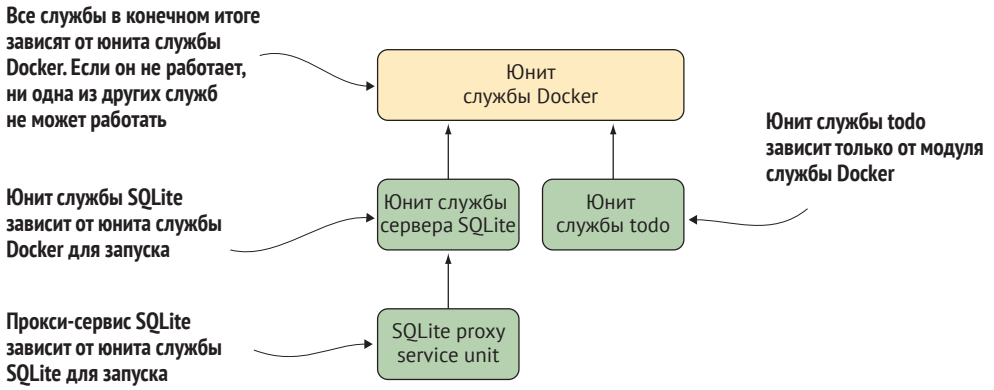


Рис. 11.2 ❖ Граф зависимостей systemd

Эта схема аналогична той, что вы видели в примере с Docker Compose в методе 77. Ключевое отличие здесь заключается в том, что вместо службы SQLite, рассматриваемой как одна монолитная сущность, каждый контейнер является дискретной сущностью. В этом случае прокси-сервер SQLite может быть остановлен независимо от сервера SQLite.

Это листинг для службы сервера SQLite. Как и раньше, он зависит от службы Docker, но у него есть пара отличий от примера из предыдущего метода.

Листинг 11.3. /etc/systemd/system/sqliteserver.service

Раздел Unit определяет общую информацию об объекте systemd

Запускает данный юнит после запуска службы Docker

Для успешного запуска этого юнита должна быть запущена служба Docker

```
[Unit]
Description=SQLite Docker Server
After=docker.service
Requires=docker.service

[Service]
Restart=always
ExecStartPre=-/bin/touch /tmp/sqlitedbs/test
 ExecStartPre=-/bin/touch /tmp/sqlitedbs/live
 ExecStartPre=/bin/bash \
-c '/usr/bin/docker kill sqliteserver || /bin/true'
 ExecStartPre=/bin/bash \
-c '/usr/bin/docker rm -f sqliteserver || /bin/true'
 ExecStartPre=/usr/bin/docker \
pull dockerinpractice/docker-compose-sqlite
```

Эти строки гарантируют, что файлы базы данных SQLite существуют до запуска службы. Косая черта перед командой touch указывает systemd, что запуск должен завершиться неудачно, если команда возвращает код ошибки

ExecStartPre определяет команду, которая будет выполнена до запуска юнита. Чтобы убедиться, что контейнер удален до запуска, в данном случае вы должны окончательно удалить его

Убедитесь, что образ скачан, перед запуском контейнера

```
 ExecStart=/usr/bin/docker run --name sqliteserver \
 -v /tmp/sqlitedbs/test:/opt/sqlite/db \
dockerinpractice/docker-compose-sqlite /bin/bash -c \
'socat TCP-L:12345,fork,reuseaddr \
EXEC:"sqlite3 /opt/sqlite/db",pty'
ExecStop=/usr/bin/docker rm -f sqliteserver

[Install]
WantedBy=multi-user.target
```

ExecStart определяет команду, которая будет выполнена, когда запущена служба. Обратите внимание, что мы поместили команду socat в вызов «/bin/bash –c», чтобы избежать путаницы, так как строка ExecStart выполняется systemd

ExecStop определяет команду, которая будет выполнена, когда служба остановлена

СОВЕТ. В system пути должны быть абсолютными.

Теперь идет листинг для службы sqliteproxy. Ключевое отличие здесь заключается в том, что эта служба зависит от только что определенного вами серверного процесса, который, в свою очередь, зависит от службы Docker.

Листинг 11.4. /etc/systemd/system/sqliteproxy.service

```
[Unit]
Description=SQLite Docker Proxy
After=sqliteserver.service
Requires=sqliteserver.service

[Service]
Restart=always
ExecStartPre=/bin/bash -c '/usr/bin/docker kill sqliteproxy || /bin/true'
ExecStartPre=/bin/bash -c '/usr/bin/docker rm -f sqliteproxy || /bin/true'
ExecStartPre=/usr/bin/docker pull dockerinpractice/docker-compose-sqlite
ExecStart=/usr/bin/docker run --name sqliteproxy \
-p 12346:12346 --link sqliteserver:sqliteserver \
dockerinpractice/docker-compose-sqlite /bin/bash \
-c 'socat TCP-L:12346,fork,reuseaddr TCP:sqliteserver:12345'
ExecStop=/usr/bin/docker rm -f sqliteproxy

[Install]
WantedBy=multi-user.target
```

Прокси-юнит должен работать после ранее определенной службы sqliteserver

Прокси-сервер требует, чтобы экземпляр сервера работал до его запуска

Запускает контейнер

С помощью этих двух файлов конфигурации мы заложили основу для установки и запуска службы SQLite под управлением systemd. Теперь можем включить эти службы:

```
$ sudo systemctl enable /etc/systemd/system/sqliteserver.service
ln -s '/etc/systemd/system/sqliteserver.service' \
```

```
'/etc/systemd/system/multi-user.target.wants/sqliteserver.service'
$ sudo systemctl enable /etc/systemd/system/sqliteproxy.service
ln -s '/etc/systemd/system/sqliteproxy.service' \
'/etc/systemd/system/multi-user.target.wants/sqliteproxy.service'
```

И запустить их:

```
$ sudo systemctl start sqliteproxy
$ telnet localhost 12346
[vagrant@centos ~]$ telnet localhost 12346
Trying ::1...
Connected to localhost.
Escape character is '^]'.
SQLite version 3.8.22013-12-0614:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from t1;
select * from t1;
test
```

Обратите внимание: поскольку служба sqliteproxy зависит от запуска службы sqliteserver, вам только нужно запустить прокси-сервер – зависимости запускаются автоматически.

ОБСУЖДЕНИЕ

Одной из проблем при администрировании долго выполняющегося приложения на локальном компьютере является управление службами зависимостей. Например, веб-приложение может ожидать, что оно будет работать в фоновом режиме в качестве службы, но также может зависеть от базы данных и веб-сервера. Это вам знакомо – вы рассматривали структуру web-app-db в методе 13.

Метод 76 показал, как настроить структуру такого типа с зависимостями и т. д., но такие инструменты, как systemd, уже давно работают над этой проблемой и могут предложить гибкость, которой нет у Docker Compose. Например, после того как вы написали свои служебные файлы, можете запустить любой из них, который вам нужен, а systemd обработает запуск любых зависимых служб, даже запуск самого демона Docker, если это необходимо.

11.2. Docker с несколькими хостами

Теперь, когда вы знакомы с довольно сложными расстановками контейнеров Docker на компьютере, пришло время подумать о большем – давайте перейдем к миру нескольких хостов, чтобы иметь возможность использовать Docker в более широком масштабе.

В оставшейся части этой главы вы вручную запустите среду с несколькими хостами с помощью Helios, чтобы познакомиться с мультихостовыми концепциями Docker. В следующей главе вы увидите более автоматизированные и изощренные способы достижения того же результата и даже больше.

МЕТОД 84 Мультихостовый Docker и Helios

Передача всего контроля над подготовкой группы компьютеров для приложения может пугать вас, поэтому будет не лишним облегчить себе задачу, используя более ручной подход.

Helios идеально подходит для компаний, которые в основном имеют статическую инфраструктуру и заинтересованы в использовании Docker для своих критически важных служб, но (по понятным причинам) хотят, чтобы в процессе наблюдался человеческий контроль.

ПРОБЛЕМА

Вы хотите иметь возможность снабдить несколько хостов Docker контейнерами, но при этом сохранить ручное управление тем, что и где выполняется.

РЕШЕНИЕ

Используйте Helios от компании Spotify, чтобы четко управлять контейнерами на других хостах.

Helios – это инструментальное средство, которое Spotify использует в настоящее время для управления своими серверами в производственном процессе. Оно обладает приятным свойством простоты начала работы и стабильности (как вы надеетесь). Helios позволяет управлять развертыванием контейнеров Docker на нескольких хостах и предоставляет единый интерфейс командной строки, который вы можете использовать, чтобы указать, что хотите запустить и где, а также возможность взглянуть на текущее состояние игры.

Поскольку мы только знакомимся с Helios, для простоты будем запускать все на одном узле в Docker. Не беспокойтесь: все, что имеет отношение к работе на нескольких хостах, будет четко выделено. Архитектура высокого уровня Helios показана на рис. 11.3.

Как видно, при запуске Helios требуется только одна дополнительная служба: Zookeeper. Helios использует Zookeeper для отслеживания состояния всех ваших хостов и в качестве канала связи между ведущими устройствами и агентами.

> **ПОДСКАЗКА.** Zookeeper – это легкая распределенная база данных, написанная на языке Java, которая оптимизирована для хранения информации о конфигурации. Это часть пакета программных продуктов с открытым исходным кодом Apache. По функциональности она похожа на etcd (о котором вы узнали в главе 9 и который увидите снова в этой главе).

Все, что вам нужно знать для этого метода, – это то, что Zookeeper хранит данные таким образом, что они могут быть распределены по нескольким узлам (как для масштабируемости, так и для надежности) путем запуска нескольких экземпляров Zookeeper. Это может показаться знакомым нашему описанию etcd в главе 9 – у них много общего.

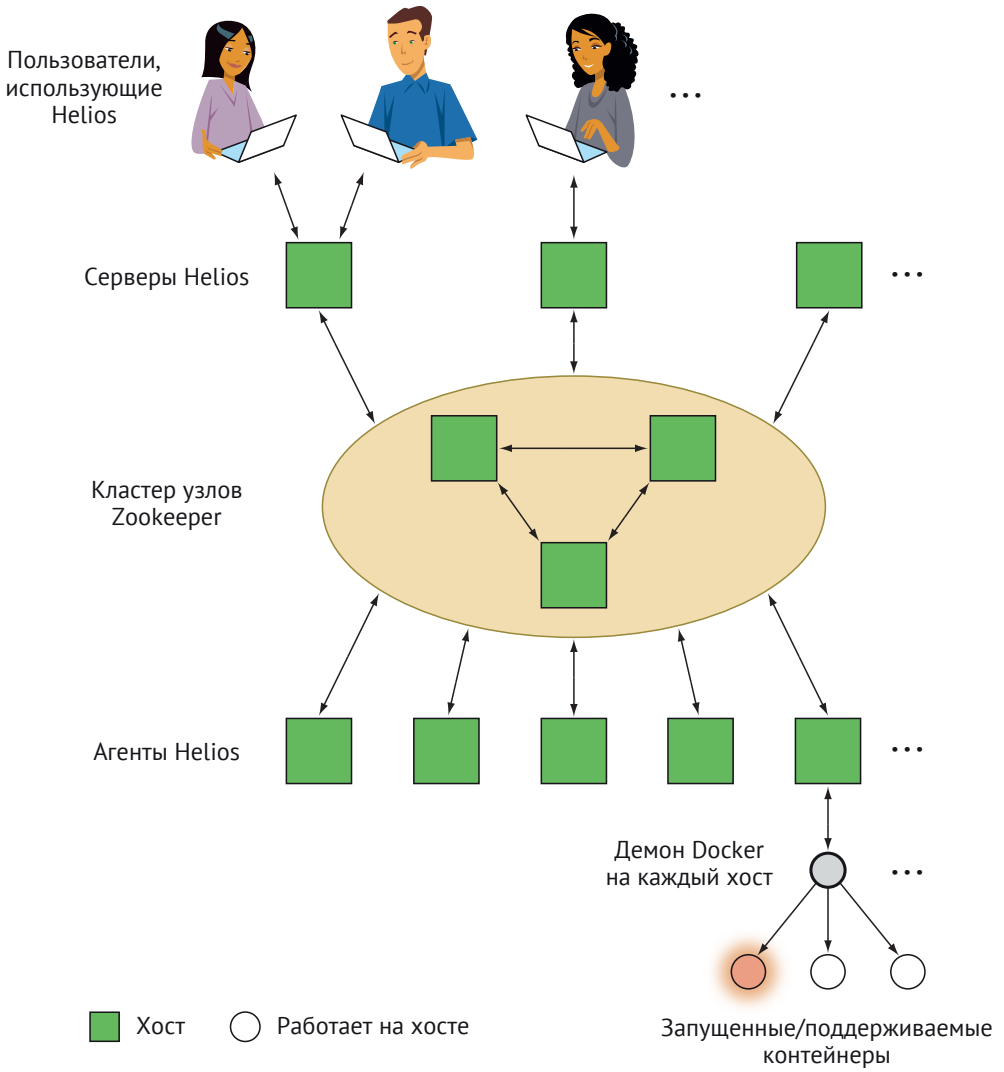


Рис. 11.3 ❖ Взгляд на установку Helios с высоты птичьего полета

Чтобы запустить отдельный экземпляр Zookeeper, который мы будем использовать в этом методе, выполните следующую команду:

```
$ docker run --name zookeeper -d jplock/zookeeper:3.4.6
cd0964d2ba18baac58b29081b227f15e05f11644adfa785c6e9fc5dd15b85910
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' zookeeper
172.17.0.9
```

> **ПРИМЕЧАНИЕ.** При запуске экземпляра Zookeeper на его собственном узле вы захотите открыть порты, чтобы сделать его доступным для других хостов, и использовать тома для сохранения данных. Посмотрите на файл Dockerfile на Docker Hub для получения подробной информации о том, какие порты и папки вы должны использовать (https://hub.docker.com/r/jplock/zookeeper/~/dockerfile/). Также вероятно, что вы захотите запустить Zookeeper на нескольких узлах, но настройка кластера Zookeeper выходит за рамки этого метода.

Можете проверить данные, хранящиеся в Zookeeper, с помощью инструментального средства zkCli.sh либо в интерактивном режиме, либо передав ему ввод. Первоначальный запуск довольно многословный, но вы попадете в интерактивную подсказку, в которой можно выполнять команды для файловой древовидной структуры, где Zookeeper хранит данные.

```
$ docker exec -it zookeeper bin/zkCli.sh
Connecting to localhost:2181
2015-03-07 02:56:05,076 [myid:] - INFO [main:Environment@100] - Client >
environment:zookeeper.version=3.4.6-1569965, built on 02/20/2014 09:09 GMT
2015-03-07 02:56:05,079 [myid:] - INFO [main:Environment@100] - Client >
environment:host.name=917d0f8ac077
2015-03-07 02:56:05,079 [myid:] - INFO [main:Environment@100] - Client >
environment:java.version=1.7.0_65
2015-03-07 02:56:05,081 [myid:] - INFO [main:Environment@100] - Client >
environment:java.vendor=Oracle Corporation
[...]
2015-03-07 03:00:59,043 [myid:] - INFO
➥ [main-SendThread(localhost:2181):ClientCnxn$SendThread@1235] -
➥ Session establishment complete on server localhost/0:0:0:0:0:0:0:1:2181,
➥ sessionid = 0x14bf223e159000d, negotiated timeout = 30000

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper]
```

В Zookeeper пока ничего нет, поэтому единственное, что в данный момент хранится, это какая-то внутренняя информация. Оставьте это приглашение открытым, и мы вернемся к нему по мере продвижения.

Сам по себе Helios разделен на три части:

- *сервер (master)* – используется в качестве интерфейса для внесения изменений в Zookeeper;
- *агент (agent)* – запускается на каждом хосте Docker, запускает и останавливает контейнеры на основе Zookeeper и сообщает о состоянии;

- *инструменты командной строки* – используются для отправки запросов мастеру.

На рис. 11.4 показано, как конечная система связывается воедино, когда мы выполняем в ней какое-либо действие (стрелки показывают поток данных).

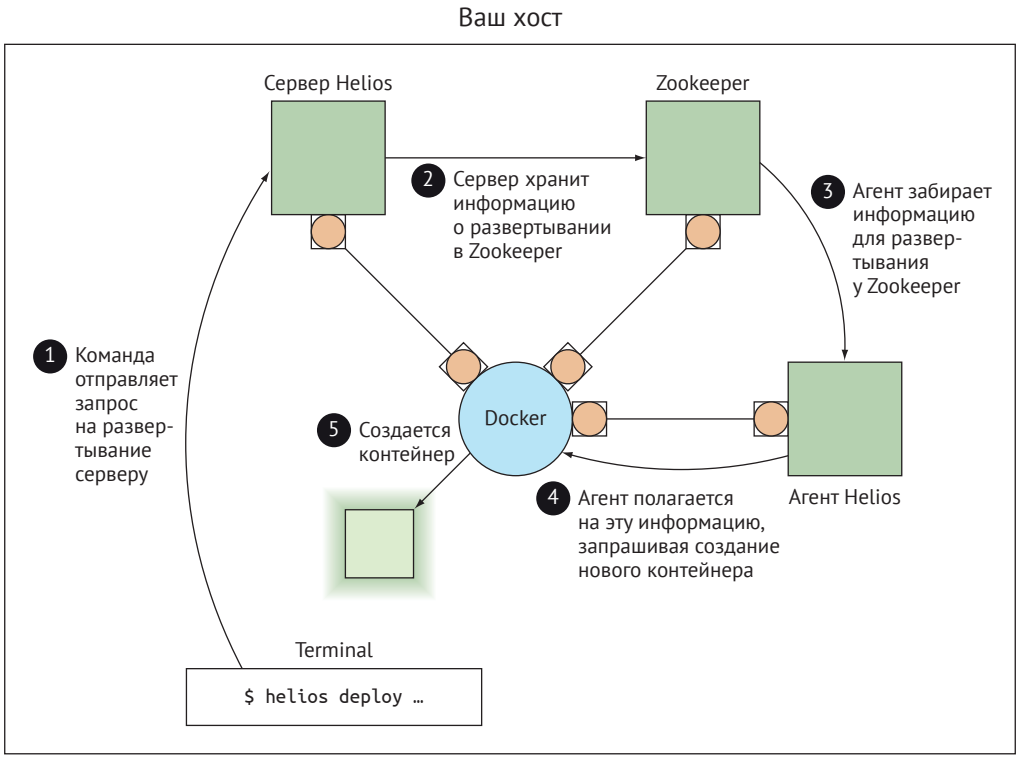


Рис. 11.4 ❖ Запуск контейнера при установке Helios с одним хостом

Теперь, когда Zookeeper работает, пришло время запустить Helios. Нам нужно запустить сервер, указав IP-адрес узла Zookeeper, запущенный ранее:

```
$ IMG=dockerinpractice/docker-helios
$ docker run -d --name hmaster $IMG helios-master --zk 172.17.0.9
896bc963d899154436938e260b1d4e6fdb0a81e4a082df50043290569e5921ff
$ docker logs --tail=3 hmaster
03:20:14.460 helios[1]: INFO [MasterService STARTING] ContextHandler: >
Started i. d.j.MutableServletContextHandler@7b48d370{/,null,AVAILABLE}
03:20:14.465 helios[1]: INFO [MasterService STARTING] ServerConnector: >
Started application@2192bcac{HTTP/1.1}{0.0.0.0:5801}
03:20:14.466 helios[1]: INFO [MasterService STARTING] ServerConnector: >
Started admin@28a0d16c{HTTP/1.1}{0.0.0.0:5802}
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' hmaster
172.17.0.11
```

Теперь посмотрим, что нового у Zookeeper:

```
[zk: localhost:2181(CONNECTED) 1] ls /
[history, config, status, zookeeper]
[zk: localhost:2181(CONNECTED) 2] ls /status/masters
[896bc963d899]
[zk: localhost:2181(CONNECTED) 3] ls /status/hosts
[]
```

Похоже, что сервер Helios создал кучу новых фрагментов конфигурации, включая регистрацию самого себя в качестве сервера. К сожалению, у нас пока нет хостов.

Давайте решим эту проблему, запустив агента, который будет использовать сокет текущего хоста для запуска контейнеров:

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock -d --name hagent \
dockerinpractice/docker-helios helios-agent --zk 172.17.0.9
5a4abcb271070d0171ca809ff2beafac5798e86131b72aeb201fe27df64b2698
$ docker logs --tail=3 hagent
03:30:53.344 helios[1]: INFO [AgentService STARTING] ContextHandler: >
Started i. d.j.MutableServletContextHandler@774c71b1{/,null,AVAILABLE}
03:30:53.375 helios[1]: INFO [AgentService STARTING] ServerConnector: >
Started application@7d9e6c27{HTTP/1.1}{0.0.0.0:5803}
03:30:53.376 helios[1]: INFO [AgentService STARTING] ServerConnector: >
Started admin@2bceb4df{HTTP/1.1}{0.0.0.0:5804}
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' hagent
172.17.0.12
```

Давайте снова вернемся в Zookeeper:

```
[zk: localhost:2181(CONNECTED) 4] ls /status/hosts
[5a4abcb27107]
[zk: localhost:2181(CONNECTED) 5] ls /status/hosts/5a4abcb27107
[agentinfo, jobs, environment, hostinfo, up]
[zk: localhost:2181(CONNECTED) 6] get /status/hosts/5a4abcb27107/agentinfo
{"inputArguments":["-Dcom.sun.management.jmxremote.port=9203", [...]
[...]
```

Здесь видно, что `/status/hosts` теперь содержит один элемент. Спуск в каталог хоста раскрывает внутреннюю информацию о хосте, которую хранит Helios.

> **ПРИМЕЧАНИЕ.** При работе на нескольких хостах вы должны передать `–name $(hostname -f)` в качестве аргумента и серверу Helios, и его агенту. Также необходимо открыть порты 5801 и 5802 для сервера и 5803 и 5804 для агента.

Давайте немного упростим взаимодействие с Helios:

```
$ alias helios="docker run -i --rm dockerinpractice/docker-helios \
helios -z http://172.17.0.11:5801"
```

Предыдущий псевдоним означает, что при вызове `helios` запускается одноразовый контейнер для выполнения желаемого действия, указывая на правильный кластер Helios, с которого нужно начать. Обратите внимание, что интерфейс командной строки должен указывать на сервер Helios, а не на Zookeeper.

Теперь все настроено. Мы можем легко взаимодействовать с нашим кластером Helios, поэтому пришло время опробовать один пример.

```
$ helios create -p nc=8080:8080 netcat:v1 ubuntu:14.04.2 -- \
sh -c 'echo hello | nc -l 8080'
Creating job: {"command":["sh","-c","echo hello | nc -l 8080"], >
"creatingUser":null,"env":{},"expires":null,"gracePeriod":null, >
"healthCheck":null,"id": >
"netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac", >
"image":"ubuntu:14.04.2","ports":{"nc":{"externalPort":8080, >
"internalPort":8080,"protocol":"tcp"}},"registration":{}, >
"registrationDomain":"","resources":null,"token":"","volumes":{}}
Done.
netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac
$ helios jobs
JOB ID              NAME     VERSION   HOSTS   COMMAND                         ENVIRONMENT
netcat:v1:2067d43   netcat   v1        0       sh -c "echo hello | nc -l 8080"
```

Helios построен на концепции *заданий* – все, нужно выполнить, должно быть выражено как задание, прежде чем его можно отправить на хост. Как минимум, вам нужен образ с основным содержимым, которое только должен знать для запуска контейнера: команда для выполнения и любые параметры порта, тома или среды. Также может потребоваться ряд других дополнительных параметров, включая проверку работоспособности, даты истечения срока действия и регистрацию служб.

Предыдущая команда создает задание, которое прослушает порт 8080, выведет слово «hello» первому, кто соединится с портом, а затем завершит работу.

Вы можете использовать команду `helios hosts` для вывода списка хостов, доступных для развертывания задания, а затем фактически выполнить развертывание с помощью команды `helios deploy`. Затем команда `helios status` показывает нам, что задание стартовало успешно:

```
$ helios hosts
HOST         STATUS        DEPLOYED RUNNING CPUS MEM  LOAD AVG MEM USAGE >
OS HELIOS DOCKER
5a4abcb27107.Up 19 minutes 0        0       4    7 gb 0.61     0.84      >
Linux 3.13.0-46-generic 0.8.2131.3.1 (1.15)
$ helios deploy netcat:v15a4abcb27107
```

```
Deploying Deployment{jobId=netcat:v1: >
2067d43fc2c6f004ea27d7bb7412aff502e3cdac, goal=START, deployerUser=null} >
on [5a4abcb27107]
5a4abcb27107: done
$ helios status
JOB ID              HOST         GOAL    STATE     CONTAINER ID  PORTS
netcat:v1:2067d43   5a4abcb27107.START   RUNNING   b1225bc       nc=8080:8080
```

Конечно, теперь мы хотим убедиться, что служба работает:

```
$ curl localhost:8080
hello
$ helios status
JOB ID              HOST         GOAL    STATE           CONTAINER ID  PORTS
netcat:v1:2067d43   5a4abcb27107.START   PULLING_IMAGE   b1225bc       nc=8080:8080
```

Результат `curl` ясно говорит нам, что служба работает, но `helios status` теперь показывает нечто интересное. При определении задания мы отметили, что после вывода «hello» задание завершит работу, но предыдущий вывод показывает состояние `PULLING_IMAGE`. Это связано с тем, как Helios управляет заданиями, – после развертывания на хосте Helios сделает все возможное, чтобы задание продолжало работу. Состояние, которое вы здесь видите, – это то, что Helios проходит полный процесс запуска задания, что предполагает гарантию извлечения образа.

Наконец, нам нужно прибраться за собой.

```
$ helios undeploy -a --yes netcat:v1
Undeploying netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac from >
[5a4abcb27107]
5a4abcb27107: done
$ helios remove --yes netcat:v1
Removing job netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac
netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac: done
```

Мы попросили удалить задание со всех узлов (завершив его работу при необходимости и остановив любые дальнейшие автоматические перезапуски), а затем удалили само задание, то есть его нельзя будет развернуть на других узлах.

ОБСУЖДЕНИЕ

Helios – это простой и надежный способ развертывания ваших контейнеров на нескольких хостах. В отличие от ряда методов, к которым мы придем позже, за кулисами не происходит никакой магии, чтобы определить подходящее место, – Helios запускает контейнеры именно там, где вы хотите, с минимальной суетой.

Но эта простота обходится дорого, как только вы переходите к более сложным сценариям развертывания – такие функции, как ограничения ресурсов, динамическое масштабирование и т. д. в настоящее время отсутствуют,

поэтому вы можете оказаться в ситуации когда нужно будет заново изобретать части таких инструментальных средств, как Kubernetes (метод 88), чтобы добиться нужного вам поведения при развертывании.

11.3. Обнаружение сервисов: что у нас здесь?

Во введении к этой главе обнаружение сервисов рассматривалось как обратная сторона оркестровки – возможность развертывания ваших приложений на сотнях разных компьютеров – это прекрасно, но, если вы потом не можете узнать, где какие приложения находятся, вы не сможете *использовать* их.

Хотя эта область не настолько насыщена, как оркестровка, у области обнаружения сервисов все же есть ряд конкурентов. То, что все они предлагают немного разные наборы функций, не срабатывает.

Существует две фрагмента функциональности, которые обычно желательны, когда дело доходит до обнаружения сервисов: общее хранилище типа «ключ/значение» и способ получения конечных точек служб через некий удобный интерфейс (вероятно, DNS). etcd и Zookeeper – примеры первого случая, тогда как SkyDNS (инструмент, который мы не будем использовать) является примером последнего. По факту SkyDNS использует etcd для хранения необходимой информации.

МЕТОД 85 Использование Consul для обнаружения сервисов

etcd – очень популярное инструментальное средство, но у него есть один конкретный конкурент, который часто упоминается вместе с ним: Consul. Это немного странно, потому что есть другие инструменты, более похожие на etcd (у Zookeeper имеется аналогичный набор функций для etcd, но реализованный на другом языке), в то время как Consul отличается некоторыми интересными дополнительными функциями, такими как обнаружение сервисов и проверка работоспособности. На самом деле если вы прищуритесь, то Consul может выглядеть как etcd, SkyDNS и Nagios в одном.

ПРОБЛЕМА

Вы должны иметь возможность распространять информацию для набора контейнеров, обнаруживать сервисы внутри него и осуществлять контроль.

РЕШЕНИЕ

Запустите контейнер с помощью Consul на каждом хосте Docker, чтобы предоставить каталог служб и систему связи конфигурации.

Consul пытается быть универсальным инструментальным средством для выполнения некоторых важных задач, необходимых, когда вам нужно координировать ряд независимых служб. Эти задачи могут выполняться другими инструментами, но их настройка в одном месте может быть полезной. На высоком уровне Consul обеспечивает следующее:

- *конфигурация служб* – хранилище типа «ключ/значение» для хранения и совместного использования небольших значений, таких как etcd и Zookeeper;
- *обнаружение служб* – API для регистрации служб и конечная точка DNS для их обнаружения, например SkyDNS;
- *контроль над службами* – API для регистрации проверок работоспособности, например Nagios.

Вы можете использовать все, некоторые или одну из этих функций, поскольку здесь нет привязки. Если у вас есть существующая инфраструктура наблюдения, не нужно менять ее на Consul.

В этом методе будет идти речь об аспектах обнаружения и контроля над службами Consul, но не о хранилище типа «ключ/значение». Сильное сходство между etcd и Consul в этом аспекте делает два последних метода в главе 9 (методы 74 и 75) переносимыми при внимательном прочтении документации к Consul.

На рис. 11.5 показана типичная настройка Consul.

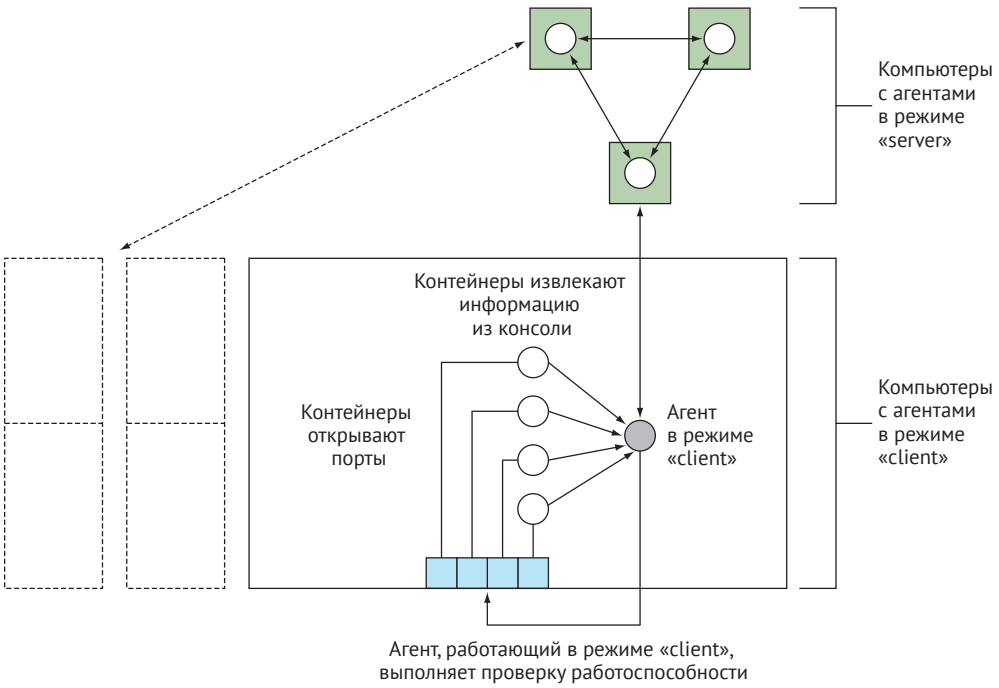


Рис. 11.5 ❖ Типичная настройка Consul

Ответственность за данные, хранящиеся в Consul, лежит на агентах, работающих в режиме *«server»*. Они ответственны за формирование *консенсуса* в отношении хранимой информации – эта концепция присутствует в большинстве распределенных систем хранения данных. Говоря кратко, если вы потеряете менее половины своих server-агентов, вы гарантированно сможете

восстановить свои данные (см. пример с использованием etcd в методе 74). Поскольку эти серверы так важны и требуют больших ресурсов, их хранение на выделенных компьютерах является типичным выбором.

> **ПРИМЕЧАНИЕ.** Хотя команды, приведенные в этом методе, оставят каталог данных Consul (/data) внутри контейнера, обычно рекомендуется указывать этот каталог как том, по крайней мере, для серверов, чтобы вы могли хранить резервные копии.

Рекомендуется, чтобы на всех компьютерах под вашим контролем, которые могут захотеть взаимодействовать с Consul, работал агент в режиме «client». Эти агенты перенаправляют запросы на серверы и запускают проверку работоспособности.

Первый шаг для запуска Consul – запустить агента в режиме «server»:

```
c1 $ IMG=dockerinpractice/consul-server
c1 $ docker pull $IMG
[...]
c1 $ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$\|vbox.*$'
    inet 192.168.1.87/24 brd 192.168.1.255 scope global wlan0
c1 $ EXTIP1=192.168.1.87
c1 $ echo '{"ports": {"dns": 53}}' > dns.json
c1 $ docker run -d --name consul --net host \
-v $(pwd)/dns.json:/config/dns.json $IMG -bind $EXTIP1 -client $EXTIP1 \
-recursor 8.8.8.8 -recursor 8.8.4.4 -bootstrap-expect 1
88d5cb48b8b1ef9ada754f97f024a9ba691279e1a863fa95fa196539555310c1
c1 $ docker logs consul
[...]
       Client Addr: 192.168.1.87 (HTTP: 8500, HTTPS: -1, DNS: 53, RPC: 8400)
      Cluster Addr: 192.168.1.87 (LAN: 8301, WAN: 8302)
[...]
==> Log data will now stream in as it occurs:

    2015/08/1412:35:41 [INFO] serf: EventMemberJoin: mylaptop 192.168.1.87
[...]
    2015/08/1412:35:43 [INFO] consul: member 'mylaptop' joined, marking >
health alive
    2015/08/1412:35:43 [INFO] agent: Synced service 'consul'
```

Поскольку мы хотим использовать Consul в качестве DNS-сервера, мы вставили файл в папку, из которой Consul считывает конфигурацию, чтобы запросить его прослушивание через порт 53 (зарегистрированный порт для протокола DNS). Затем мы использовали последовательность команд, которую вы можете распознать по более ранним методам, чтобы попытаться найти внешний IP-адрес компьютера для связи с другими агентами и для прослушивания запросов клиентов.

> **ПРИМЕЧАНИЕ.** IP-адрес `0.0.0.0` обычно используется, чтобы указать, что приложению следует прослушивать все доступные интерфейсы на компьютере. Мы сознательно не сделали этого, потому что в некоторых дистрибутивах Linux есть демон кеширования DNS, слушающий `127.0.0.1`, который запрещает прослушивание `0.0.0.0:53`.

В предыдущей команде `docker run` есть три момента, о которых стоит упомянуть:

- мы использовали `--net host`. Хотя в мире Docker это считают ошибкой, в качестве альтернативы можно открыть до восьми портов в командной строке – это вопрос личных предпочтений, но мы считаем, что в данном случае это оправдано. Это также помогает обойти потенциальную проблему при обмене данными по протоколу UDP. Если бы вы пошли по ручному маршруту, не было бы необходимости устанавливать DNS-порт – вы могли бы открыть DNS-порт по умолчанию (8600) как порт 53 на хосте;
- два аргумента `recursor` сообщают Consul, на какие DNS-серверы следует смотреть, если запрашиваемый адрес неизвестен самому Consul;
- Аргумент `-bootstrap-wait 1` означает, что кластер Consul начнет работать только с одним агентом, который не является устойчивым. При типичной настройке ставится 3 (или больше), чтобы кластер не запускался до тех пор, пока не подключится необходимое количество серверов. Чтобы запустить дополнительных агентов в режиме «server», добавьте аргумент `-join`. Мы обсудим это при запуске клиента.

Теперь давайте перейдем ко второму компьютеру, запустим агента в режиме «client» и добавим его в наш кластер.

> **ПРИМЕЧАНИЕ.** Поскольку Consul ожидает, что сможет прослушивать определенный набор портов при обмене данными с другими агентами, сложно настроить несколько агентов на одном компьютере, в то же время демонстрируя, как это будет работать в реальном мире. Теперь мы будем использовать другой хост – если вы решите использовать псевдоним IP, убедитесь, что вы передали `-node newAgent`, потому что по умолчанию будет использоваться имя хоста, что приведет к конфликту.

```
c2 $ IMG=dockerinpractice/consul-agent
c2 $ docker pull $IMG
[...]
c2 $ EXTIP1=192.168.1.87
c2 $ ip addr | grep docker0 | grep inet
    inet 172.17.42.1/16 scope global docker0
c2 $ BRIDGEIP=172.17.42.1
```

```
c2 $ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$'
    inet 192.168.1.80/24 brd 192.168.1.255 scope global wlan0
c2 $ EXTIP2=192.168.1.80
c2 $ echo '{"ports": {"dns": 53}}' > dns.json
c2 $ docker run -d --name consul-client --net host \
-v $(pwd)/dns.json:/config/dns.json $IMG -client $BRIDGEIP -bind $EXTIP2 \
-join $EXTIP1 -recursor 8.8.8.8 -recursor 8.8.4.4
5454029b139cd28e8500922d1167286f7e4fb4b7220985ac932f8fd5b1cdef25
c2 $ docker logs consul-client
[...]
    2015/08/14 19:40:20 [INFO] serf: EventMemberJoin: mylaptop2 192.168.1.80
[...]
    2015/08/14 13:24:37 [INFO] consul: adding server mylaptop >
(Addr: 192.168.1.87:8300) (DC: dc1)
```

> **ПРИМЕЧАНИЕ.** Образы, которые мы использовали, основаны на gliderlabs/consul-server:0.5 и gliderlabs/consul-agent:0.5 и поставляются с более новой версией Consul, чтобы избежать возможных проблем при обмене данными по протоколу UDP, о чем свидетельствует постоянная регистрация таких строк, как: «Опровержение подозрительного сообщения». После выхода версии образов 0.6 вы можете переключиться обратно на образы из gliderlabs.

Все клиентские службы (HTTP, DNS и т. д.) настроены на прослушивание IP-адреса моста Docker. Это дает контейнерам известное местоположение, из которого они могут получать информацию от Consul, и предоставляет только внутренний доступ к Consul на компьютере, заставляя другие компьютеры получать прямой доступ к агентам, работающим в режиме «server», а не идти по более медленному маршруту через агента в режиме «client» к server-агенту. Чтобы убедиться, что IP-адрес моста согласован для всех ваших хостов, можете посмотреть на аргумент `--bip` для демона Docker.

Как и раньше, мы нашли внешний IP-адрес и привязали к нему обмен данными между кластерами. Аргумент `-join` сообщает Consul, где сначала искать кластер. Не беспокойтесь о микроуправлении формированием кластера – когда два агента первоначально встречаются друг с другом, они *сплетничают*, передавая информацию о поиске других агентов в кластер. Последние аргументы `-recursor` сообщают Consul, какие вышестоящие DNS-серверы использовать для DNS-запросов, которые не пытаются искать зарегистрированные службы.

Давайте проверим, что агент подключился к серверу с помощью HTTP API на клиентском компьютере. API-вызов, используемый нами, вернет список членов, которые, по мнению client-агента, находятся в кластере. В больших быстро меняющихся кластерах это может не всегда соответствовать элементам кластера – для этого есть еще один (более медленный) API- вызов.

```
c2 $ curl -sSL $BRIDGEIP:8500/v1/agent/members | tr ',' '\n' | grep Name
[{"Name":"mylaptop2"
{"Name":"mylaptop"
```

Теперь, когда инфраструктура Consul настроена, пришло время посмотреть, как можно регистрировать и обнаруживать сервисы. Типичный процесс регистрации – заставить ваше приложение выполнить API-вызов для локального client-агента после инициализации, что побуждает этого агента распространять информацию агентам, работающим в режиме «server». В качестве демонстрации мы выполним шаг регистрации вручную.

```
c2 $ docker run -d --name files -p 8000:80 ubuntu:14.04.2 \
python3 -m http.server 80
96ee81148154a75bc5c8a83e3b3d11b73d738417974eed4e019b26027787e9d1
c2 $ docker inspect -f '{{.NetworkSettings.IPAddress}}' files
172.17.0.16
c2 $ /bin/echo -e 'GET / HTTP/1.0\r\n\r\n' | nc -i1172.17.0.16 80 \
| head -n 1
HTTP/1.0 200 OK
c2 $ curl -X PUT --data-binary '{"Name": "files", "Port": 8000}' \
$BRIDGEIP:8500/v1/agent/service/register
c2 $ docker logs consul-client | tail -n 1
    2015/08/15 03:44:30 [INFO] agent: Synced service 'files'
```

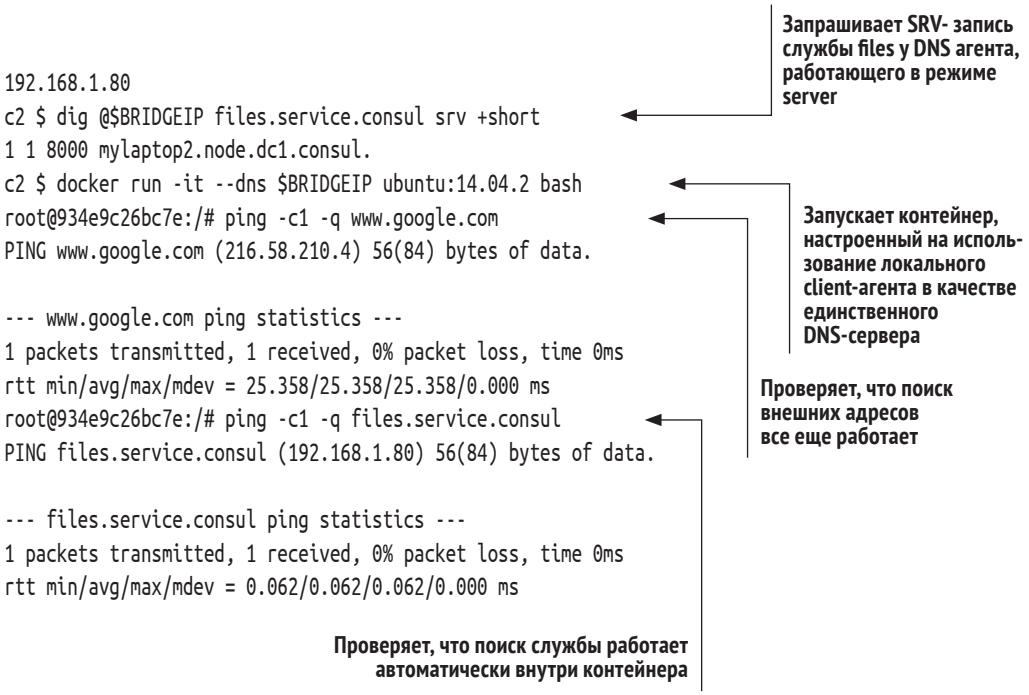
Здесь мы настроили простой HTTP-сервер в контейнере, предоставив его на порту 8000 на хосте, и проверили, работает ли он. Затем использовали curl и HTTP API Consul для регистрации определения сервиса. Единственное, что здесь абсолютно необходимо, это название службы – порт наряду с другими полями, перечисленными в документации к Consul – все это необязательно. Стоит упомянуть поле ID – по умолчанию используется имя службы, но оно должно быть уникальным для всех служб. Если вам нужно несколько экземпляров службы, необходимо будет указать его.

Строка журнала Consul сообщает нам, что служба синхронизирована, поэтому мы должны иметь возможность получать информацию о ней из интерфейса DNS. Эта информация поступает от агентов, работающих в режиме «server», поэтому она служит подтверждением того, что служба была принята в каталог Consul. Вы можете использовать команду `dig` для запроса информации о DNS и проверки ее наличия:

Ищет IP-адрес службы files из DNS агента, работающего в режиме «server». Эта DNS-служба доступна для случайных компьютеров, не входящих в кластер Consul, что позволяет им также извлечь выгоду из обнаружения сервисов

```
c2 $ EXTIP1=192.168.1.87
c2 $ dig @$EXTIP1 files.service.consul +short
192.168.1.80
c2 $ BRIDGEIP=172.17.42.1
c2 $ dig @$BRIDGEIP files.service.consul +short
```

Ищет IP-адрес службы files из DNS агента, работающего в режиме «client». Если не удается использовать $BRIDGEIP, можете попробовать использовать $EXTIP1



ПРИМЕЧАНИЕ. SRV-записи – это способ передачи служебной информации в DNS, включая протокол, порт и другие записи. В предыдущем случае вы можете увидеть номер порта в ответе, и вам дано каноническое хост-имя компьютера, предоставляющего сервис, а не IP-адрес.

У опытных пользователей может возникнуть желание избежать ручной установки аргумента `--dns`, настроив аргументы `--dns` и `--bip` для самого демона Docker, но не забудьте переопределить значения по умолчанию для агента Consul, иначе можете столкнуться с неожиданным поведением.

Интересно сходство между DNS-службой Consul и виртуальными сетями Docker в методе 80: оба позволяют обнаруживать контейнеры по удобочитаемому имени, а у Docker есть встроенная возможность выполнять эту работу на нескольких узлах с оверлейными сетями. Ключевое отличие состоит в том, что Consul существует за пределами Docker, и поэтому его может быть легче интегрировать в существующие системы.

Однако, как уже упоминалось в начале этого метода, у Consul есть еще одна интересная особенность, которую мы рассмотрим: проверка работоспособности.

Проверка работоспособности – обширная тема, поэтому оставим детали для подробной документации к Consul и рассмотрим один из вариантов мониторинга – проверку сценариев. Она запускает команду и устанавливает работоспособность на основе возвращаемого значения: 0 – успешно, 1 – внимание и любое другое значение для критических ситуаций. Вы можете

зарегистрировать проверку работоспособности при первоначальном определении службы или в отдельном API- вызове, как мы это сделаем здесь.

```
c2 $ cat >check <<'EOF'
#!/bin/sh
set -o errexit
set -o pipefail

SVC_ID="$1"
SVC_PORT=\
"$(wget -qO - 172.17.42.1:8500/v1/agent/services | jq ".$SVC_ID.Port")"
wget -qsO - "localhost:$SVC_PORT"
echo "Success!"
EOF
c2 $ cat check | docker exec -i consul-client sh -c \
'cat > /check && chmod +x /check'
c2 $ cat >health.json <<'EOF'
{
  "Name": "filescheck",
  "ServiceID": "files",
  "Script": "/check files",
  "Interval": "10s"
}
EOF
c2 $ curl -X PUT --data-binary @health.json \
172.17.42.1:8500/v1/agent/check/register
 c2 $ sleep 300
 c2 $ curl -sSL 172.17.42.1:8500/v1/health/service/files | \
python -m json.tool | head -n 13
 [
    {
        "Checks": [
                    {
                        "CheckID": "filescheck",
                        "Name": "filescheck",
                        "Node": "mylaptop2",
                        "Notes": "",
                        "Output": "/check: line 6: jq: not \
found\nConnecting to 172.17.42.1:8500 (172.17.42.1:8500)\n",
                        "ServiceID": "files",
                        "ServiceName": "files",
                        "Status": "critical"
                    },
c2 $ dig @$BRIDGEIP files.service.consul srv +short
 c2 $
```

Создает сценарий проверки, проверяющий, что код состояния HTTP от службы – «200 OK». Сервисный порт ищется из идентификатора сервиса, переданного сценарию в качестве аргумента

Копирует сценарий проверки в контейнер агента Consul

Создает определение проверки работоспособности для отправки в API HTTP Consul. Идентификатор службы должен быть указан как в поле ServiceID, так и в командной строке сценария

Отправляет файл проверки работоспособности в формате JSON агенту Consul

Ожидает, что выходные данные проверки будут переданы агентам, работающим в режиме server

Получает информацию о проверке работоспособности для проверки, которую вы зарегистрировали

Попытки службы files, без результатов

> **ПРИМЕЧАНИЕ.** Поскольку результаты проверки работоспособности могут меняться при каждом выполнении (например, при наличии временных меток), Consul синхронизирует результаты проверки с сервером только при изменении состояния или каждые пять минут (хотя этот интервал можно настроить). Поскольку статусы начинаются как критические, в этом случае не происходит первоначального изменения статуса, поэтому вам нужно будет переждать интервал, чтобы получить выходные данные.

Мы добавили проверку работоспособности файловой службы, которая будет запускаться каждые 10 секунд, но при проверке видно, что состояние службы критическое. Из-за этого Consul автоматически удалил отказавшую конечную точку из записей, возвращаемых DNS, оставив нас без серверов. Это особенно полезно для автоматического удаления серверов из службы с несколькими бэкэндами в рабочем окружении.

Основная причина ошибки, с который мы столкнулись, важна, и о ней необходимо знать при запуске Consul внутри контейнера. Все проверки также выполняются внутри контейнера, поэтому, поскольку сценарий проверки нужно было скопировать в контейнер, вам также важно убедиться, что все необходимые команды установлены в контейнере. В данном конкретном случае мы пропускаем команду `jq` (это полезная утилита для извлечения информации из JSON), которую можно установить вручную, хотя правильным подходом для эксплуатации будет добавление слоев к образу.

```
c2 $ docker exec consul-client sh -c 'apk update && apk add jq'
fetch http://dl-4.alpinelinux.org/alpine/v3.2/main/x86_64/APKINDEX.tar.gz
v3.2.3 [http://dl-4.alpinelinux.org/alpine/v3.2/main]
OK: 5289 distinct packages available
(1/1) Installing jq (1.4-r0)
Executing busybox-1.23.2-r0.trigger
OK: 14 MiB in 28 packages
c2 $ docker exec consul-client sh -c \
'wget -qO - 172.17.42.1:8500/v1/agent/services | jq ".files.Port"'
8000
c2 $ sleep 15
c2 $ curl -sSL 172.17.42.1:8500/v1/health/service/files | \
python -m json.tool | head -n 13
[
    {
        "Checks": [
            {
                "CheckID": "filescheck",
                "Name": "filescheck",
                "Node": "mylaptop2",
                "Notes": "",
```

```
        "Output": "Success!\n",
        "ServiceID": "files",
        "ServiceName": "files",
        "Status": "passing"
    },
```

Теперь мы установили `jq` на образ с помощью диспетчера пакетов Alpine Linux (см. метод 57), проверили ее работоспособность, вручную выполнив строку, которая ранее не работала в сценарии, а затем подождали, пока проверка не будет выполнена повторно. Сейчас все прошло успешно!

С проверками работоспособности сценария у вас теперь есть жизненно важный строительный блок для построения мониторинга вокруг приложения. Если вы можете выразить проверку работоспособности в виде серии команд, выполняемых в терминале, вы можете настроить так, чтобы Consul запускал это автоматически, – это не ограничивается статусом HTTP. Если захотите проверить код состояния, возвращаемый конечной точкой HTTP, вам повезло, так как это настолько распространенная задача, что этому посвящен один из трех типов проверки работоспособности в Consul, и вам не нужно использовать проверку работоспособности сценария (мы сделали это выше в иллюстративных целях).

Последний тип проверки работоспособности, время жизни, требует более глубокой интеграции с вашим приложением. Состояние должно периодически устанавливаться в положение «исправное», иначе проверка будет автоматически отключена. Сочетание этих трех типов проверки работоспособности дает вам возможность создать комплексный мониторинг поверх вашей системы.

В завершение этого метода мы рассмотрим дополнительный веб-интерфейс Consul, который поставляется с образом агента, работающего в режиме «server». Он предоставляет полезную информацию о текущем состоянии вашего кластера. Можете зайти на него, перейдя на порт 8500 с внешнего IP-адреса server-агента. В этом случае мы бы хотели зайти на `$EXTIP1:8500`. Помните, что даже если вы находитесь на хосте агента, `localhost` или `127.0.0.1` не будут работать.

ОБСУЖДЕНИЕ

Мы многое рассмотрели в этом методе, Consul – это обширная тема! К счастью, так же, как полученные вами знания об использовании хранилищ типа «ключ/значение» с etcd в методе 74 можно перенести в другие хранилища данного типа (например, Consul), знания об обнаружении сервисов можно передавать другим инструментальным средствам, предлагающим DNS-интерфейсы (SkyDNS – одно из тех, с которым вы можете столкнуться).

Рассмотренные нами тонкости, связанные с использованием стека сети хоста и внешних IP-адресов, также можно передавать. Большинство контейнеризированных распределенных инструментов, требующих обнаружения на нескольких узлах, могут иметь схожие проблемы, и о них стоит знать.

МЕТОД 86 Автоматическая регистрация служб с использованием Registrator

Очевидным недостатком Consul (и любого средства обнаружения служб) является необходимость управления созданием и удалением записей служб. Если вы интегрируете его в свои приложения, у вас будет несколько реализаций и несколько мест, где могут возникнуть проблемы.

Интеграция также не подходит для приложений, над которыми у вас нет полного контроля, поэтому придется писать сценарии-оболочки при запуске базы данных и тому подобное.

ПРОБЛЕМА

Вы не хотите вручную управлять записями служб и проверками работоспособности в Consul.

РЕШЕНИЕ

Используйте Registrator.

Этот метод будет построен поверх предыдущего и предполагает, что у вас есть кластер Consul, состоящий из двух частей, как было описано ранее. Мы также предполагаем, что в нем нет служб, поэтому вы, возможно, захотите воссоздать свои контейнеры, чтобы начать с нуля.

Registrator (http://gliderlabs.com/registrator/latest/) устраняет большую часть сложности управления службами Consul – он отслеживает запуск и остановку контейнеров, регистрируя службы на основе открытых портов и переменных среды контейнера. Самый простой способ увидеть его в действии – это войти внутрь.

Все, что мы делаем, будет реализовано на компьютере с агентом, работающим в режиме «client». Как обсуждалось ранее, никаких контейнеров, кроме агента в режиме «server», на другом компьютере быть не должно.

Следующие команды – все, что вам нужно для запуска Registrator:

```
$ IMG=gliderlabs/registrator:v6
$ docker pull $IMG
[...]
$ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$'
    inet 192.168.1.80/24 brd 192.168.1.255 scope global wlan0
$ EXTIP=192.168.1.80
$ ip addr | grep docker0 | grep inet
    inet 172.17.42.1/16 scope global docker0
$ BRIDGEIP=172.17.42.1
$ docker run -d --name registrator -h $(hostname)-reg \
-v /var/run/docker.sock:/tmp/docker.sock $IMG -ip $EXTIP -resync \
60 consul://$BRIDGEIP:8500 # if this fails, $EXTIP is an alternative
b3c8a04b9dfaf588e46a255ddf4e35f14a9d51199fc6f39d47340df31b019b90
$ docker logs registrator
2015/08/1420:05:57 Starting registrator v6 ...
```

```
2015/08/1420:05:57 Forcing host IP to 192.168.1.80
2015/08/1420:05:58 consul: current leader 192.168.1.87:8300
2015/08/1420:05:58 Using consul adapter: consul://172.17.42.1:8500
2015/08/1420:05:58 Listening for Docker events ...
2015/08/1420:05:58 Syncing services on 2 containers
2015/08/1420:05:58 ignored: b3c8a04b9dfa no published ports
2015/08/1420:05:58 ignored: a633e58c66b3 no published ports
```

Первые две команды для извлечения образа и поиска внешнего IP-адреса должны выглядеть знакомыми. Этот IP-адрес предоставляется Registrator, чтобы он знал, какой IP-адрес нужно объявить службам. Сокет Docker смонтирован, чтобы позволить Registrator получать автоматическое уведомление о запуске и остановке контейнеров по мере их появления. Мы также сообщили Registrator, как он может подключиться к агенту Consul, и что мы хотим, чтобы все контейнеры обновлялись каждые 60 секунд. Поскольку Registrator должен автоматически получать уведомления об изменениях контейнера, этот последний параметр полезен для смягчения воздействия возможного отсутствия обновлений.

Теперь, когда Registrator работает, зарегистрировать первую службу очень легко:

```
$ curl -sSL 172.17.42.1:8500/v1/catalog/services | python -m json.tool
{
    "consul": []
}
$ docker run -d -e "SERVICE_NAME=files" -p 8000:80 ubuntu:14.04.2 python3 \
-m http.server 80
3126a8668d7a058333d613f7995954f1919b314705589a9cd8b4e367d4092c9b
$ docker inspect 3126a8668d7a | grep 'Name.*/'
    "Name": "/evil_hopper",
$ curl -sSL 172.17.42.1:8500/v1/catalog/services | python -m json.tool
{
    "consul": [],
    "files": []
}
$ curl -sSL 172.17.42.1:8500/v1/catalog/service/files | python -m json.tool
[
    {
        "Address": "192.168.1.80",
        "Node": "mylaptop2",
        "ServiceAddress": "192.168.1.80",
        "ServiceID": "mylaptop2-reg:evil_hopper:80",
        "ServiceName": "files",
        "ServicePort": 8000,
        "ServiceTags": null
    }
]
```

Единственное усилие, которое нам пришлось приложить при регистрации службы, – передать переменную среды, чтобы сообщить Registrator, какое имя службы использовать. По умолчанию Registrator использует имя на основе компонента имени контейнера после косой черты и перед тегом: «mycorp.com/myteam/myimage:0.5» будет иметь имя «myimage». Полезно ли это или вы хотите указать что-то вручную, зависит от ваших соглашений об именах.

Остальные значения в значительной степени соответствуют вашим ожиданиям. Registrator обнаружил прослушиваемый порт, добавил его в Consul и установил идентификатор службы, который пытается дать подсказку о том, где можно найти контейнер (именно поэтому имя хоста было задано в контейнере Registrator).

ОБСУЖДЕНИЕ

Registrator отлично справляется с быстро меняющейся средой с высокой текучестью контейнеров, благодаря чему вам не нужно беспокоиться о создании проверок создания службы.

В дополнение к сведениям о службах `python -m json.tool | head -n 13` будет собирать информацию из разных сред, если она присутствует, включая теги, имена служб для каждого порта (если их несколько), и использовать проверку работоспособности (если вы используете Consul в качестве хранилища данных). Можно привести в действие все три типа проверок работоспособности Consul, указав подробности проверки в среде в формате JSON. Подробнее об этом см. в разделе Consul документации «Бэкэнды Registrator» по адресу https://gliderlabs.com/registrator/latest/user/backends/#consul или вернитесь к предыдущему методу, чтобы получить краткое представление о проверках работоспособности Consul.

Резюме

- Юниты systemd полезны для управления выполнением контейнера на одном компьютере.
- Зависимости могут быть выражены в юнитах systemd для обеспечения оркестровки при запуске.
- Helios – это качественное, простое решение для оркестровки с несколькими хостами.
- Consul может хранить информацию о ваших сервисах, что позволяет динамически обнаруживать их.
- Registrator может автоматически регистрировать контейнерные службы в Consul.

Глава 12

Центр обработки данных в качестве ОС с Docker

О чем рассказывается в этой главе:

- как использовать официальное решение Docker для оркестровки;
- различные способы использования Mesos для управления контейнерами Docker;
- два тяжеловеса в экосистеме оркестровки Docker: Kubernetes и OpenShift.

Если вы вернетесь к рис. 11.1 из предыдущей главы, то теперь мы продолжим движение вниз по ветвям дерева и перейдем к инструментам, которые убирают некоторые детали для повышения производительности. Большинство из них предназначены для более крупных развертываний на нескольких компьютерах, но нет никаких причин, по которым вы не можете использовать их на одном компьютере.

Что касается последней главы, то мы рекомендуем попытаться придумать сценарий для каждого инструментального средства, чтобы прояснить возможные варианты использования в вашей среде. В процессе работы мы продолжим приводить примеры в качестве отправных точек.

12.1. Мультихостовый Docker

Наилучший процесс перемещения контейнеров Docker на целевые компьютеры и их запуска – предмет многочисленных дискуссий в мире Docker. Ряд известных компаний создали свои собственные способы справиться с этим и сделали их доступными для всего мира. Вы можете извлечь огромную пользу из этого, если сможете решить, какие средства использовать.

Это быстро меняющаяся тема – мы видели рождение и смерть нескольких средств оркестровки для Docker и рекомендуем соблюдать осторожность при рассмотрении вопроса о том, стоит ли переходить на совершенно новый инструмент. В результате мы попытались выбрать инструменты со значительной стабильностью или движущей силой (или и то и другое).

МЕТОД 87 Бесшовный кластер Docker с режимом swarm

Здорово, когда у вас есть полный контроль над кластером, но иногда микроуправление не требуется. На самом деле, если у вас есть несколько приложений без сложных требований, вы можете в полной мере воспользоваться обещанием Docker о возможности работать где угодно – нет причины, по которой вы не должны бросать контейнеры в кластер и позволять кластеру решать, где их запустить.

Режим Swarm мог бы быть полезен для исследовательской лаборатории, если бы лаборатория могла разбить вычислительно сложную задачу на кусочки размером с укус. Это позволило бы им очень легко запустить свою проблему на кластере компьютеров.

ПРОБЛЕМА

У вас есть несколько хостов с Docker, и вы хотите иметь возможность запускать контейнеры без необходимости микроуправления там, где они будут работать.

РЕШЕНИЕ

Используйте режим swarm (от *англ*. swarm – «рой») для Docker, функцию, встроенную в Docker для решения задач оркестровки.

Режим swarm для Docker – это официальное решение от Docker Inc., позволяющее обрабатывать кластер хостов как единый демон Docker и развертывать на них службы. У него есть командная строка, очень похожая на ту, с которой вы знакомы по работе с командой `docker run`. Режим swarm возник из официального инструмента Docker, который вы использовали бы вместе с Docker, и был интегрирован в демон Docker. Если вы где-то видите старые ссылки на «Docker Swarm», они могут относиться к более старому инструменту.

Swarm состоит из нескольких узлов. Каждый узел может быть менеджером или рабочим узлом. Эти роли являются гибкими и могут быть изменены в любое время. Менеджер координирует развертывание служб на доступных узлах, тогда как рабочие узлы будут запускать только контейнеры. По умолчанию еще доступны менеджеры для запуска контейнеров, но вы также увидите, как это изменить.

Когда менеджер запускается, он инициализирует некое состояние для роя, а затем прослушивает входящие соединения от дополнительных узлов, чтобы добавить их к рою.

> **ПРИМЕЧАНИЕ.** Все версии Docker, используемые в рое, должны быть как минимум 1.12.0. В идеале необходимо попытаться сохранить все версии одинаковыми, иначе вы можете столкнуться с проблемами из-за несовместимости версий.

Сперва давайте создадим новый кластер (рой):

```
h1 $ ip addr show | grep 'inet ' | grep -v 'lo$\|docker0$' # get external IP
    inet 192.168.11.67/23 brd 192.168.11.255 scope global eth0
h1 $ docker swarm init --advertise-addr 192.168.11.67
Swarm initialized: current node (i5vtd3romfl9jg9g4bxtg0kis) is now a
manager.

To add a worker to this swarm, run the following command:

docker swarm join \
--token SWMTKN-1-4blo74l0m2bu5p8synq3w4239vxr1pyoa29cgkrjonx0tuid68
➥ -dhl9o1b62vrhhi0m817r6sxp2 \
192.168.11.67:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and
follow the instructions.
```

Мы создали новый рой и настроили демон Docker хоста `h1` в качестве менеджера.

Теперь вы можете его проверить:

```
h1 $ docker info
[...]
Swarm: active
 NodeID: i5vtd3romfl9jg9g4bxtg0kis
 Is Manager: true
 ClusterID: sg6sfmsa96nir1fbwcf939us1
 Managers: 1
 Nodes: 1
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
  Election Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
```

```
 Node Address: 192.168.11.67
 Manager Addresses:
  192.168.11.67:2377
[...]
h1 $ docker node ls
$ docker node ls
ID                            HOSTNAME    STATUS    AVAILABILITY    MANAGER STATUS
i5vtd3romfl9jg9g4bxtg0kis *   h1          Ready     Active          Leader
```

Затем создайте демон Docker на другом хосте в качестве рабочего узла, выполнив команду, указанную после запуска менеджера:

```
h2 $ docker swarm join \
    --token SWMTKN-1-4blo74l0m2bu5p8synq3w4239vxr1pyoa29cgkrjonx0tuid68
➥ -dhl9o1b62vrhhi0m817r6sxp2 \
    192.168.11.67:2377
This node joined a swarm as a worker.
```

h2 теперь добавлен в наш кластер в качестве рабочего узла. Выполнение команды `docker info` на любом из хостов покажет, что количество узлов возросло до 2, и `docker node ls` выведет список обоих узлов.

Наконец, давайте запустим контейнер. В режиме swarm это называется развертыванием службы, поскольку существуют дополнительные функции, которые не имеют смысла для контейнера. Перед развертыванием службы пометим менеджера как имеющего доступность типа `drain` – по умолчанию для запуска контейнеров доступны все менеджеры, но в этом методе мы хотим продемонстрировать возможности планирования на удаленном компьютере, поэтому введем ограничения, чтобы скрыться от менеджера. `rain` приведет к тому, что любые контейнеры, которые уже находятся на узле, будут повторно развернуты в другом месте, и на нем не будет запланировано никаких новых служб.

```
h1 $ docker node update --availability drain i5vtd3romfl9jg9g4bxtg0kis
h1 $ docker service create --name server -d -p 8000:8000 ubuntu:14.04 \
       python3 -m http.server 8000
vp0fj8p9khzh72eheoye0y4bn
h1 $ docker service ls
ID              NAME      MODE          REPLICAS    IMAGE           PORTS
vp0fj8p9khzh    server    replicated    1/1         ubuntu:14.04    *:8000->8000/tcp
```

Здесь есть несколько моментов, на которые стоит обратить внимание. Наиболее важным является то, что рой автоматически выбрал компьютер для запуска контейнера, – если бы у вас было несколько рабочих узлов, менеджер выбрал бы один на основе балансировки нагрузки. Вероятно, некоторые из аргументов для `docker service create` покажутся вам знакомыми по работе с командой `docker run`, – количество аргументов является общим, но стоит прочитать документацию. Например, аргумент `--volume` для `docker run` имеет другой формат в аргументе `--mount`, документацию для которого вы должны прочитать.

Пришло время проверить, работает ли наша служба:

```
h1 $ docker service ps server
ID            NAME      IMAGE         NODE  DESIRED STATE  CURRENT STATE
➥ ERROR PORTS
mixc9w3frple  server.1  ubuntu:14.04  h2    Running        Running 4
minutes ago
h1 $ docker node inspect --pretty h2 | grep Addr
Address:                192.168.11.50
h1 $ curl -sSL 192.168.11.50:8000 | head -n4
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
➥ "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ascii">
```

В режиме swarm есть часть дополнительных функций, включенных по умолчанию, которые называются *сеткой маршрутизации*. Она позволяет каждому узлу в рое выглядеть так, как будто он может обслуживать запросы для всех служб в рое, у которых есть опубликованные порты, – все входящие соединения перенаправляются на соответствующий узел.

Например, если вы снова вернетесь на узел менеджера `h1` (который, как мы знаем, не запускает службу, поскольку имеет доступность типа `drain`), он все равно будет отвечать на любые запросы на порту 8000:

```
h1 $ curl -sSL localhost:8000 | head -n4
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
➥ "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ascii">
```

Это может быть особенно полезно для простого вида обнаружения службы – если вы знаете адрес одного узла, можете очень легко получить доступ ко всем вашим службам.

После того как вы покончили с роем, можете закрыть все службы и удалить кластер.

```
$ docker service rm server
server
$ docker swarm leave
Error response from daemon: You are attempting to leave the swarm on a >
node that is participating as a manager. Removing the last manager erases >
all current state of the swarm. Use `--force` to ignore this message.
$ docker swarm leave --force
Node left the swarm.
```

Как видно здесь, режим swarm предупредит, если вы закроете последний менеджер в узле, потому что вся информация о рое будет потеряна. Вы можете переопределить это предупреждение с помощью `--force`. Также нужно будет выполнить команду `docker swarm leave` на всех рабочих узлах.

ОБСУЖДЕНИЕ

Это было краткое введение в режим swarm в Docker, и многие вещи здесь не были рассмотрены. Например, вы, возможно, заметили, что в тексте справки, после того как мы инициализировали рой, упоминается возможность подключения к рою дополнительных ведущих устройств – это полезно для устойчивости. Дополнительные объекты интереса – встроенные фрагменты функциональных возможностей, которые хранят информацию о конфигурации службы (как вы делали это с etcd в методе 74), используя ограничения для управления размещением контейнеров, и информацию о том, как обновить контейнеры с помощью откатов при сбое. Мы рекомендуем вам обратиться к официальной документации по адресу https://docs.docker.com/engine/swarm/ для получения дополнительной информации.

МЕТОД 88 Использование кластера Kubernetes

Вы уже были свидетелями двух крайностей в подходах к оркестровке: консервативный подход с использованием Helios и гораздо более свободный подход с использованием Docker Swarm. Но некоторые пользователи и компании ожидают, что их инструментальные средства будут несколько сложнее.

Потребность в настраиваемой оркестровке можно удовлетворить множеством вариантов, но есть те, что используются и обсуждаются чаще, чем другие. В одном случае это, несомненно, частично связано с стоящей за этим компанией, но можно надеяться, что Google знает, как создавать программное обеспечение для оркестровки.

ПРОБЛЕМА

Вы хотите управлять службами Docker на хостах.

РЕШЕНИЕ

Используйте Kubernetes и его мощные абстракции для управления вашим парком контейнеров.

Kubernetes – это инструментальное средство, созданное Google. Оно предназначено для компаний, которые предпочитают иметь четкие рекомендации и передовые практики по организации приложений и отношений между ними. Kubernetes использует специально разработанные средства для управления динамической инфраструктурой на основе указанной структуры.

Прежде чем мы перейдем к Kubernetes, давайте кратко рассмотрим архитектуру высокого уровня Kubernetes, изображенную на рис. 12.1.

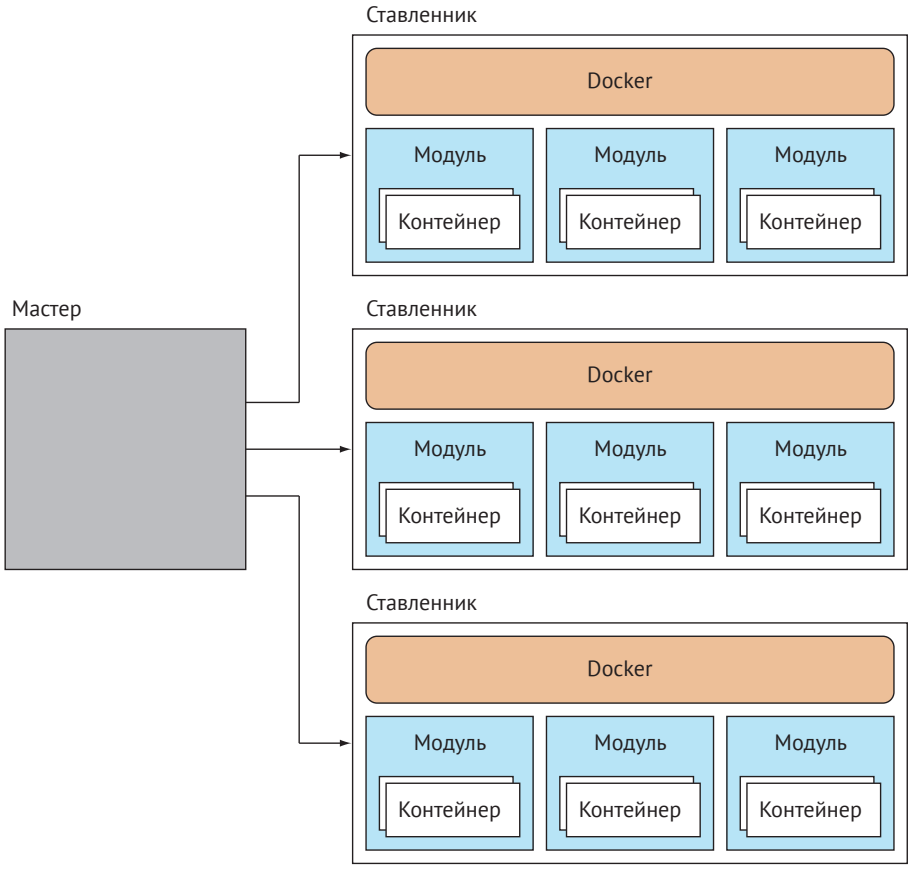


Рис. 12.1 ❖ Высокоуровневый обзор Kubernetes

Kubernetes имеет архитектуру master-minion. Главные узлы отвечают за получение приказов о том, что следует запускать в кластере, и оркестровку его ресурсов. На каждом «ставленнике» (minion) установлен Docker, а также служба *kubelet*, которая управляет модулями (наборами контейнеров), запущенными на каждом узле. Информация о кластере хранится в etcd, распределенном хранилище данных типа «ключ/значение» (см. метод 74), и это – источник истины кластера.

> **ПОДСКАЗКА.** Мы рассмотрим это позже в данном методе, поэтому не стоит слишком беспокоиться об этом сейчас, но *модуль* – это группа связанных контейнеров. Такая концепция существует для облегчения управления контейнерами Docker и их обслуживания.

Конечная цель Kubernetes – сделать так, чтобы ваши контейнеры работали в масштабе, просто объявив, что вы хотите, и предоставив Kubernetes возможность обеспечить соответствие кластера вашим потребностям. В этом методе

вы увидите, как масштабировать простую службу до заданного размера, выполнив одну команду.

> **ПРИМЕЧАНИЕ.** Kubernetes был первоначально разработан Google как средство управления контейнерами в масштабе. Google запускает контейнеры в масштабе на протяжение более десяти лет и решил разработать эту систему оркестровки контейнеров, когда Docker стал популярным. Kubernetes опирается на уроки, извлеченные из обширного опыта Google. Он также известен как «K8s».

Полное рассмотрение установки, настройки и функций Kubernetes – большая и быстро меняющаяся тема, которая выходит за рамки этой книги (и, без сомнения, как и сама по себе книга в скором времени). Здесь мы сосредоточимся на основных концепциях Kubernetes и создадим простой сервис, чтобы вы смогли прочувствовать его.

Установка Kubernetes

Вы можете либо установить Kubernetes непосредственно на своем хосте через Minikube, в результате чего у вас будет кластер с одним ставленником, либо использовать Vagrant для установки кластера с несколькими ставленниками, управляемым с помощью виртуальных машин. В этом методе мы сосредоточимся на первом варианте – последний лучше всего достигается с помощью исследований, чтобы определить правильный вариант для последней версии Kubernetes.

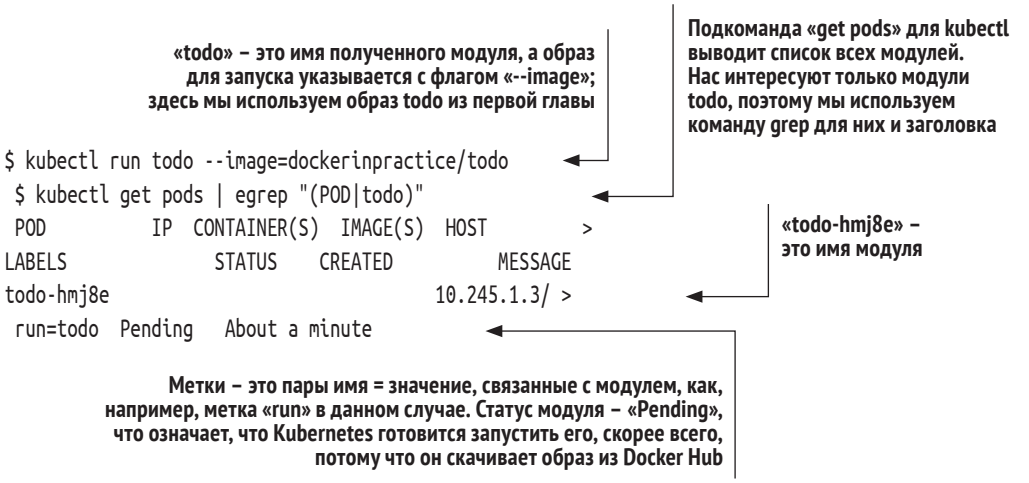
Рекомендуемый подход для локального начала работы с Kubernetes – установить кластер с одним ставленником на вашем хосте, следуя официальной документации для Minikube по адресу https://kubernetes.io/docs/tasks/tools/install-minikube/.

Minikube – это специализированный инструмент из проекта Kubernetes, созданный для облегчения процесса локальной разработки, но в настоящее время он слегка ограничен. Если вы хотите немного напрячься, рекомендуем поискать руководство по настройке многоузлового кластера Kubernetes с помощью Vagrant – этот процесс обычно меняется в зависимости от версии Kubernetes, поэтому мы не будем здесь давать конкретные рекомендации (хотя на момент написания этих строк мы посчитали https://github.com/Yolean/kubeadm-vagrant разумной отправной точкой).

После того как вы установили Kubernetes, можно идти дальше. Приведенный ниже вывод будет основан на многоузловом кластере. Мы начнем с создания одного контейнера и использования Kubernetes для его масштабирования.

Масштабирование одного контейнера

`Kubectl` – команда, используемая для управления Kubernetes. В этом случае вы будете использовать подкоманду `run` для запуска данного образа в качестве контейнера внутри модуля.



Kubernetes выбирает имя модуля, взяв его из команды `run` (`todo` в предыдущем примере), добавляя тире и случайную строку. Это гарантирует, что он не конфликтует с именами других модулей.

После того как вы подождете несколько минут, пока скачается образ todo, вы в конечном итоге увидите, что его статус изменился на «Running»:

```
$ kubectl get pods | egrep "(POD|todo)"
POD         IP          CONTAINER(S)   IMAGE(S)                >
HOST                    LABELS              STATUS       CREATED      MESSAGE
todo-hmj8e 10.246.1.3                                           >
10.245.1.3/10.245.1.3  run=todo   Running  4 minutes
                        todo            dockerinpractice/todo   >
                                        Running      About a minute
```

На этот раз столбцы IP, CONTAINER (S) и IMAGE (S) будут заполнены. Столбец IP содержит адрес модуля (в данном случае 10.246.1.3), а столбец контейнера имеет по одной строке на контейнер в модуле (в этом случае у нас только один контейнер, `todo`).

Вы можете проверить, что контейнер (todo) действительно работает и обслуживает запросы, напрямую обращаясь к IP-адресу и порту:

```
$ wget -qO- 10.246.1.3:8000
<html manifest="/todo.appcache">
[...]
```

На данный момент мы не видим большой разницы от непосредственного запуска контейнера Docker. Чтобы получить свой первый опыт работы с Kubernetes, вы можете расширить эту службу, выполнив команду `resize`:

```
$ kubectl resize --replicas=3 replicationController todo
resized
```

Эта команда сообщает Kubernetes, что вы хотите, чтобы контроллер репликации todo гарантировал, что в кластере запущено три экземпляра приложения todo.

> **ПОДСКАЗКА.** Контроллер репликации – это служба Kubernetes, которая гарантирует, что в кластере выполняется нужное количество модулей.

Вы можете проверить, что дополнительные экземпляры приложения todo были запущены с помощью команды `kubectl get pods`:

```
$ kubectl get pods | egrep "(POD|todo)"
POD         IP           CONTAINER(S)   IMAGE(S)                 >
HOST                     LABELS              STATUS       CREATED        MESSAGE
todo-2ip3n  10.246.2.2                                            >
10.245.1.4/10.245.1.4    run=todo  Running   10 minutes
                          todo           dockerinpractice/todo     >
                                             Running      8 minutes
todo-4os5b 10.246.1.3                                             >
10.245.1.3/10.245.1.3    run=todo  Running   2 minutes
                          todo           dockerinpractice/todo     >
                                             Running      48 seconds
todo-cuggp 10.246.2.3                                             >
10.245.1.4/10.245.1.4    run=todo  Running   2 minutes
                          todo           dockerinpractice/todo     >
```

Kubernetes принял инструкцию `resize` и контроллер репликации todo и обеспечил запуск нужного количества модулей. Обратите внимание, что он разместил два модуля на одном хосте (10.245.1.4) и один на другом (10.245.1.3). Это связано с тем, что в планировщике по умолчанию в Kubernetes есть алгоритм, который, как принято изначально, распределяет модули по узлам.

> **ПОДСКАЗКА.** Планировщик – это часть программного обеспечения, которая решает, где и когда должны выполняться элементы работы. Например, в ядре Linux есть планировщик, решающий, какую задачу следует выполнить дальше. Планировщики варьируются от крайне простых до невероятно сложных.

Вы начали понимать, как Kubernetes может упростить управление контейнерами на нескольких хостах. Далее мы подробно изучим основную концепцию модулей Kubernetes.

Использование модулей

Модуль – это набор контейнеров, предназначенных для того, чтобы работать вместе тем или иным образом и совместно использовать ресурсы.

Каждый модуль получает свой собственный IP-адрес и использует одни и те же тома и диапазон сетевых портов. Поскольку контейнеры модуля совместно используют локальный хост, контейнеры могут полагаться на различные службы, доступные и видимые везде, где они развернуты.

На рис. 12.2 проиллюстрировано это с помощью двух контейнеров, которые совместно используют том. На этом рисунке контейнер 1 может быть веб-сервером, считывающим файлы данных из общего тома, который, в свою очередь, обновляется контейнером 2. Таким образом, оба контейнера не сохраняют состояния; состояние хранится в общем томе.

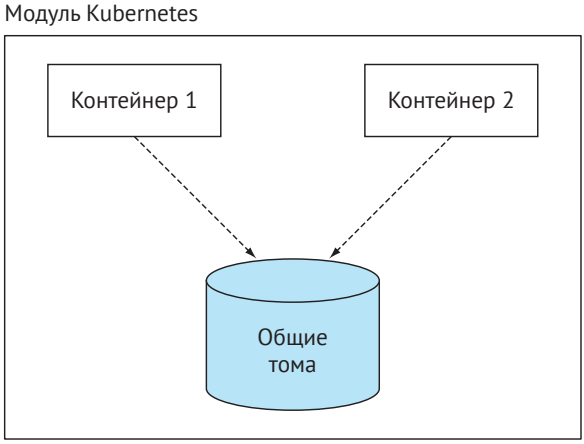


Рис. 12.2 ❖ Модуль с двумя контейнерами

Такая структура разделения обязанностей облегчает подход к микросервисам, позволяя вам управлять каждой частью вашего сервиса отдельно. Можете обновить один контейнер в модуле, не заботясь о других.

Следующая спецификация модуля определяет сложный модуль с одним контейнером, который записывает случайные данные (`simplewriter`) в файл каждые 5 секунд, и другим контейнером, который выполняет чтение из того же файла. Файл используется совместно через том (`pod-disk`).

Листинг 12.1. complexpod.json

```
{
  "id": "complexpod",
  "kind": "Pod",
  "apiVersion": "v1beta1",
  "desiredState": {
    "manifest": {
      "version": "v1beta1",
      "id": "complexpod",
      "containers": [{
```

Дает сущности имя

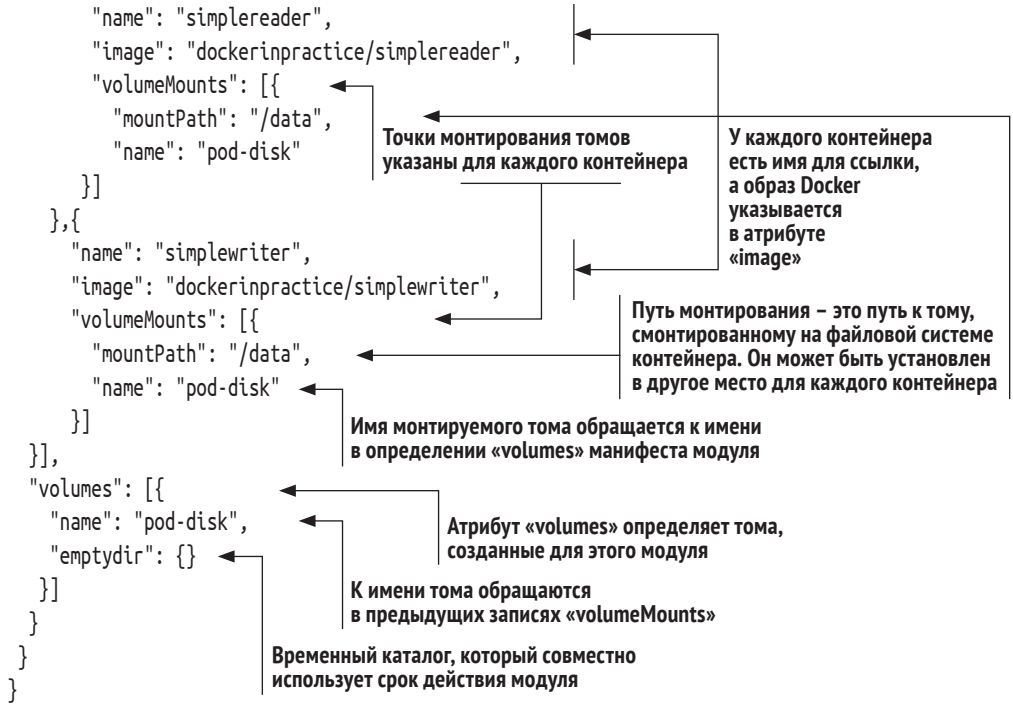
Определяет тип объекта

Суть спецификации модуля находится в атрибутах «requiredState» и «manifest»

Указывает Kubernetes версию, на которую ориентирован файл JSON

Дает сущности имя

Детали контейнеров в модуле хранятся в этом массиве в формате JSON



Чтобы загрузить эту спецификацию модуля, создайте файл с предыдущим листингом и выполните следующую команду:

```
$ kubectl create -f complexpod.json
pods/complexpod
```

Подождав минуту, пока скачается образ, вы увидите вывод журнала контейнера, выполнив команду `kubectl log` и указав сначала модуль, а затем интересующий вас контейнер.

```
$ kubectl log complexpod simplereader
2015-08-04T21:03:36.535014550Z '? U
[2015-08-04T21:03:41.537370907Z] h(^3eSk4y
[2015-08-04T21:03:41.537370907Z] CM(@
[2015-08-04T21:03:46.542871125Z] qm>5
[2015-08-04T21:03:46.542871125Z] {Vv_
[2015-08-04T21:03:51.552111956Z] KH+74 f
[2015-08-04T21:03:56.556372427Z] j?p+!\
```

ОБСУЖДЕНИЕ

Здесь мы лишь коснулись поверхностных возможностей и потенциала Kubernetes, но это должно дать вам представление о том, что можно сделать с его помощью и как это может упростить оркестровку контейнеров Docker.

Следующий метод рассматривает непосредственное использование некоторых дополнительных возможностей Kubernetes. Kubernetes также используется за кулисами в качестве движка оркестровки OpenShift в методах 90 и 99.

МЕТОД 89 Доступ к API Kubernetes из модуля

Часто модули могут работать совершенно независимо друг от друга, даже не зная, что они работают как часть кластера Kubernetes. Но Kubernetes все же предоставляет богатый API, и обеспечение контейнерам доступа к нему открывает дверь для самоанализа и адаптивного поведения, а также способность контейнеров самостоятельно управлять кластером Kubernetes.

ПРОБЛЕМА

Вы хотите получить доступ к API Kubernetes из модуля.

РЕШЕНИЕ

Выполните команду `curl` для доступа к API Kubernetes из контейнера в модуле, используя информацию об авторизации, доступную для контейнера.

Это один из самых коротких приемов в книге, но в нем много чего нужно распаковать. Это одна из причин, по которой данный метод полезен для изучения. Помимо прочего, мы рассмотрим:

- команду `kubectl`;
- запуск модулей Kubernetes;
- получение доступа к модулям Kubernetes;
- анти-шаблон Kubernetes;
- токены на предъявителя;
- секреты Kubernetes;
- «нисходящий API» Kubernetes.

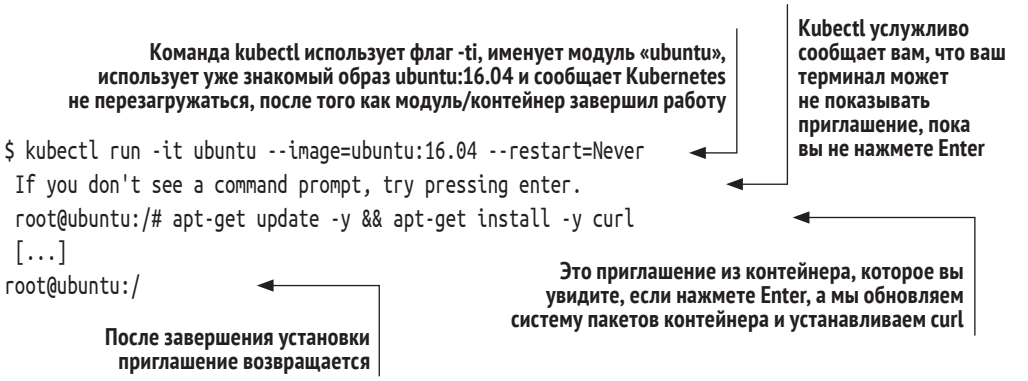
Если у вас не имеется доступа к кластеру Kubernetes, есть несколько вариантов. Существует много облачных провайдеров, которые предлагают платные кластеры Kubernetes. Для наименьшего числа зависимостей тем не менее мы рекомендуем использовать Minikube (упомянутый в последнем методе), который не требует наличия кредитной карты.

Для получения информации об установке Minikube см. документацию на странице https://kubernetes.io/docs/tasks/tools/install-minikube/.

Создание модуля

Сначала вы создадите контейнер в новом модуле `ubuntu` с помощью команды `kubectl`, а затем получите доступ к оболочке в этом контейнере в командной строке. (`kubectl run` в настоящее время устанавливает между модулями и контейнерами соотношение 1-1, хотя модули более гибкие, чем это в целом.)

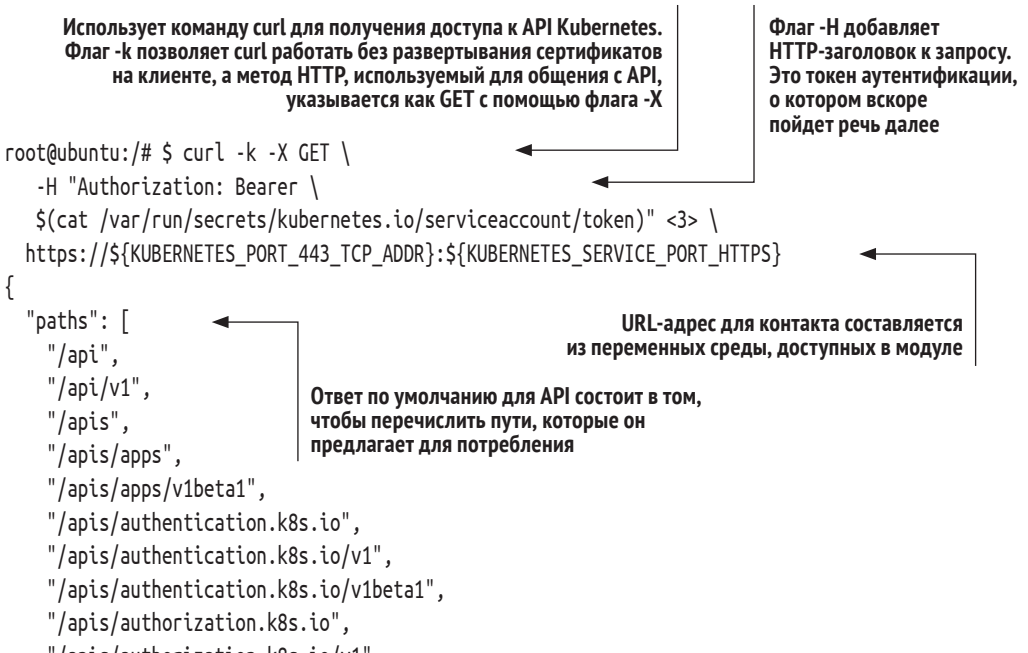
Листинг 12.2. Создание и настройка контейнера



Теперь вы находитесь в контейнере, созданном командой `kubectl`, и можете убедиться, что curl установлена.

> **ВНИМАНИЕ.** Получение доступа к модулю и его изменение из оболочки считается анти-шаблоном Kubernetes. Мы используем его здесь, чтобы продемонстрировать, что можно сделать изнутри модуля, а не то, как использовать модули.

Листинг 12.3. Доступ к API Kubernetes из модуля



```
    "/apis/authorization.k8s.io/v1beta1",
    "/apis/autoscaling",
    "/apis/autoscaling/v1",
    "/apis/autoscaling/v2alpha1",
    "/apis/batch",
    "/apis/batch/v1",
    "/apis/batch/v2alpha1",
    "/apis/certificates.k8s.io",
    "/apis/certificates.k8s.io/v1beta1",
    "/apis/extensions",
    "/apis/extensions/v1beta1",
    "/apis/policy",
    "/apis/policy/v1beta1",
    "/apis/rbac.authorization.k8s.io",
    "/apis/rbac.authorization.k8s.io/v1alpha1",
    "/apis/rbac.authorization.k8s.io/v1beta1",
    "/apis/settings.k8s.io",
    "/apis/settings.k8s.io/v1alpha1",
    "/apis/storage.k8s.io",
    "/apis/storage.k8s.io/v1",
    "/apis/storage.k8s.io/v1beta1",
    "/healthz",
    "/healthz/ping",
    "/healthz/poststarthook/bootstrap-controller",
    "/healthz/poststarthook/ca-registration",
    "/healthz/poststarthook/extensions/third-party-resources",
    "/logs",
    "/metrics",
    "/swaggerapi/",
    "/ui/",
    "/version"
  ]
}
root@ubuntu:/# curl -k -X GET -H "Authorization: Bearer $(cat
➥ /var/run/secrets/kubernetes.io/serviceaccount/token)"
➥ https://${KUBERNETES_PORT_443_TCP_ADDR}:
➥ ${KUBERNETES_SERVICE_ORT_HTTPS}/version
{
  "major": "1",
  "minor": "6",
  "gitVersion": "v1.6.4",
  "gitCommit": "d6f433224538d4f9ca2f7ae19b252e6fcb66a3ae",
  "gitTreeState": "dirty",
  "buildDate": "2017-06-22T04:31:09Z",
  "goVersion": "go1.7.5",
```

Сделан еще один запрос, на этот раз к пути /version

В ответе на запрос /version указывается работающая версия Kubernetes

```
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

В предыдущем листинге приводится много нового материала, но мы надеемся, что он даст представление о том, что можно делать в модулях Kubernetes динамически, без какой-либо настройки.

Ключевым моментом, который следует извлечь из этого листинга, является то, что информация предоставляется пользователям внутри модуля, что позволяет модулю связываться с API Kubernetes. Эти элементы информации вместе называются «нисходящим API». В настоящее время нисходящий API состоит из двух классов данных: переменных среды и файлов, доступных для модуля.

В предыдущем примере используется файл для предоставления API Kubernetes токена аутентификации. Этот токен доступен в файле /var/run/secrets/kubernetes. io/serviceaccount/token. В листинге 12.3 этот файл запускается с помощью команды `cat`, а вывод команды предоставляется как часть HTTP-заголовка `Authorization:`. Этот заголовок указывает, что используемая авторизация имеет тип `Bearer`, а токен на предъявителя – это вывод `cat`, поэтому аргумент `-H` для `curl` выглядит так:

```
-H "Authorization: Bearer
➥ $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)"
```

> **ПРИМЕЧАНИЕ.** *Токены на предъявителя* – это метод аутентификации, который требует, только чтобы был предоставлен указанный токен; идентификация не требуется (например, имя пользователя/пароль). *Акции на предъявителя* действуют по аналогичному принципу, когда предъявителем акций является тот, кто имеет право их продавать. Наличные деньги работают точно так же – действительно, в британских фунтах на банкнотах есть фраза «Обязуюсь выплатить предъявителю по требованию сумму в размере...»

Открытые элементы нисходящего API – это форма «секрета» Kubernetes. Любой секрет может быть создан с помощью API Kubernetes и открыт через файл в модуле. Этот механизм позволяет отделить секреты от образов Docker и модуля Kubernetes или конфигурации развертывания, что означает, что права доступа могут обрабатываться отдельно от этих более открытых элементов.

ОБСУЖДЕНИЕ

Стоит обратить внимание на этот метод, поскольку он охватывает много вопросов. Ключевым моментом для понимания является то, что в модулях Kubernetes доступна информация, которая позволяет им взаимодействовать с API Kubernetes. Это позволяет приложениям работать в Kubernetes, которые отслеживают и выполняют действия, происходящие вокруг кластера. Например, у вас может быть модуль инфраструктуры, который наблюдает в API

за недавно появившимися модулями, исследует их действия и записывает эти данные в другом месте.

Хотя управление доступом на основе ролей выходит за рамки книги, стоит упомянуть, что это имеет значение для безопасности, поскольку необязательно, чтобы какой-либо пользователь в вашем кластере имел такой уровень доступа. Следовательно, некоторым фрагментам API потребуется больше, чем просто токен на предъявителя, чтобы получить доступ.

Эти соображения, имеющие отношение к безопасности, делают данный метод наполовину связанным с Kubernetes и наполовину с безопасностью. В любом случае, метод важен для тех, кто хочет использовать Kubernetes «по-настоящему», чтобы помочь им понять, как работает API и как им можно злоупотреблять.

МЕТОД 90 Использование OpenShift для локального запуска API-интерфейсов AWS

Одной из больших проблем с локальной разработкой является тестирование приложения с другими сервисами. Docker поможет в этом, если сервис может быть помещен в контейнер, но это оставляет большой мир внешних сторонних сервисов нерешенным.

Распространенным решением является создание тестовых экземпляров API, но они часто дают ложные ответы – более полный тест функциональности невозможен, если приложение строится вокруг службы. Например, представьте, что вы хотите использовать AWS S3 в качестве места загрузки для своего приложения, где оно затем обрабатывает загрузки, – такое тестирование будет стоить денег.

ПРОБЛЕМА

Вы хотите, чтобы AWS-подобные API-интерфейсы были доступны локально для разработки.

РЕШЕНИЕ

Установите LocalStack и используйте доступные сервисные эквиваленты AWS.

В этом пошаговом руководстве вы настроите систему OpenShift с помощью Minishift, а затем запустите LocalStack в модуле в ней. OpenShift – это спонсируемая RedHat оболочка для Kubernetes, которая предоставляет дополнительную функциональность, более подходящую для развертывания Kubernetes для эксплуатации в производственной среде.

В этом методе мы рассмотрим:

- создание маршрутов в OpenShift;
- ограничения контекста безопасности;
- различия между OpenShift и Kubernetes;
- тестирование сервисов AWS с использованием общедоступных образов Docker.

ПРИМЕЧАНИЕ. Чтобы следовать этому методу, вам нужно установить Minishift. Minishift похож на Minikube, который вы видели в методе 89. Разница состоит в том, что он содержит установку OpenShift (подробно описанную в методе 99).

LocalStack

LocalStack – это проект, цель которого – предоставить вам как можно более полный набор API-интерфейсов AWS для разработки без каких-либо затрат. Он отлично подходит для тестирования или опробования кода перед тем, как запускать его по-настоящему в AWS и потенциально тратить время и деньги.

LocalStack запускает следующие основные облачные API на вашем локальном компьютере:

- API- Gateway на http://localhost:4567;
- Kinesis на http://localhost:4568;
- DynamoDB на http://localhost:4569;
- DynamoDB Streams на http://localhost:4570;
- Elasticsearch на http://localhost:4571;
- S3 на http://localhost:457;
- Firehose на http://localhost:4573;
- Lambda в http://localhost:4574;
- SNS на http://localhost:4575;
- SQS на http://localhost:4576;
- Redshift на http://localhost:4577;
- ES (Elasticsearch Service) на http://localhost:4578;
- SES на http://localhost:4579;
- Route53 на http://localhost:4580;
- CloudFormation на http://localhost:4581;
- CloudWatch на http://localhost:4582.

LocalStack поддерживает запуск в контейнере Docker или непосредственно на компьютере. Он создан на основе Moto, который, в свою очередь, является фреймворком для мокирования, построенным на Boto, комплекте средств разработки (SDK) для Python, предназначенного для работы с сервисами AWS.

Работа в кластере OpenShift дает возможность запускать многие из этих сред API AWS. Затем вы можете создавать отдельные конечные точки для каждого набора служб и изолировать их друг от друга. Кроме того, можно меньше беспокоиться об использовании ресурсов, так как позаботится об этом планировщик кластера. Но LocalStack не запускается из коробки, поэтому мы покажем вам, что нужно сделать, чтобы заставить его работать.

Убеждаемся, что Minishift настроен

На данный момент мы предполагаем, что вы настроили Minishift, – вам следует ознакомиться с официальной документацией для начала работы по адресу https://docs.openshift.org/latest/minishift/getting-started/index.html.

Листинг 12.4. Проверяем, что Minishift настроен нормально

```
$ eval $(minishift oc-env)
$ oc get all
No resources found.
```

Ограничения контекста безопасности – это концепция OpenShift, которая позволяет более детально контролировать возможности контейнеров Docker. Они управляют контекстами SELinux (см. метод 100), могут удалять функциональные возможности из запущенных контейнеров (см. метод 93), определять, от какого пользователя может запускаться модуль, и т. д.

Для запуска вы измените ограничение контекста по умолчанию `restricted`. Также можете создать отдельное ограничение контекста безопасности и применить его к определенному проекту, но можно попробовать это самостоятельно.

Чтобы изменить ограничение контекста `restricted`, вам необходимо стать администратором кластера:

```
$ oc login -u system:admin
```

Затем нужно отредактировать ограничение контекста с помощью следующей команды:

```
$ oc edit scc restricted
```

Вы увидите определение ограничение контекста `restricted`.

На этом этапе придется сделать две вещи:

- разрешить запуск контейнеров от имени любого пользователя (в данном случае от имени `root`);
- не позволять ограничению контекста безопасности ограничивать ваши мандаты до setuid и setgid.

RunAsAny

Контейнер LocalStack по умолчанию запускается от имени пользователя root, но по соображениям безопасности OpenShift не позволяет контейнерам это делать. Вместо этого он выбирает идентификатор пользователя в очень высоком диапазоне и запускается в качестве этого идентификатора. Обратите внимание, что идентификаторы пользователя – это числа, в отличие от имен пользователей, которые являются строками, отображенными в идентификатор пользователя.

Чтобы упростить задачу и разрешить контейнеру LocalStack работать от имени пользователя root, измените эти строки:

```
runAsUser:
type: MustRunAsRange
```

на эти:

```
runAsUser:
type: RunAsAny
```

Это позволяет контейнерам запускаться от имени *любого* пользователя, а не в пределах диапазона идентификаторов пользователя.

Мандаты SETUID и SETGID

Когда LocalStack запускается, он должен стать другим пользователем, чтобы запустить ElastiCache. Служба ElastiCache не запускается от имени пользователя root.

Чтобы обойти это, LocalStack позволяет пользователю выполнить команду запуска с помощью команды `su` в контейнере. Поскольку ограниченный SCC явно запрещает действия, которые изменяют идентификатор пользователя или группы, вам необходимо снять эти ограничения. Сделайте это, удалив строки:

```
- SETUID
- SETGID
```

После того как вы выполнили эти два шага, сохраните файл.

Запомните этот хост. Если вы выполните эту команду:

```
$ minishift console --machine-readable | grep HOST | sed 's/^HOST=\(.*\)/\1/'
```

вы получите хост, на котором экземпляр Minishift доступен как с вашего компьютера. Запомните его, так как вам придется позже заменить его.

Развертывание модуля

Развернуть LocalStack так же просто, как выполнить эту команду:

```
$ oc new-app localstack/localstack --name="localstack"
```

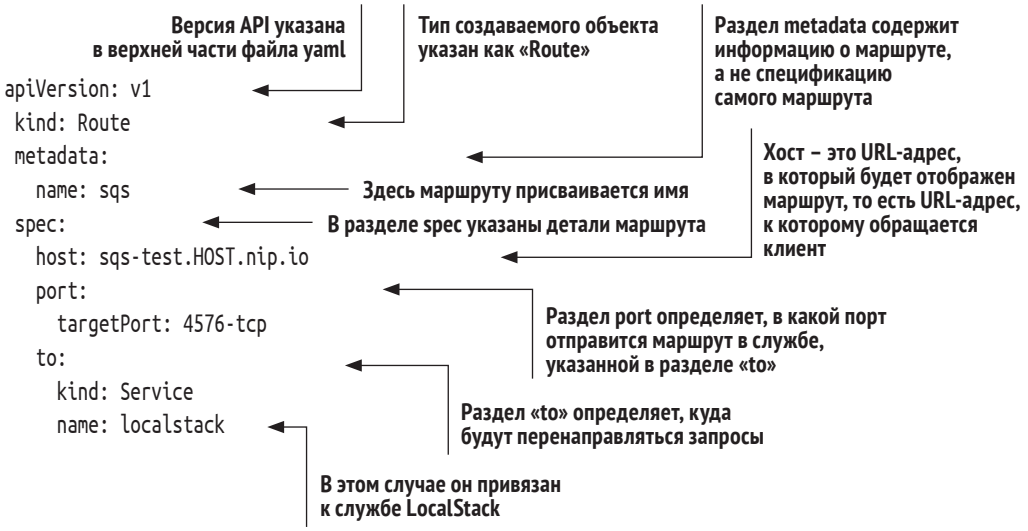
> **ПРИМЕЧАНИЕ.** Если вы хотите подробнее взглянуть на образ localstack, он доступен на странице https://github.com/localstack/localstack.

Мы берем образ localstack/localstack и создаем вокруг него приложение OpenShift, настраивая внутренние службы (на основе открытых портов в файле Dockerfile образа LocalStack), запускаем контейнер в модуле и выполняем различные другие задачи управления.

Создание маршрутов

Если вы хотите получить доступ к службам извне, нужно задать маршруты OpenShift, которые создают внешний адрес для доступа к службам в сети OpenShift. Например, чтобы придумать маршрут для службы SQS, создайте файл, подобный этому, с именем route.yaml:

Листинг 12.5. route.yaml

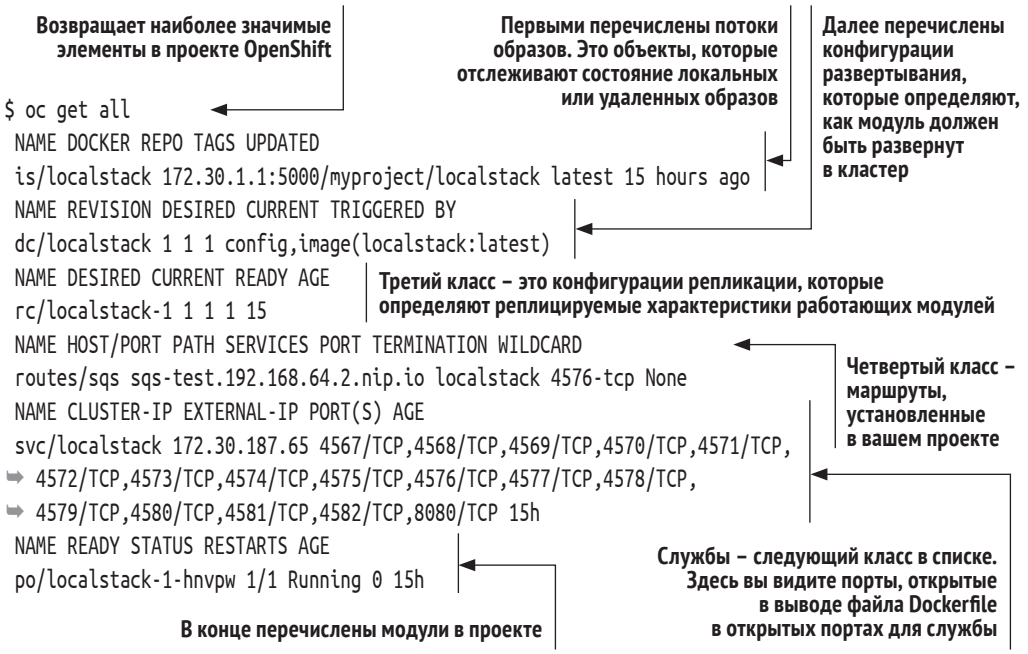


Создайте маршрут, выполнив эту команду,

```
$ oc create -f route.yaml
```

берущую маршрут из файла yaml, только что созданный вами. Затем этот процесс повторяется для каждой службы, которую вы хотите настроить.

После этого выполните команду `oc get all`, чтобы увидеть, что вы создали в своем проекте OpenShift:

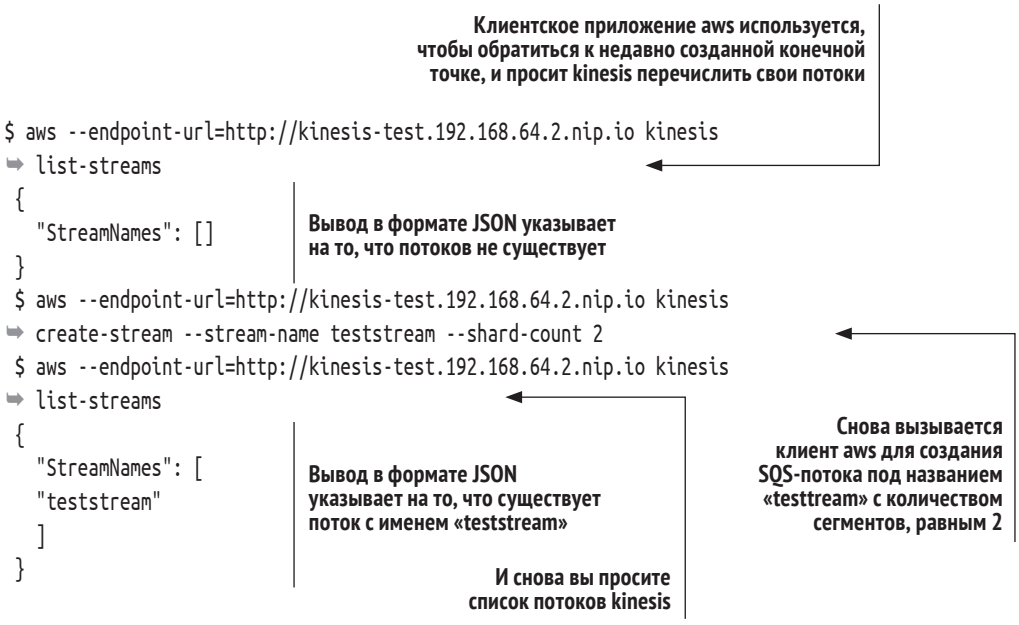


Хотя в вашем проекте технически доступны не *все* объекты, команда `oc get all` показывает наиболее важные для запуска приложений.

SQS-подобный сервис AWS теперь доступен в качестве конечной точки URL-адреса для проверки вашего кода.

Получение доступа к службам

Теперь можете получить доступ к службам вашего хоста. Вот пример создания потока SQS:



> **ПРИМЕЧАНИЕ.** Клиент `aws` – это установка, которая вам понадобится для работы. С другой стороны, можете обратиться к конечной точке API напрямую с помощью утилиты cURL, но мы не советуем этого делать. Также предполагается, что вы выполнили команду `aws configure` и указали свои ключи AWS и регион по умолчанию. Указанные значения не играют роли для LocalStack, так как он не выполняет аутентификацию.

Здесь мы рассмотрели только один тип службы, но этот метод легко распространяется на другие типы, перечисленные в начале этого метода.

ОБСУЖДЕНИЕ

Этот метод дал вам представление о силе OpenShift (и Kubernetes, на котором основан OpenShift). Запуск полезного приложения с доступной конечной точкой и полной заботой о внутреннем устройстве – это во многом реализация обещания переносимости, которое предлагает Docker, масштабируемого до центра обработки данных.

Например, можно пойти дальше и запустить несколько экземпляров LocalStack в том же кластере OpenShift. Тесты на API-интерфейсах AWS могут проводиться параллельно, не требуя дополнительных ресурсов (разумеется, в зависимости от размера вашего кластера OpenShift и требований тестов). Поскольку все это – код, непрерывная интеграция может быть настроена на динамический запуск и остановку экземпляров LocalStack при каждой фиксации вашей кодовой базы AWS.

Наряду с указанием различных аспектов Kubernetes, этот конкретный метод также демонстрирует, что такие продукты, как OpenShift, строятся поверх Kubernetes для расширения его функциональности. Например, ограничения контекста безопасности являются концепцией OpenShift (хотя контексты безопасности также существуют в Kubernetes), а «маршруты» были концепцией OpenShift, созданного поверх Kubernetes, который в конечном итоге был адаптирован для реализации непосредственно в Kubernetes. Со временем функции, разработанные для OpenShift, были переданы в Kubernetes и стали частью его подношения.

Вы снова встретитесь с OpenShift в методе 99, где мы рассмотрим, как он может служить платформой, позволяющей пользователям безопасно запускать контейнеры.

МЕТОД 91 Создание фреймворка на основе Mesos

Обсуждая множество возможностей оркестровки, вы, вероятно, найдете одну, в частности, упоминаемую в качестве альтернативы Kubernetes. Это – Mesos. Когда заходит речь о Mesos, обычно следуют неясные утверждения типа «Mesos – это фреймворк для фреймворка» и «Kubernetes можно запустить поверх Mesos».

Наиболее подходящая аналогия, с которой мы сталкиваемся, – это полагать, что Mesos предоставляет ядро для вашего центра обработки данных. По отдельности с ним нельзя сделать ничего полезного – ценность приходит, когда вы объединяете его с системой инициализации и приложениями.

Для простоты представьте, что перед панелью сидит обезьяна, управляющая всеми вашими компьютерами и обладающая способностью запускать и останавливать приложения по желанию. Естественно, вам нужно дать обезьяне *очень* четкий список инструкций о том, что делать в конкретных ситуациях, когда запускать приложение и т. д. Вы могли бы сделать все самостоятельно, но это отнимает много времени, а обезьяны стоят дешево.

Mesos – это та самая обезьяна! Он идеально подходит для компании с высокодинамичной и сложной инфраструктурой, которая, вероятно, обладает опытом внедрения собственных решений для оркестровки эксплуатации в производственных условиях. Если вы не соответствуете этим условиям, вам лучше будет использовать готовое решение, вместо того чтобы тратить время на Mesos.

ПРОБЛЕМА

У вас есть ряд правил для управления запуском приложений и заданий, и вы хотите применить их, не запуская их вручную на удаленных компьютерах и не отслеживая их состояние.

РЕШЕНИЕ

Используйте Mesos, гибкий и мощный инструмент, который обеспечивает абстракцию управления ресурсами.

Mesos – зрелое программное обеспечение для абстракции управления ресурсами на нескольких компьютерах. Он был проверен в действии в рабочем окружении компаниями, о которых вы слышали, и в результате он стабилен и надежен.

> **ПРИМЕЧАНИЕ.** Чтобы Mesos мог использовать правильную версию API Docker, вам необходим Docker 1.6.2 или более поздней версии.

На рис. 12.3 показано, как выглядит типовая установка рабочего окружения Mesos.

На этом рисунке видно, как выглядит базовый жизненный цикл Mesos для запуска задачи:

1. Ведомое устройство работает на узле, отслеживая доступность ресурсов и информируя ведущее устройство.
2. Ведущее устройство получает от одного или нескольких ведомых информацию относительно доступных ресурсов и делает предложения ресурсов планировщикам.
3. Планировщик получает предложения ресурсов от ведущего устройства, решает, где он хочет выполнять задачи, и сообщает об этом ведущему устройству.
4. Ведущее устройство передает информацию о задаче соответствующим ведомым устройствам.
5. Каждый ведомое устройство передает информацию о задаче существующему исполнителю на узле или запускает нового.
6. Исполнитель считывает информацию о задаче и запускает задачу на узле.
7. Задача выполняется.

Проект Mesos предоставляет ведущее и ведомое устройство, а также встроенный исполнитель оболочки.

Ваша задача – предоставить *фреймворк* (или *приложение*), который состоит из планировщика («список инструкций» в нашей аналогии с обезьяной) и (по желанию) пользовательского исполнителя.

Многие сторонние проекты предоставляют фреймворки, которые вы можете перенести в Mesos (и мы рассмотрим один из них более подробно в следующем методе), но, чтобы лучше понять, как можно полностью использовать возможности Mesos в Docker, создадим собственный фреймворк, состоящий

только из планировщика. Если у вас очень сложная логика для запуска приложений, это может быть ваш конечный выбранный маршрут.

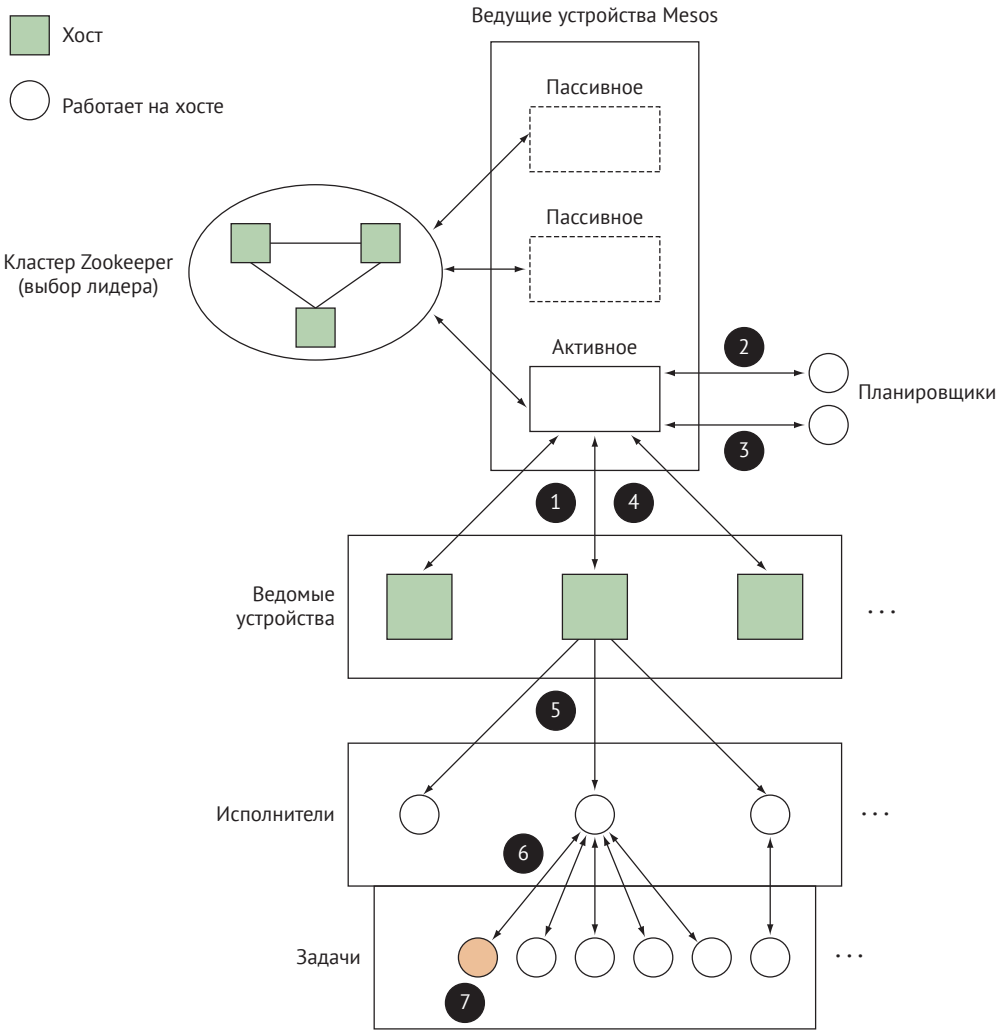


Рис. 12.3 ❖ Типовая установка рабочего окружения Mesos

ПРИМЕЧАНИЕ. Вам не нужно использовать Docker с Mesos, но, так как это книга о Docker, мы будем это делать. Не станем вдаваться в подробности, потому что Mesos очень гибок. Мы также запустим Mesos на одном компьютере, но постараемся сделать его максимально реалистичным и указать, что вам нужно сделать, чтобы запустить его в жизнь.

Мы еще не объяснили, где Docker вписывается в жизненный цикл Mesos, – последняя часть этой головоломки состоит в том, что Mesos поддерживает средства контейнеризации, позволяя изолировать исполнителей или задачи (или и то, и другое). Docker не единственный инструмент, который можно использовать здесь, но он настолько популярен, что в Mesos есть некоторые специфичные для Docker функции, с которых можно начать.

В нашем примере будем контейнеризировать только те задачи, которые мы выполняем, потому что используем исполнителя по умолчанию. Если у вас есть пользовательский исполнитель, выполняющий только языковую среду, где каждая задача включает в себя динамическую загрузку и выполнение некоего кода, вместо этого вы можете рассмотреть возможность контейнеризации исполнителя. В качестве примера использования у вас может быть виртуальная машина Java, работающая в качестве исполнителя, которая загружает и выполняет фрагменты кода на лету, избегая затрат на запуск виртуальной машины для потенциально очень маленьких задач.

На рис. 12.4 показано, что будет происходить за кулисами в нашем примере при создании новой задачи.

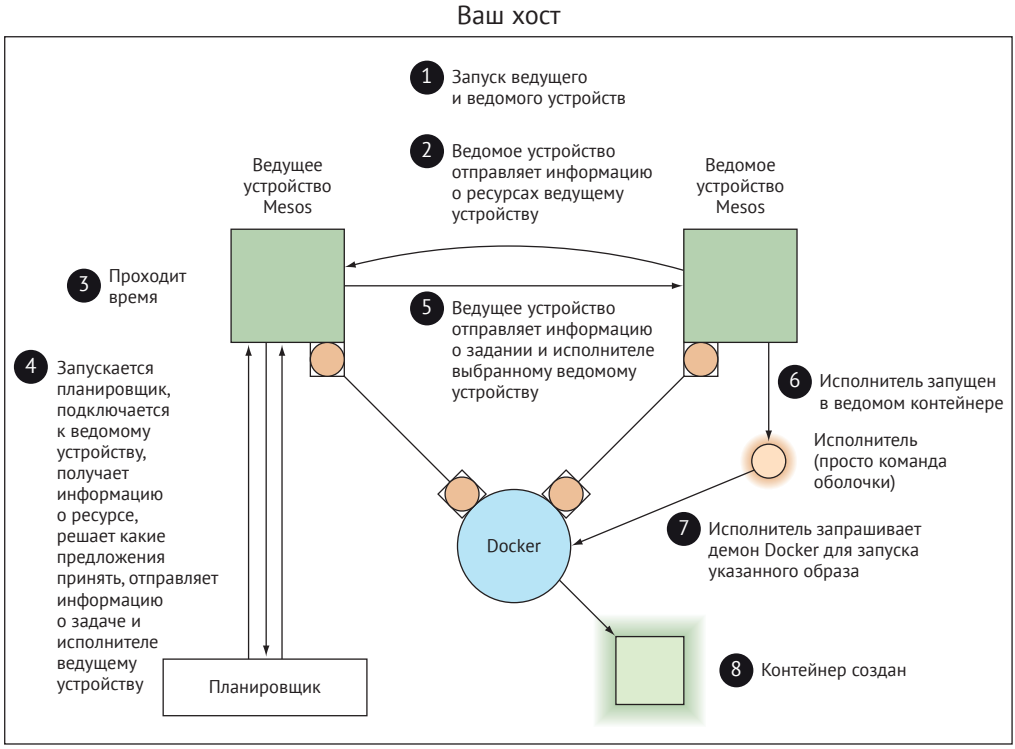


Рис. 12.4 ❖ Установка Mesos с одним хостом, запускающая контейнер

Без лишних слов давайте начнем. Для начала нужно запустить ведущее устройство:

Листинг 12.6. Запуск ведущего устройства

```
$ docker run -d --name mesmaster redjack/mesos:0.21.0 mesos-master \
--work_dir=/opt
24e277601260dcc6df35dc20a32a81f0336ae49531c46c2c8db84fe99ac1da35
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' mesmaster
172.17.0.2
$ docker logs -f mesmaster
I0312 01:43:59.182916 1 main.cpp:167] Build: 2014-11-22 05:29:57 by root
I0312 01:43:59.183073 1 main.cpp:169] Version: 0.21.0
I0312 01:43:59.183084 1 main.cpp:172] Git tag: 0.21.0
[...]
```

Запуск ведущего устройства несколько многословен, но вы должны обнаружить, что он быстро прекращает логирование. Оставьте этот терминал открытым, чтобы видеть, что происходит, когда вы запускаете другие контейнеры.

> **ПРИМЕЧАНИЕ.** Обычно в настройке Mesos будет несколько ведущих устройств Mesos (одно активное и несколько резервных копий), а также кластер Zookeeper. Его настройка описана на странице «Режим высокой доступности Mesos» на сайте Mesos (http://mesos.apache.org/documentation/latest/high-availability). Вам также необходимо открыть порт 5050 для внешнего обмена данными и использовать папку work_dir в качестве тома для сохранения постоянной информации.

Вам также нужно ведомое устройство. К сожалению, это несколько неудобно. Одной из определяющих характеристик Mesos является возможность принудительного ограничения ресурсов для задач, что требует от ведомого устройства возможности свободно проверять процессы и управлять ими. В результате команда для запуска ведомого устройства нуждается в раскрытии ряда деталей внешней системы внутри контейнера.

Листинг 12.7. Запуск ведомого устройства

```
$ docker run -d --name messlave --pid=host \
  -v /var/run/docker.sock:/var/run/docker.sock -v /sys:/sys \
  redjack/mesos:0.21.0 mesos-slave \
  --master=172.17.0.2:5050 --executor_registration_timeout=5mins \
  --isolation=cgroups/cpu,cgroups/mem --containerizers=docker,mesos \
  --resources="ports(*):[8000-8100]"
1b88c414527f63e24241691a96e3e3251fbb24996f3bfba3ebba91d7a541a9f5
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' messlave
172.17.0.3
$ docker logs -f messlave
```

```
I0312 01:46:43.341623 2398 main.cpp:142] Build: 2014-11-22 05:29:57 by root
I0312 01:46:43.341789 2398 main.cpp:144] Version: 0.21.0
I0312 01:46:43.341795 2398 main.cpp:147] Git tag: 0.21.0
[...]
I0312 01:46:43.554498 2429 slave.cpp:627] No credentials provided. >
Attempting to register without authentication
I0312 01:46:43.554633 2429 slave.cpp:638] Detecting new master
I0312 01:46:44.419646 2424 slave.cpp:756] Registered with master >
master@172.17.0.2:5050; given slave ID 20150312-014359-33558956-5050-1-S0
[...]
```

В этот момент вы также должны были увидеть активность в главном терминале Mesos, начиная с пары строк:

```
I0312 01:46:44.332494 9 master.cpp:3068] Registering slave at >
slave(1)@172.17.0.3:5051 (8c6c63023050) with id >
20150312-014359-33558956-5050-1-S0
I0312 01:46:44.333772 8 registrar.cpp:445] Applied 1 operations in >
134310ns; attempting to update the 'registry'
```

Вывод этих двух журналов показывает, что ваше ведомое устройство запущено и подключено к ведущему устройству. Если вы этого не видите, остановитесь и дважды проверьте IP-адрес своего ведущего устройства. Позже может быть неприятно пытаться устранить причину, по которой фреймворк не запускает никаких задач, когда нет подключенных ведомых устройств для запуска.

В любом случае в листинге 12.7 много чего происходит с командой. Аргументы после `run` и перед `redjack / mesos: 0.21.0` – это аргументы Docker, которые в основном состоят из предоставления ведомому контейнеру большого количества информации о внешнем мире. Аргументы, следующие после `mesos-slave`, представляют больший интерес. Во-первых, `master` сообщает ведомому устройству, где найти своего ведущего (или кластер Zookeeper). Следующие три аргумента, `executor _registration_timeout`, `isolation` и `containerizers`, – твики для настроек Mesos, которые всегда должны применяться при работе с Docker. И последнее, но не менее важное: вы должны сообщить ведомому устройству Mesos, какие порты можно использовать в качестве ресурсов. По умолчанию Mesos предлагает 31 000–32 000, но нам нужно что-то поменьше и более запоминающееся.

Теперь простых шагов уже нет, и мы подошли к финальной стадии настройки Mesos – созданию планировщика.

К счастью, у нас есть пример фреймворка, готовой к использованию. Давайте попробуем его и посмотрим, что он делает, а затем рассмотрим, как он работает. Оставьте открытыми две команды `docker logs -f`=на вашем ведущем и ведомом контейнерах, чтобы вы могли видеть обмен данными.

Следующие команды получат исходный репозиторий для примера фреймворка от GitHub и запустят его.

Листинг 12.8. Загрузка и запуск примера фреймворка

```
$ git clone https://github.com/docker-in-practice/mesos-nc.git
$ docker run -it --rm -v $(pwd)/mesos-nc:/opt redjack/mesos:0.21.0 bash
# apt-get update && apt-get install -y python
# cd /opt
# export PYTHONUSERBASE=/usr/local
# python myframework.py 172.17.0.2:5050
I0312 02:11:07.642227 182 sched.cpp:137] Version: 0.21.0
I0312 02:11:07.645598 176 sched.cpp:234] New master detected at >
master@172.17.0.2:5050
I0312 02:11:07.645800 176 sched.cpp:242] No credentials provided. >
Attempting to register without authentication
I0312 02:11:07.648449 176 sched.cpp:408] Framework registered with >
20150312-014359-33558956-5050-1-0000
Registered with framework ID 20150312-014359-33558956-5050-1-0000
Received offer 20150312-014359-33558956-5050-1-O0. cpus: 4.0, mem: 6686.0, >
ports: 8000-8100
Creating task 0
Task 0 is in state TASK_RUNNING
[...]
Received offer 20150312-014359-33558956-5050-1-O5. cpus: 3.5, mem: 6586.0, >
ports: 8005-8100
Creating task 5
Task 5 is in state TASK_RUNNING
Received offer 20150312-014359-33558956-5050-1-O6. cpus: 3.4, mem: 6566.0, >
ports: 8006-8100
Declining offer
```

Вы заметите, что мы установили Git-репозиторий внутри образа Mesos, потому что он содержит все библиотеки Mesos, которые нам нужны. К сожалению, в противном случае устанавливать их может быть несколько проблематично.

Наш фреймворк `mesos-nc` предназначен для выполнения команды `echo 'hello <task id>' | nc -l <port>` на всех доступных хостах, на всех доступных портах от 8000 до 8005. Из-за того, как работает netcat, эти «серверы» прекратят работу, как только вы получите к ним доступ, будь то через curl, Telnet, nc или ваш браузер. Можете убедиться в этом, выполнив команду `curl localhost: 8003` в новом терминале. Он вернет ожидаемый ответ, и ваши журналы Mesos покажут порождение задачи для замены завершенной. Вы также можете отслеживать, какие задачи выполняются с помощью команды `docker ps`.

Здесь стоит указать на то, что Mesos отслеживает выделенные ресурсы и помечает их как доступные после завершения задачи. В частности, когда вы получили доступ к localhost: 8003 (не стесняйтесь попробовать снова),

внимательно посмотрите на строку `Received offer` – она показывает два диапазона портов (поскольку они не подключены), включая только что освобожденный:

```
Received offer 20150312-014359-33558956-5050-1-O45. cpus: 3.5, mem: 6586.0, >
ports: 8006-8100,8003-8003
```

Ведомое устройство Mesos именует все контейнеры, которые оно запускает, используя префикс «mesos-», и это предполагает, что оно может свободно управлять чем-либо подобным. Будьте осторожны с именами своих контейнеров, иначе в конечном итоге ведомое устройство Mesos может себя «убить».

Код фреймворка (myframework.py) хорошо закомментирован на случай, если вы хотите приключений. Мы пройдемся по дизайну высокого уровня.

```
class TestScheduler
(mesos.interface.Scheduler):
[...]
    def registered(self, driver, frameworkId, masterInfo):
[...]
    def statusUpdate(self, driver, update):
[...]
    def resourceOffers(self, driver, offers):
[...]
```

Все планировщики Mesos делят базовый класс планировщика Mesos на подклассы и реализуют ряд методов, которые Mesos будет вызывать в соответствующих точках, чтобы ваш фреймворк реагировал на события. Хотя мы реализовали три метода в предыдущем фрагменте кода, два из них не являются обязательными и были реализованы для добавления дополнительного журналирования в демонстрационных целях. Единственный метод, который вы *должны* реализовать, – это `resourceOffers`: нет особого смысла во фреймворке, который не знает, когда он может запускать задачи. Вы можете добавить любые дополнительные методы для собственных целей, такие как *init* и `_makeTask`, пока они не конфликтуют ни с одним из методов, которые Mesos надеется использовать, поэтому обязательно прочитайте документацию (http://mesos.apache.org/documentation/latest/app-framework-development-guide/).

> **СОВЕТ.** Если вы в конечном итоге напишете свой собственный фреймворк, вы захотите взглянуть на документирование методов и конструкций. К сожалению, на момент написания единственная сгенерированная документация относится к методам Java. Читатели, ищущие отправную точку, чтобы покопаться в структурах, могут начать с файла include/mesos/mesos.proto в исходном коде Mesos. Удачи!

Давайте рассмотрим немного подробнее основной интересующий нас метод: `resourceOffers`. Здесь принимается решение запустить задачи или

отклонить предложение. На рис. 12.5 показан поток выполнения после вызова `resourceOffers` в нашем фреймворке Mesos (обычно потому, что некоторые ресурсы стали доступны для использования фреймворком).

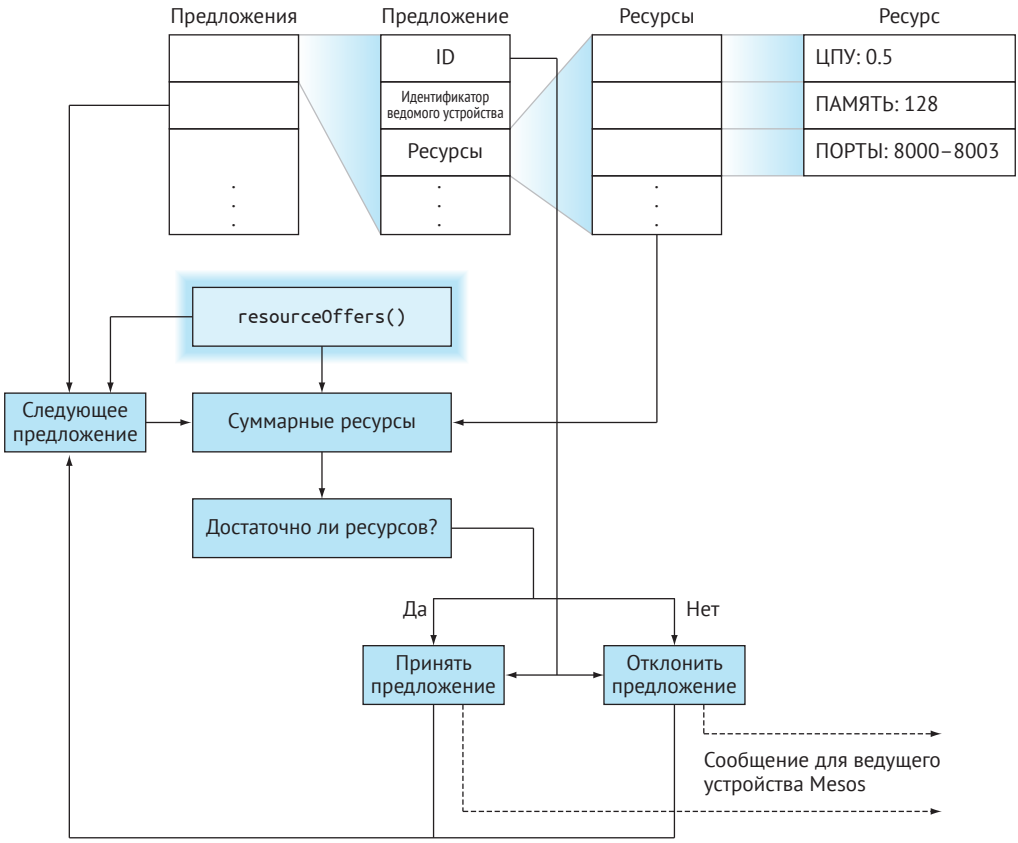


Рис. 12.5 ❖ Поток выполнения resourceOffers

`resourceOffers` предоставляется список предложений, где каждое предложение соответствует одному ведомому устройству Mesos. Предложение содержит сведения о ресурсах, доступных для задачи, запущенной на ведомом устройстве, а типичная реализация будет использовать эту информацию для определения наиболее подходящих мест для запуска задач, которые она хочет выполнить. При запуске задачи ведущему устройству Mesos отправляется сообщение, после чего продолжается жизненный цикл, описанный на рис. 12.3.

ОБСУЖДЕНИЕ

Важно отметить гибкость `resourceOffers` – ваши решения о запуске задач могут зависеть от любых критериев, которые вы выберете, от проверок работоспособности внешних служб до фазы Луны. Такая гибкость может быть

бременем, поэтому существуют готовые фреймворки, позволяющие убрать некоторые из этих низкоуровневых деталей и упростить использование Mesos. Один из этих фреймворков рассматривается в следующем методе.

Возможно, вы захотите обратиться к книге Роджера Игнацио *Mesos в действии* (Manning, 2016) для получения более подробной информации о том, что можно делать с помощью Mesos, – мы только немного коснулись этой темы, и вы увидели, как легко подключается Docker.

МЕТОД 92 Микроуправление Mesos с помощью Marathon

К настоящему времени вы уже поняли, что вам нужно многое обдумывать при использовании Mesos, даже когда речь идет об очень простом фреймворке. Иметь возможность полагаться на правильность развертывания приложений чрезвычайно важно – влияние ошибки во фреймворке может варьироваться от невозможности развертывания новых приложений до полного отказа службы.

Ставки увеличиваются по мере масштабирования, и, если ваша команда не будет писать надежный код динамического развертывания, можно подумать о более проверенном подходе – сам Mesos очень стабилен, но собственный, созданный на заказ фреймворк может быть не таким надежным, как вам бы хотелось.

Marathon подходит для компании, не имеющей собственного инструментария для развертывания, но требующей хорошо поддерживаемого и простого в использовании решения для развертывания контейнеров в отчасти динамичной среде.

ПРОБЛЕМА

Вам нужен надежный способ использовать всю мощь Mesos, не увязая в написании собственного фреймворка.

РЕШЕНИЕ

Используйте Marathon, слой поверх Mesos, который предоставляет более простой интерфейс для повышения производительности.

Marathon – это фреймворк Apache Mesos, созданный компанией Mesosphere для управления долгоживущими приложениями. Маркетинговые материалы описывают его как демон `init` или `upstart` для центра обработки данных (где Mesos – это ядро). Эта аналогия имеет под собой основу.

Marathon позволяет легко начать работу, запуская один контейнер с ведущим иди ведомым устройством Mesos и самим Marathon внутри. Это полезно для демонстраций, но не подходит для рабочего развертывания. Чтобы получить реалистичную настройку Marathon, вам понадобятся ведущее и ведомое устройства Mesos (из предыдущего метода), а также экземпляр Zookeeper (из метода 84). Убедитесь, что все это работает, и мы начнем с запуска контейнера Marathon.

```
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' mesmaster
172.17.0.2
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' messlave
172.17.0.3
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' zookeeper
172.17.0.4
$ docker pull mesosphere/marathon:v0.8.2
[...]
$ docker run -d -h $(hostname) --name marathon -p 8080:8080 \
mesosphere/marathon:v0.8.2 --master 172.17.0.2:5050 --local_port_min 8000 \
--local_port_max 8100 --zk zk://172.17.0.4:2181/marathon
accd6de46cfab65572539ccffa5c2303009be7ec7dbfb49e3ab8f447453f2b93
$ docker logs -f marathon
MESOS_NATIVE_JAVA_LIBRARY is not set. Searching in /usr/lib /usr/local/lib.
MESOS_NATIVE_LIBRARY, MESOS_NATIVE_JAVA_LIBRARY set to >
'/usr/lib/libmesos.so'
[2015-06-2319:42:14,836] INFO Starting Marathon 0.8.2 >
(mesosphere.marathon.Main$:87)
[2015-06-2319:42:16,270] INFO Connecting to Zookeeper... >
(mesosphere.marathon.Main$:37)
[...]
[2015-06-3018:20:07,971] INFO started processing 1 offers, >
launching at most 1 tasks per offer and 1000 tasks in total
➥ (mesosphere.marathon.tasks.IterativeOfferMatcher$:124)
[2015-06-3018:20:07,972] INFO Launched 0 tasks on 0 offers, >
declining 1 (mesosphere.marathon.tasks.IterativeOfferMatcher$:216)
```

Как и сам Mesos, Marathon довольно многословен, но (как и Mesos) останавливается довольно быстро. На этом этапе он войдет в цикл, с которым вы знакомы по написанию своего собственного фреймворка – рассматривая предложения ресурсов и решая, что с ними делать. Поскольку мы еще ничего не запустили, не должно быть видно никакой активности; отсюда `declining 1` в предыдущем журнале.

Marathon поставляется с симпатичным веб-интерфейсом, поэтому мы открыли порт 8080 на хосте – перейдите по адресу http://localhost:8080 в своем браузере, чтобы получить его.

Мы собираемся погрузиться в Marathon напрямую, поэтому давайте создадим новое приложение. Чтобы немного прояснить терминологию, «приложение» в мире Marathon – это группа из одной или нескольких задач с точно таким же определением.

Нажмите кнопку **New App** (Новое приложение) в правом верхнем углу, чтобы открыть диалоговое окно, которое можно использовать для определения приложения, которое вы хотите запустить. Мы продолжим в духе фреймворка, который создали сами, установив значение идентификатора как «marathon-nc», оставив для процессора, памяти и дискового пространства значения

по умолчанию (чтобы соответствовать ограничениям ресурсов, наложенным на наш фреймворк mesos-nc) и установив команду как `echo "hello $MESOS_TASK_ID" | nc -l $PORT0` (используя переменные среды, доступные для задачи – обратите внимание, что это ноль). Установите в поле **Порты** значение 8000, чтобы указать, где вы хотите слушать. Другие поля мы сейчас пропустим. Нажмите **Create** (Создать).

Ваше вновь определенное приложение теперь будет отображаться в веб-интерфейсе. Состояние будет кратко отображаться как **Deploying** (Развертывание), а затем **Running** (Запуск). Ваше приложение теперь запущено!

Если нажмете на запись «/marathon-nc» в списке приложений, вы увидите уникальный идентификатор своего приложения. Можете получить полную конфигурацию из REST API, как показано в следующем фрагменте, а также убедиться, что все работает, обратившись к ведомому контейнеру Mesos на соответствующем порту с помощью утилиты cURL. Убедитесь, что вы сохранили полную конфигурацию, возвращенную REST API, так как это пригодится позже, – в приведенном ниже примере она была сохранена в файле app.json.

```
$ curl http://localhost:8080/v2/apps/marathon-nc/versions
{"versions":["2015-06-30T19:52:44.649Z"]}
$ curl -s \
http://localhost:8080/v2/apps/marathon-nc/versions/2015-06-30T19:52:44.649Z \
> app.json
$ cat app.json
{"id":"/marathon-nc", >
"cmd":"echo \"hello $MESOS_TASK_ID\" | nc -l $PORT0",[...]
$ curl http://172.17.0.3:8000
hello marathon-nc.f56f140e-19e9-11e5-a44d-0242ac110012
```

Обратите внимание на текст, следующий за словом «hello», – он должен соответствовать уникальному идентификатору в интерфейсе. Однако делайте это быстро – выполнение команды `curl` приведет к завершению приложения, Marathon перезапустит его, и уникальный идентификатор в веб-интерфейсе изменится. После того как вы проверили все это, нажмите кнопку **Destroy App** (Уничтожить приложение), чтобы удалить marathon-nc.

Работает хорошо, но вы, возможно, заметили, что мы не достигли того, что намеревались сделать с помощью Marathon, – это оркестровка контейнеров Docker. Хотя наше приложение находится внутри контейнера, оно было запущено в ведомом контейнере Mesos, а не в собственном. При прочтении документации к Marathon мы узнаем, что для создания задач внутри контейнеров Docker требуется больше настроек (как это было при написании собственного фреймворка).

К счастью, у ведомого устройства Mesos, который мы запускали ранее, есть необходимые настройки, поэтому просто нужно изменить некоторые параметры Marathon, в частности, параметры приложения. Взяв ответ Marathon API из предыдущего примера (сохраненного в app.json), мы можем

сосредоточиться на добавлении настроек Marathon, которые позволяют использовать Docker. Чтобы выполнить здесь эту манипуляцию, будем использовать удобный инструмент `jq`, хотя это также легко сделать с помощью текстового редактора.

```
$ JQ=https://github.com/stedolan/jq/releases/download/jq-1.3/jq-linux-x86_64
$ curl -Os $JQ && mv jq-linux-x86_64 jq && chmod +x jq
$ cat >container.json <<EOF
{
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "ubuntu:14.04.2",
      "network": "BRIDGE",
      "portMappings": [{"hostPort": 8000, "containerPort": 8000}]
    }
  }
}
$ # merge the app and container details
$ cat app.json container.json | ./jq -s add > newapp.json
```

Теперь можем отправить определение нового приложения в API и увидеть, как Marathon запускает его:

```
$ curl -X POST -H 'Content-Type: application/json; charset=utf-8' \
--data-binary @newapp.json http://localhost:8080/v2/apps
{"id":"/marathon-nc", >
"cmd":"echo \"hello $MESOS_TASK_ID\" | nc -l $PORT0",[...]
$ sleep 10
$ docker ps --since=marathon
CONTAINER ID  IMAGE         COMMAND              CREATED          >
STATUS             PORTS                   NAMES
284ced88246c ubuntu:14.04 "\"/bin/sh -c 'echo About a minute ago  >
Up About a minute 0.0.0.0:8000->8000/tcp mesos- >
1da85151-59c0-4469-9c50-2bfc34f1a987
$ curl localhost:8000
hello mesos-nc.675b2dc9-1f88-11e5-bc4d-0242ac11000e
$ docker ps --since=marathon
CONTAINER ID  IMAGE         COMMAND              CREATED          >
STATUS             PORTS                   NAMES
851279a9292f ubuntu:14.04 "\"/bin/sh -c 'echo 44 seconds ago >
Up 43 seconds 0.0.0.0:8000->8000/tcp mesos- >
37d84e5e-3908-405b-aa04-9524b59ba4f6
284ced88246c ubuntu:14.04 "\"/bin/sh -c 'echo 24 minutes ago >
Exited (0) 45 seconds ago                 mesos-1da85151-59c0-
➥ 4469-9c50-2bfc34f1a987
```

Как и в случае с нашей пользовательским фреймворком в последнем методе, Mesos запустил для нас контейнер Docker с работающим приложением. Выполнение команды `curl` завершает работу приложения и контейнера, и автоматически запускается новое.

ОБСУЖДЕНИЕ

Есть несколько существенных отличий между пользовательским фреймворком из последнего метода и Marathon. Например, в пользовательском фреймворке у нас был чрезвычайно детальный контроль над принятием предложений ресурсов до того момента, когда мы могли выбирать отдельные порты для прослушивания. Чтобы сделать подобное в Marathon, следует применить настройки для каждого отдельного ведомого устройства.

С другой стороны, Marathon поставляется с множеством встроенных функций, которые могут создавать ошибки, включая проверку работоспособности, систему уведомлений о событиях и REST API. Эти вещи не являются банальными для реализации, и использование Marathon позволяет вам работать с уверенностью, что вы не первый, кто его пробует. Если не сказать больше, получить поддержку Marathon намного проще, чем специализированную среду, и мы обнаружили, что документация для Marathon более доступна, чем для Mesos.

Мы рассмотрели основы настройки и использования Marathon, но есть еще много вещей, которые можно посмотреть и сделать. Одним из наиболее интересных предложений является использование Marathon для запуска других сред Mesos, потенциально включая вашу собственную! Мы рекомендуем вам это исследовать. Mesos – высококачественное инструментальное средство для оркестровки, а Marathon предлагает удобный слой поверх него.

Резюме

- Вы можете запускать службы на кластере компьютеров в режиме Docker Swarm.
- Написание собственного фреймворка для Mesos может дать вам детальный контроль над расписанием контейнеров.
- Фреймворк Marathon поверх Mesos предоставляет простой способ использовать некоторые возможности Mesos.
- Kubernetes – это инструментальное средство для оркестровки. У него есть API, который вы можете применять.
- OpenShift можно использовать для настройки локальной версии некоторых сервисов AWS.

Глава 13

Платформы Docker

О чем рассказывается в этой главе:

- факторы, влияющие на выбор платформы Docker;
- области рассмотрения, необходимые при принятии Docker;
- положение рынка поставщиков Docker по состоянию на 2018 год.

Название этой главы может показаться странным. Разве в предыдущей главе уже не шла речь о таких платформах Docker как Kubernetes и Mesos?

Ну, и да, и нет. Хотя Kubernetes и Mesos, возможно, являются платформами, на которых вы можете запускать Docker, в этой книге мы берем *платформу* для обозначения продукта (или интегрированного набора технологий), который позволяет вам запускать и управлять работой контейнеров Docker в структурированном виде. Можете рассматривать эту главу как более инфраструктурную, нежели чисто техническую.

На момент написания этой главы существует несколько платформ Docker:

- AWS Fargate;
- OpenShift;
- AWS ECS (Elastic Container Service);
- центр обработки данных Docker;
- AWS EKS (Elastic Kubernetes Service);
- «нативный» Kubernetes;
- Azure AKS (Azure Kubernetes Service).

ПРИМЕЧАНИЕ. «Нативный» Kubernetes означает запуск и управление собственным кластером в зависимости от того, какую базовую инфраструктуру вы предпочитаете. Возможно, вы захотите запустить его на выделенном оборудовании в собственном центре обработки данных или на виртуальных машинах у облачного провайдера.

Сложный момент в принятии платформы – это решить, какую платформу выбрать, и знать, что следует учитывать при рассмотрении вопроса о внедрении Docker в организации. В этой главе будет представлена карта решений, которые необходимо принять, чтобы сделать разумный выбор платформы. Это поможет вам понять, почему можно выбрать OpenShift вместо Kubernetes или AWS ECS вместо Kubernetes и т. д.

Глава состоит из трех частей. В первой рассматриваются факторы, которые определяют решения о том, какие технологии или решения подходят для организации, желающей внедрить Docker. Во второй части обсуждаются области, которые необходимо учитывать при выборе Docker. В третьей обсуждается положение рынка поставщиков по состоянию на 2018 год.

Мы развертывали Docker в нескольких организациях и говорили о проблемах перехода на многочисленных конференциях, а также в этих организациях. Данный опыт научил нас тому, что, хотя совокупность проблем, с которыми сталкиваются эти организации, уникальна, существуют шаблоны решений и классы задач, которые необходимо понять, прежде чем отправиться в путь.

13.1. Факторы организационного выбора

В этом разделе будут описан ряд основных факторов в вашей организации, которые могут повлиять на выбор платформы для Docker. На рис. 13.1 показаны некоторые из этих факторов и их взаимосвязь.

Прежде чем подробно обсудить их, мы кратко опишем каждый из них и расскажем, что он означает. Возможно, вы уже рассматривали все эти факторы и поняли, что это такое, но разная терминология внутри и между организациями может привести к путанице в терминологии, а некоторые термины чаще используются в одних организациях, чем в других.

- *Покупка по сравнению со сборкой*. Относится к разнице в подходах, которые организации используют для развертывания нового программного обеспечения. Некоторые организации предпочитают покупать решения, а другие – создавать и поддерживать их самостоятельно. Это, в свою очередь, может влиять на то, какая платформа (или платформы) выбрана.
- *Технические факторы*. Некоторые предприятия дифференцируются в зависимости от конкретных характеристик своей технологии, таких как высокий уровень производительности или экономическая эффективность эксплуатации. То, что лежит в основе этих характеристик, может быть очень нишевым, и конкретные технические компоненты могут не обслуживаться товарными услугами или инструментами. Это может привести к появлению большего количества индивидуальных решений, которые стимулируют подход типа «создавать», а не «покупать».
- *Монолитное против частичного*. Опять же, это общий культурный подход, который могут использовать организации в отношении программных решений. Некоторые предпочитают централизовать решения

в одном монолитном объекте (централизованном сервере или службе), а другие – решать проблемы по частям. Последний подход можно рассматривать как более гибкий и адаптируемый, тогда как первый может быть более эффективным в масштабе.

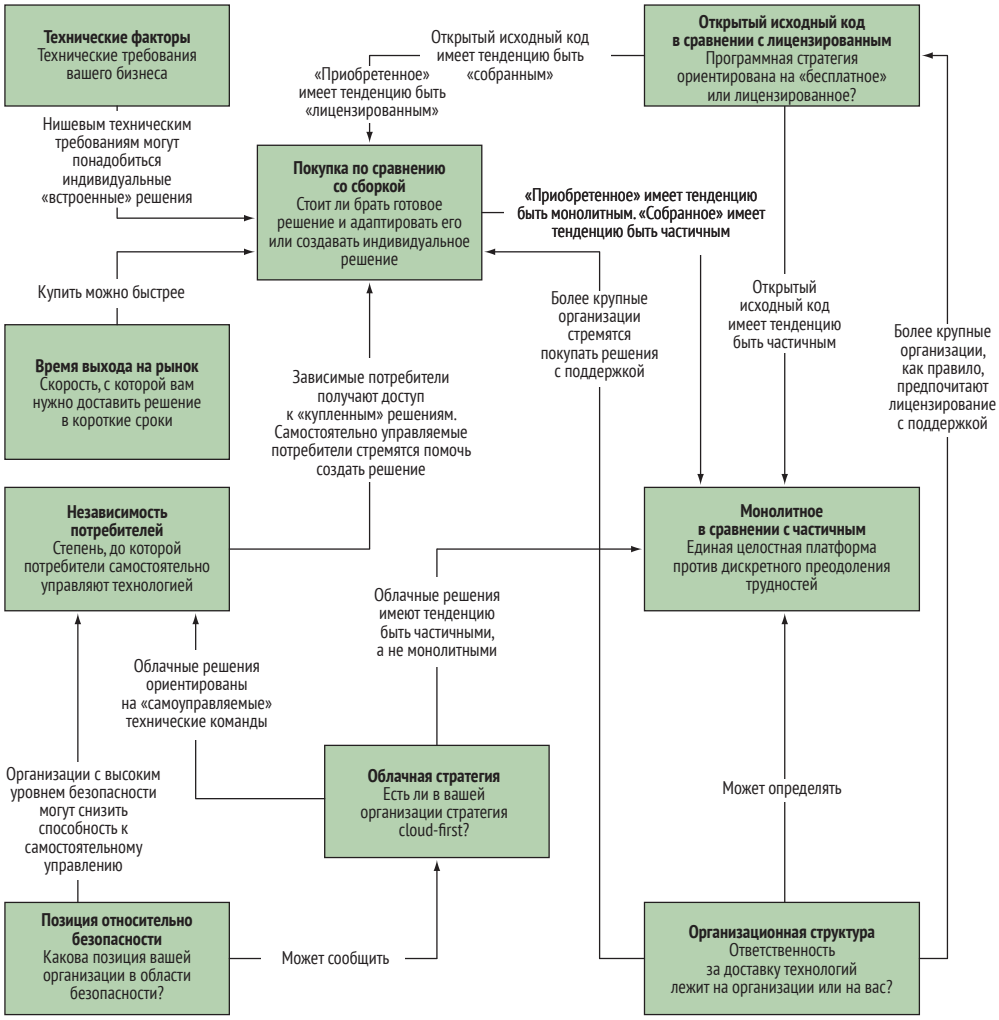


Рис. 13.1 ❖ Факторы, определяющие выбор платформы

- *Время выхода на рынок*. Часто организации испытывают давление (по коммерческим или культурным причинам), чтобы быстро предоставить решение своим пользователям. Это давление может отдавать предпочтение определенным платформам по сравнению с другими за счет затрат или гибкости в будущем.
- *Открытый исходный код по сравнению с лицензированным*. В наши дни организации обычно предпочитают открытый исходный код

по сравнению с лицензированными продуктами, но для лицензирования продукта у поставщика все же могут быть веские причины. Еще одна связанная с этим тема, которая подталкивает организации к выбору решений с открытым исходным кодом, – это боязнь привязки к конкретному поставщику или платформе, что ведет к увеличению стоимости лицензий, поскольку с течением времени зависимость от этого продукта сохраняется.

- *Независимость от потребителя.* У развернутой платформы будут потребители. Это могут быть отдельные лица, команды или целые бизнес-единицы. Какими бы ни были размеры этих потребителей, у них будет культура и режим работы. Ключевые вопросы, которые следует здесь задать: насколько они самоуправляемы в своем операционном контексте технически, и каковы их потребности в развитии? Ответы на эти вопросы могут определить характер платформы, которую вы решите развернуть.
- *Облачная стратегия.* В настоящее время есть мало организаций, которые не имеют определенной позиции в отношении облачных вычислений. Независимо от того, хотите ли вы немедленно перенести рабочие нагрузки в облако или нет, степень, в которой решение является облачным, может стать фактором в процессе принятия решений. Даже если вы решили перейти в облако, вам все равно нужно подумать, ограничена ли стратегия одним облаком или она предназначена для переноса в облака и обратно в центр обработки данных.
- *Отношение к безопасности*. Организации все более серьезно относятся к безопасности в рамках своей ИТ-стратегии. Будь то спонсируемые государством лица, хакеры-любители (или профессионалы), промышленный шпионаж или обычная кража, безопасность – это то, на что у каждого есть своя позиция. Уровень внимания, уделяемого этой области, варьируется, поэтому это может повлиять на выбор платформы.
- *Организационная структура*. Многие из приведенных выше определений могут значить для вас больше, если вы работаете в организации предприятия, нежели если вы работаете в организации противоположного типа.

В этой книге мы в общих чертах определили *предприятие* как организацию, в которой существует низкая степень независимости между отдельными функциями внутри нее. Например, если вы запускаете централизованную ИТ-функцию, можете ли вы без последствий развертывать решения без привязки к какой-либо другой части бизнеса (например, безопасности, командам разработчиков, командам разработчиков инструментальных средств, финансам, командам DevOps)? Если это так, то мы рассматриваем это как противоположность организации предприятия. Корпоративные организации склонны быть большими (поэтому функции более дискретные) и более регулируемыми (внутренне и внешне), что склонно ограничивать их свободу осуществлять изменения с меньшими последствиями.

В отличие от этого организация, не относящаяся к предприятию (в этой книге), представляет собой организацию, в которой функции могут свободно развертывать решения по своему усмотрению в процессе самоопределения. Согласно этому определению, стартапы часто рассматриваются как организации, не относящиеся к предприятию, потому что они могут принимать решения быстро и без ссылки на - или с более быстрым определением – потребности других.

Хотя организации, не относящиеся к предприятиям, склонны отдавать предпочтение некоторым стратегиям (таким как создание вместо приобретения), им все равно стоит задуматься о последствиях таких решений для бизнеса в долгосрочной перспективе.

Давайте посмотрим более конкретно на то, как различные факторы взаимодействуют, чтобы противостоять разным платформам или против них. Надеемся, что некоторые из них будут резонировать с вашим опытом или ситуацией.

После этого обсуждения мы продолжим рассмотрение конкретных проблем, которые может создать платформа Docker. С учетом этих факторов в качестве контекста вы можете принять обоснованное решение о том, какая технология наилучшим образом соответствует потребностям вашей организации.

13.1.1. Время выхода на рынок

Может быть полезно сначала рассмотреть самый простой из факторов: время выхода на рынок. Каждый, кто работает в организации, испытывает некоторое давление, чтобы быстро предлагать решения, но степень обсуждения или желаний может варьироваться.

Если прямой конкурент принял стратегию контейнеризации и успешно использует ее для снижения затрат, старшее руководство может быть заинтересовано в том, сколько времени потребуется для реализации вашего решения.

В качестве альтернативы, если вы работаете в более консервативной организации, быстрое решение может привести к негативным последствиям, таким как привязка к ускоренной доставке или платформе «хит сезона», которая не может идти дальше с учетом изменяющихся потребностей.

Мудрые руководители могут посоветовать вам не поддаваться искушению принять первое надежное решение перед лицом этих опасностей.

В целом, стремление к быстрым поставкам заставляет двигаться в направлении решений «покупать», а не «собирать» и «монолитно», а не «по частям», когда имеешь дело со сложными корпоративными задачами. (Все варианты будут обсуждаться более подробно в следующем разделе.) Эти проблемы можно решить, возложив ответственность за их решение на решения этих поставщиков. Но это не всегда возможно, особенно если продукт не зрелый.

Необходимость доставки также может привести к поспешным поставкам индивидуальных решений, которые удовлетворяют краткосрочные потребности бизнеса. Это особенно распространено в организациях с высокой технической направленностью, и может быть очень эффективным, обеспечивая

преимущество над конкурентами посредством контроля над основными технологиями и знание того, как они работают. Если технология не является критическим отличием для вашего бизнеса, это может привести к тому, что позднее отойти от технологии белых слонов станет труднее, если отрасль опередит ваше преимущество.

Аналогичным образом внедрение облачных технологий «нажми и потребляй» может значительно сократить время выхода на рынок. Недостатком будет последующая привязка к решению этого провайдера, увеличение затрат по мере масштабирования и стоимость любого будущего удаления. Это также может снизить гибкость технических функций или решений, что делает вас зависимым от роста и разработки продукта поставщика облачных вычислений.

13.1.2. Покупка по сравнению со сборкой

Покупка решения может быть эффективной стратегией по нескольким причинам. Как вы уже видели, это может привести к сокращению времени выхода на рынок. Если ваша организация ограничена с точки зрения штата разработчиков, вы также можете использовать расширяющий набор функций продукта, чтобы предложить своим клиентам больше при относительно небольших инвестициях.

Покупка может также снизить операционные расходы, если вы решите использовать продукт вне офиса в качестве услуги, предоставляемой поставщиком. Степень, в которой вы выбираете этот путь, может быть ограничена вашей позицией безопасности: программное обеспечение можно считать безопасным для запуска, только если оно работает на оборудовании, принадлежащем и эксплуатируемом организацией, использующей его.

Если вы читаете эту книгу, вам может понравиться самостоятельное создание платформы либо с нуля, либо из существующего программного обеспечения с открытым исходным кодом. Вы, несомненно, многому научитесь в ходе этого процесса, но при таком подходе существует множество опасностей с точки зрения бизнеса.

Во-первых, вам, вероятно, понадобится высококвалифицированный персонал, чтобы продолжать создавать и поддерживать этот продукт. Найти людей, которые могут программировать и управлять сложными ИТ-системами, особенно в последние годы, когда такие навыки пользуются повышенным спросом, может быть намного сложнее, чем вы думаете (особенно если вы были окружены специалистами по информатике на работе и в университете).

Во-вторых, с течением времени мир контейнерных платформ станет зрелым, и зарекомендовавшие себя игроки предложат схожие наборы функций и полезные для них навыки. На фоне этих предложений индивидуальное решение, созданное для нужд конкретной организации несколько лет назад, может показаться излишне дорогостоящим там, где когда-то это было отличительным признаком рынка.

Одна из стратегий, которая может быть принята, – это «создать, а затем купить», когда организация создает платформу для удовлетворения своих насущных потребностей, но стремится покупать, когда рынок остановился на продукте, который выглядит как стандарт. Конечно, есть опасность, что встроенная платформа станет «домашним любимцем», от которого трудно отказаться. На момент написания статьи Kubernetes, похоже, приобрел практически полное господство в качестве основы большинства популярных платформ Docker. Поэтому вы можете отказаться от своего индивидуального решения в пользу Kubernetes, если считаете, что это хорошая ставка на будущее.

Есть платформа, которая сделала две ставки на ранних этапах. Это была OpenShift, которая приняла Docker вскоре после того, как тот вышел на техническую сцену. Она переписала всю свою кодовую базу для Docker и Kubernetes. В настоящее время это очень популярный вариант для предприятий. С другой стороны, Amazon использовала Mesos в качестве основы своего решения ECS, которое становилось все более нишевым, поскольку Kubernetes стал более распространенным.

13.1.3. Монолитное против частичного

Вопрос о том, следует ли запускать единую «монолитную» платформу для всех своих потребностей в Docker или создавать функциональность из отдельных «частичных» решений, тесно связан с вопросом «покупка по сравнению со сборкой». При рассмотрении вопроса о покупке монолитного решения у поставщика, время выхода на рынок может быть убедительным поводом присоединиться к ним. Опять же, здесь есть компромиссы.

Самая большая опасность – это так называемая блокировка. Некоторые поставщики взимают плату за каждый компьютер, на котором развернуто решение. Если ваши объемы работы с Docker со временем значительно возрастают, затраты на лицензирование могут стать чрезмерно высокими, и платформа может стать финансовым камнем преткновения, висящим у вас на шее. Некоторые поставщики даже отказываются поддерживать контейнеры Docker, доставленные другими поставщиками, что делает практически невозможным их реалистичное использование.

Есть частичный подход против этого. Говоря «частичный» мы подразумеваем, что у вас может (например) быть одно решение для сборки контейнеров, другое решение для их хранения (например, реестр Docker), еще одно – для сканирования контейнеров и одно – для их запуска (возможно, даже несколько решений для этой или любой другой предыдущей категории).

Мы более подробно расскажем о том, какие «кусочки» могут понадобиться для решения, в следующем разделе этой главы.

Опять же, если вы небольшая (и, возможно, богатая) организация, которая требует быстрого продвижения, монолитный подход может помочь вам. Пошаговый подход позволяет принимать различные решения для разных частей по мере необходимости, давая вам больше гибкости и сосредоточенности в ваших усилиях.

13.1.4. Открытый исходный код по сравнению с лицензированным

ПО с открытым исходным кодом прошло долгий путь за последнее десятилетие, так что теперь оно является стандартным требованием для продаваемых или поддерживаемых решений. Это содержит в себе опасность, которая не всегда очевидна. Хотя многие решения имеют открытый исходный код, не факт, что вы избежите блокировки. Теоретически интеллектуальная собственность программного обеспечения доступна для использования, если вы не согласны с поставщиком поддержки, но зачастую навыки, необходимые для управления и поддержки кодовой базы таковыми не являются.

Как недавно сказал один из участников конференции, «поддержка открытого исходного кода плюс поддержка поставщиков – это новая блокировка». Можно утверждать, что это является обоснованным оправданием вклада, который поставщик вносит в вашу организацию, – если для того, чтобы управлять необходимой платформой требуется много редких навыков, вам нужно будет заплатить за нее так или иначе.

Интересным дополнением к этому сочетанию являются решения для облачных вычислений, которые можно рассматривать как ПО с открытым исходным кодом, так и по лицензии. Они часто основаны на программном обеспечении с открытым исходным кодом и открытых стандартах (таких как Amazon EKS), но они могут привязать вас к конкретной реализации этих стандартов и технологий и таким образом закрепить вашу блокировку.

Еще один интересный микс наблюдается в таких платформах, как OpenShift от Red Hat. OpenShift – это предоставляемая поставщиком платформа с лицензиями, необходимыми для ее запуска. Но ее код доступен на GitHub, и вклады сообщества могут быть приняты в основной строке. Red Hat предлагает дополнительную поддержку, разработку функций и поддержку исторической базы кода. Поэтому теоретически вы можете отказаться от их реализации, если чувствуете, что не получаете выгоды от их предложения.

13.1.5. Отношение к безопасности

Проблемы безопасности могут оказать сильное влияние на выбор платформы. Корпоративные поставщики, такие как Red Hat, имеют богатую историю управления безопасностью, а OpenShift добавляет защиту SELinux для безопасности контейнеров поверх защиты, уже предоставленной нативным Kubernetes.

Степень, до которой безопасность важна для вас, может сильно различаться. Мы были вовлечены в компании, где у разработчиков есть полный и надежный доступ к рабочим базам данных, а также в компании, где паранойя по поводу безопасности находится на самом высоком уровне. Эти разные уровни беспокойства приводят к совершенно разным моделям поведения при разработке и эксплуатации, а, следовательно, при выборе платформы.

Возьмем один простой пример: доверяете ли свои данные и код стандартам безопасности и продуктам Amazon Web Services (AWS)? Мы не выделяем здесь

AWS – насколько известно и испытано на себе, их стандарты безопасности обычно считаются непревзойденными в облачном пространстве. Более того, доверяете ли вы своему развитию?

Кроме того, доверяете ли вы своим командам разработчиков управление обязанностями, которые неизбежно лежат на командах, ответственных за управление приложений?

Есть достаточно историй о предоставлении закрытых данных в хранилищах AWS S3, чтобы для многих компаний это стало поводом для беспокойства.

> **ПРИМЕЧАНИЕ.** Ответственность за раскрытие данных на S3 лежит на пользователе AWS, а не на самом AWS. AWS предоставляет вам комплексные инструменты для управления безопасностью, но оно не может управлять вашими требованиями безопасности и операциями.

13.1.6. Независимость потребителей

Одним из факторов, который не часто рассматривается, является степень самоуправления, необходимая командам. В небольших организациях эта тенденция меньше, чем в более крупных. В более крупных организациях вы можете получить команды разработчиков – от высококвалифицированных, которым требуются самые современные технологические платформы, до менее квалифицированных, которым просто нужен курируемый способ развертывания простых и стабильных веб-приложений.

Различные требования могут привести к выбору разных платформ. Например, мы видели среды, в которых одно бизнес-подразделение удовлетворено централизованной, курируемой и монолитной платформой, тогда как другое бизнес-подразделение требует высокой степени контроля и предъявляет особые технические требования. Такие пользователи могут подтолкнуть вас к более индивидуальной платформе, чем те, что поставляются. Если эти пользователи готовы помочь в создании и поддержке платформы, может возникнуть продуктивное партнерство.

Если ваша организация достаточно велика и сообщество разработчиков достаточно разнородно, можете даже рассмотреть возможность использования нескольких вариантов для ваших платформ Docker.

13.1.7. Облачная стратегия

Большинство компаний, работающих с ИТ, занимают какую-либо позицию по отношению к облачным платформам.

Некоторые приняли их от всего сердца, а другие все еще начинают свой путь к ним, находятся в процессе перехода или даже возвращаются к старомодному центру обработки данных.

Принятие вашей организацией облачной платформы Docker может быть определено этой позицией. Факторы, на которые следует обратить внимание, связаны с тем, существует ли страх так называемой блокировки поставщика облачных вычислений, когда переход ваших приложений и данных из центров

обработки данных поставщиков облачных вычислений становится слишком дорогостоящим, чтобы его можно было поддерживать. От этого можно защититься, используя открытые стандарты и продукты или даже с помощью запуска существующих продуктов поверх типовых вычислительных ресурсов, предоставляемых этими поставщиками облачных вычислений (вместо того чтобы использовать их рекомендованные продукты, а иногда и продукты, специфичные для поставщиков облачных вычислений).

13.1.8. Организационная структура

Организационная структура является фундаментальной характеристикой любой компании, и она учитывает здесь все остальные факторы. Например, если команды разработчиков отделены от рабочих групп, это приводит к утверждению о необходимости стандартизированной платформы, с которой обе команды могут работать и которой могут управлять.

Точно так же, если ответственность за разные части операции распределяется по разным группам, это имеет тенденцию поддерживать частичный подход к доставке платформы. Одним из примеров этого, который мы видели, является управление реестрами Docker в более крупных организациях. Если хранилище артефактов с централизованным управлением уже существует, имеет смысл просто обновить существующее хранилище и использовать его в качестве реестра Docker (при условии, что оно поддерживает этот вариант использования). Таким образом, управление и эксплуатация хранилища обходятся дешевле, чем создание отдельного решения для того, что по сути является той же задачей.

13.1.9. Несколько платформ?

В этом месте может быть уместно упомянуть одну модель: для крупных организаций с различными потребностями возможен другой подход. У вас могут быть потребители, предпочитающие управляемые платформы, которые они, возможно, используют, а другие потребители в той же организации могут потребовать больше индивидуальных решений.

В таких случаях может иметь смысл предоставить высокопрофессиональную и более легкую в управлении платформу для первого набора пользователей, а также более гибкое и, возможно, более самоуправляемое решение для других. В одном случае, о котором мы знаем, доступны три варианта: самоуправляемый кластер Nomad, решение под управлением AWS и OpenShift.

Очевидная сложность такого подхода заключается в увеличении затрат на управление при работе с несколькими классами платформ и трудностях эффективного распространения этих вариантов в рамках всей организации.

13.1.10. Организационные факторы. Заключение

Надеемся, что эта дискуссия нашла отклик у вас и дала некоторое представление о сложности выбора подходящей платформы для Docker (или даже любой технологии) в организациях с различными потребностями. Хотя это

может показаться несколько абстрактным, следующий раздел будет намного меньше, так как мы рассмотрим конкретные проблемы, которые, возможно, нужно будет учитывать при выборе решения для вашего бизнеса. Эта дискуссия дала нам подходящие линзы для оценки данных проблем и их возможных решений.

13.2. Области, которые следует учитывать при переходе на Docker

Наконец, мы поговорим о конкретных функциональных проблемах, которые, возможно, придется решать при реализации платформы Docker.

Этот раздел состоит из трех частей:

- *Безопасность и контроль* – рассматриваются элементы, которые будут зависеть от отношения к безопасности и контролю в вашей организации;
- *Создание и доставка образов* – рассматриваются некоторые вещи, которые вам необходимо учитывать при разработке и доставке ваших изображений и рабочих нагрузок;
- *Запуск контейнеров* – учитывается, о чем нужно думать при работе с платформой.

По ходу дела мы рассмотрим конкретные современные технологии. Упоминание продукта никоим образом не подразумевает нашего одобрения, и продукты, которые мы упоминаем, не будут исчерпывающими. Программные продукты способны улучшаться и ухудшаться и могут быть заменены или объединены. Они упоминаются здесь только для иллюстрации практических последствий вашего выбора платформы.

Если многие из обсуждаемых нами вопросов кажутся неясными или неуместными для вашей организации, скорее всего, вы не станете действовать в условиях множества ограничений и, следовательно, будете иметь большую свободу действий по своему усмотрению. Если это так, можете рассматривать эту главу как предложение для понимания некоторых проблем, которые наблюдаются на крупных и регулируемых предприятиях.

13.2.1. Безопасность и контроль

Сначала мы разберемся с безопасностью, потому что во многих отношениях ваша позиция в отношении безопасности и контроля будет существенно влиять на то, как вы подходите ко всем остальным темам. Кроме того, если ваша организация меньше заботится о безопасности, чем другие, вы можете быть менее озабочены решением проблем, описанных в этом разделе.

> **ПРИМЕЧАНИЕ.** Под словом «контроль» мы подразумеваем системы управления, которые накладываются на работу команды разработчиков и управляют ее работой. Это включает в себя жизненные циклы

разработки программного обеспечения с централизованным управлением, управление лицензиями, аудит безопасности, общий аудит и так далее. Некоторые организации имеют очень легкое прикосновение, а другие более тяжелые.

Сканирование образов

Где бы вы ни хранили свои образы, есть прекрасная возможность в момент хранения проверить, что эти образы соответствуют вашим желаниям. То, что вы, возможно, хотите проверить, зависит от вашего варианта использования, но вот несколько примеров конкретных вопросов, на которые захочется получить ответы в более или менее реальном времени:

- В каких образах есть версия bash с уязвимостью Shellshock?
- Есть ли устаревшая SSL-библиотека в каком-либо образе?
- Какие образы основаны на подозрительном базовом образе?
- В каких образах есть нестандартные (или просто неправильные) библиотеки или инструменты разработки?

ПРИМЕЧАНИЕ. Shellshock – особенно серьезный набор уязвимостей в bash, обнаруженных в 2014 году. Компании по обеспечению безопасности зарегистрировали миллионы атак и проб, связанных с этой ошибкой, в дни, последовавшие за раскрытием первой из серии связанных ошибок.

На рис. 13.2 показан основной рабочий процесс сканирования образа в жизненном цикле разработки программного обеспечения. Образ собирается и помещается в реестр, после чего инициируется сканирование. Сканер может либо проверить образ на месте в реестре, либо скачать его и работать над этим. В зависимости от уровня паранойи, которую вы испытываете по отношению к образам, можете синхронно проверять образ и запрещать его использование, до тех пор пока оно не получит статус OK, или можете проверить его асинхронно и предоставить отчет пользователю.

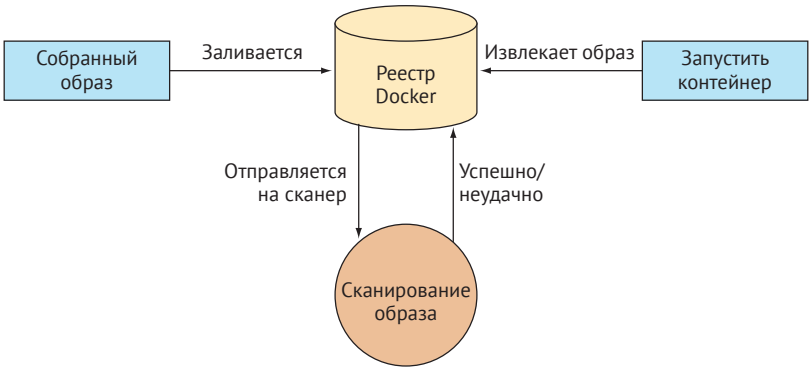


Рис. 13.2 ❖ Рабочий процесс сканирования образа

Обычно параноидальный подход применяется в отношении образов, используемых в эксплуатации, а асинхронный консультативный подход применяется в разработке.

В мире сканирования образов есть много вариантов, но не все они одинаковы. Самая важная вещь, которую нужно понять, – это то, что сканеры делятся примерно на две категории: те, которые ориентированы на установленные пакеты, и те, которые в первую очередь предназначены для глубокого сканирования программного обеспечения в образе. Примерами первых являются Clair и OpenSCAP, а примеры вторых – Black Duck Software, Twistlock, Aqua Security, Docker Inc. и многие другие. Между этими двумя категориями есть некоторое совпадение, но основная разделительная черта – это стоимость: поддерживать необходимые базы данных, чтобы быть в курсе уязвимостей в различных типах библиотек или двоичных файлах, дороже, поэтому сканеры с функцией глубокого сканирования, как правило, требуют гораздо больших затрат.

Это разделение может иметь отношение к вашему принятию решений. Если ваши образы не обладают полным доверием, можно предположить, что пользователи не являются вредоносными, и использовать более простой сканер пакетов. Это даст вам показатели и информацию о стандартных пакетах и их соответствующем уровне риска без особых затрат.

Хотя сканеры могут снизить риск наличия вредоносного или нежелательного программного обеспечения в ваших образах, это не волшебные пули. Наш опыт их оценки показывает, что даже лучшие сканеры не идеальны, и что они лучше выявляют проблемы с некоторыми типами двоичных файлов или библиотек, чем другие. Например, некоторые могут более успешно идентифицировать проблемы пакета npm, чем, скажем, сканеры, написанные на C ++, или наоборот. Смотрите метод 94 в главе 14, где приводится образ, который мы использовали для тренировки и тестирования этих сканеров.

Еще одна вещь, о которой следует знать, – это то, что, хотя сканеры могут работать с неизменяемыми образами и проверять их статическое содержимое, все еще существует высокий риск того, что контейнеры могут создавать и запускать вредоносное программное обеспечение во время выполнения. Статический анализ образов не может решить эту проблему, поэтому вам может потребоваться рассмотреть и контроль за средой выполнения.

Как и во всех темах этого раздела, вы должны думать о том, чего необходимо достичь при выборе сканера. Возможно, вы захотите:

- запретить злоумышленникам вставлять объекты в ваши сборки;
- обеспечить соблюдение общекорпоративных стандартов использования программного обеспечения;
- быстро исправлять известные и стандартные базы данных общеизвестных уязвимостей информационной безопасности (CVE).

ПРИМЕЧАНИЕ. CVE (Common Vulnerabilities and Exposures) – это идентификатор уязвимости программного обеспечения, позволяющий широко и недвусмысленно идентифицировать конкретные неисправности.

Наконец, вы также можете рассмотреть стоимость интеграции этого инструментального средства в ваш конвейер DevOps. Если найдете сканер, которым вы довольны, и он хорошо интегрирован с вашей платформой (или другими соответствующими инструментами DevOps), это может стать еще одним фактором в его пользу.

Целостность образов

Целостность образов и их сканирование часто путают, но это не одно и то же. В то время как *сканирование образа* определяет, что находится *в* образе, *целостность образа* гарантирует, что то, что *получено* из реестра Docker, совпадает с тем, что было безопасно размещено там. (*Проверка образа* является еще одним распространенным способом описания этого требования.)

Представьте себе следующий сценарий: Элис помещает образ в репозиторий (образ A), и, после того как он проходит какой-либо обязательный процесс проверки, Боб хочет запустить этот образ на сервере. Боб запрашивает образ A с сервера, но неизвестный ему злоумышленник (Кэрол) взломал сеть и поместил прокси между Бобом и реестром. Когда Боб скачивает образ, ему фактически передают вредоносный образ (образ C), который будет запускать код, пересылающий конфиденциальные данные на сторонний IP-адрес вне сети (см. рис. 13.3).

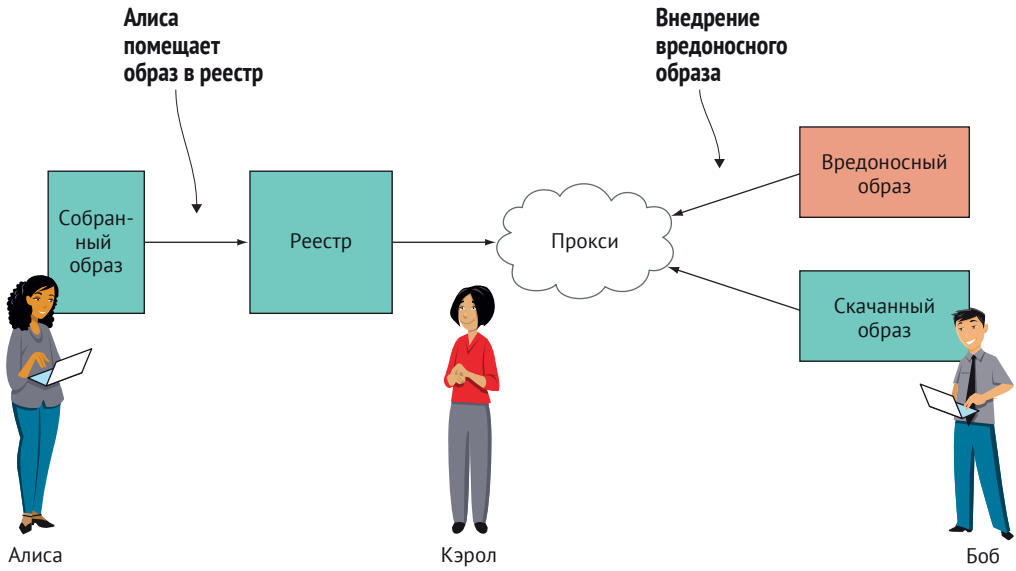


Рис. 13.3 ❖ Нарушение целостности образа

Возникает вопрос: когда скачивается образ Docker, как можно быть уверенным, что это тот образ, который вы просили? Уверенность в этом – то, к чему обращается целостность образа.

Docker Inc. привела сюда свой продукт Content Trust, также известный как Notarius. Он подписывает образ с помощью закрытого ключа, который гарантирует, что при расшифровке содержимого с помощью открытого ключа содержимое будет таким же, как было загружено в реестр.

Content Trust предлагает дополнительную функциональность при делегировании ключевых обязанностей, которую мы не будем здесь рассматривать.

Помимо предложения Docker, с 2018 года сообщать практически не о чем, что является своего рода данью их руководству в этой области. Ведущие продукты, такие как Kubernetes и OpenShift, предлагают очень мало в этой области из коробки, поэтому, если вы не покупаете продукты Docker, возможно, придется интегрировать их самостоятельно. Для многих организаций такие попытки не стоят усилий, поэтому они будут полагаться на существующие средства защиты (вероятно, периметра).

Если вам удастся внедрить решение для обеспечения целостности образов, все равно нужно подумать, как в вашей организации будет вестись управление ключами. Организации, которые заботятся о том, чтобы продвинуться так далеко, вероятно, разработают для этого политику и решения.

Сторонние образы

Продолжая тему образов, другой распространенной проблемой при предоставлении платформы является то, как вы собираетесь подходить к теме внешних образов. Опять же, основная трудность здесь заключается в доверии: если у вас есть поставщик, который хочет перенести образ Docker на вашу платформу, как вы можете быть уверены в его безопасности? Это особенно важный вопрос в мультиарендной среде, где разные группы (которые не обязательно доверяют друг другу) должны запускать контейнеры на одних и тех же хостах.

Один из подходов состоит в том, чтобы просто запретить все сторонние образы и разрешить их создание только из известных и рекомендованных базовых образов с использованием кода и артефактов, хранящихся в корпоративной сети. Некоторые образы поставщиков могут работать в этом режиме. Если образ поставщика – это, по сути, JAR- файл (архив Java), работающий на стандартной виртуальной машине Java, образ можно воссоздать и собрать в сети из этого артефакта и запустить его в утвержденном образе виртуальной машины.

Однако не все образы или поставщики будут подвержены такому подходу, и это неизбежно. Если давление с целью разрешить сторонние образы достаточно сильно (и по нашему опыту это так), у вас есть несколько вариантов:

- довериться своему сканеру;
- изучить образ на глаз;
- сделать так, чтобы команда, привносящая образ в организацию, отвечала за его управление.

Маловероятно, что вы будете полностью доверять своему сканеру, чтобы обеспечить себе достаточную уверенность в безопасности стороннего образа без полного его встраивания с течением времени, поэтому ответственность, возможно, придется возложить на что-то еще.

Второй вариант, проверка образов вручную, не масштабируется и подвержен ошибкам. Последний вариант является самым простым и легким для реализации.

Мы видели среды, в которых используются все три подхода, когда команда управления платформой проверяет работоспособность образа, но окончательную ответственность за него несет команда управления приложениями. Часто существует процесс доставки образов виртуальных машин в организацию, поэтому можно просто скопировать эту процедуру для образов Docker. Здесь следует отметить одно ключевое отличие: хотя виртуальные машины являются мультиарендными в том смысле, что они совместно используют гипервизор вместе со своими «арендаторами», образы Docker применяют полнофункциональную операционную систему, которая дает атакам гораздо большую площадь поверхности (подробнее об этом см. главу 14, посвященную безопасности).

Еще один вариант – песочница для запуска образов в их собственной аппаратной среде, например, с помощью маркировки узлов Kubernetes в кластере, или с использованием отдельных экземпляров облачных продуктов, таких как ECS, или запуска совершенно отдельной платформы на отдельном оборудовании или даже в сетях.

Секреты

Каким-то образом (и особенно когда вы приступаете к работе), конфиденциальной информацией нужно будет управлять безопасным способом. Конфиденциальная информация включает в себя файлы или данные, переданные в сборки, такие как:

- SSL-ключи;
- комбинации имени пользователя и пароля;
- данные, идентифицирующие клиента.

Эта передача секретных данных в жизненный цикл программного обеспечения может быть выполнена в нескольких точках. Один из подходов заключается во встраивании секретов в ваши образы во время сборки. Такой подход не одобряется, поскольку он распространяет конфиденциальные данные, куда бы ни направлялся образ.

Более одобренный метод состоит в том, чтобы платформа помещала секреты в ваши контейнеры во время выполнения. Есть разные способы сделать это, но имеются несколько вопросов, на которые нужно ответить:

- Зашифрован ли секрет при хранении?
- Зашифрован ли секрет при передаче?
- У кого есть доступ к секрету (в хранилище или во время выполнения в контейнере)?

- Как секрет предоставляется внутри контейнера?
- Можете ли вы отслеживать или проверять, кто видел или использовал секрет?

Kubernetes обладает так называемой способностью «секретов». Многих удивляет, что они хранятся в виде обычного текста в постоянном хранилище (база данных etcd). Технически они закодированы в base64, но с точки зрения безопасности это обычный текст (он не зашифрован и легко реверсируется). Если кто-то унесет диск, содержащий эту информацию, он может без труда получить доступ к этим секретам.

В его нынешнем виде существуют проверенные на практике реализации приложений, такие как хранилище HashiCorp, для интеграции с Kubernetes. Docker Swarm имеет более надежную поддержку секретов из коробки, но Docker Inc., похоже, в конце 2017 года присоединилась к Kubernetes.

Аудит

При занятости в рабочей среде (или в любой другой чувствительной среде) это может стать ключом к демонстрации того, что у вас есть контроль над тем, кто и какую команду выполнял. Это то, что может быть неочевидным для разработчиков, которые не так озабочены восстановлением этой информации.

Причины этой «корневой» проблемы описаны в главе 14, но их можно кратко рассмотреть здесь, сказав, что предоставление пользователям эффективного доступа к сокету Docker дает им корневой контроль над всем хостом. Во многих организациях это запрещено, поэтому доступ к Docker обычно должен быть как минимум отслеживаемым.

Вот некоторые из вопросов, на которые вам может потребоваться ответить:

- Кто (или что) может выполнить команду `docker`?
- Какой у вас контроль над тем, кто ее выполняет?
- Какой у вас контроль над тем, что запускается?

Решения для этой проблемы существуют, но они относительно новы и, как правило, являются частью других более крупных решений. OpenShift, например, стал лидером, добавив в Kubernetes надежное управление доступом на основе ролей. У облачных провайдеров обычно есть более облачные способы достижения такого контроля посредством (в случае с AWS) использования ролей управления идентификационными данными и доступом или аналогичных функций, встроенных в ECS или EKS.

Инструментальные средства обеспечения безопасности контейнеров, предоставляемые такими поставщиками, как Twistlock и Aqua Security, предлагают средства для управления тем, какие конкретные подкоманды и флаги Docker кем могут запускаться, обычно путем добавления промежуточного сокета или другого прокси-сервера между вами и сокетом Docker, который может обеспечивать доступ в команды Docker.

С точки зрения записи, кто что сделал, нативная функциональность медленно появлялась в таких продуктах, как OpenShift, но сейчас она есть. Если

вы посмотрите на другие продукты, не думайте, что такая функциональность полностью реализована!

Контроль за средой выполнения

Контроль за средой выполнения можно рассматривать как аудит на более высоком уровне. Регулируемые предприятия, вероятно, захотят иметь возможность определять, что происходит на всей их территории, и сообщать об этом. Вывод таких отчетов можно сравнить с существующей базой данных управления конфигурацией, чтобы увидеть аномалии или текущие рабочие нагрузки, которые не могут быть учтены.

На этом уровне вас просят ответить на следующие вопросы:

- Откуда вы знаете, что запущено?
- Можете ли вы сопоставить это содержимое с вашим реестром/реестрами и/или вашей базой данных управления конфигурацией?
- Какие-либо контейнеры меняли критические файлы с момента запуска?

Опять же, это сопряжено с некоторыми другими продуктами, которые могут стать частью вашей стратегии Docker, так что следите за ними. Или это может быть побочным эффектом вашей общей стратегии развертывания приложений и сетевой архитектуры. Например, если вы создаете и запускаете контейнеры в Amazon VPC, установить и сообщить, что в них, – относительно банальная проблема, которую можно решить.

Еще один распространенный сильный аргумент в этой области – обнаружение аномалий. Разработки в сфере обеспечения ИТ-безопасности предлагают новомодные решения для машинного обучения. Они изучают, что должен делать контейнер, и предупреждают вас, если им кажется, что он совершает что-то необычное, например, подключается к порту постороннего приложения.

Звучит замечательно, но вам нужно подумать о том, как это будет работать в оперативном режиме. Вы можете получить много ложных срабатываний, и их нужно тщательно курировать – у вас есть для этого ресурсы? Вообще говоря, чем крупнее организация и чем больше она заботится о безопасности, тем больше она должна быть этим обеспокоена.

Расследование киберпреступлений

Это напоминает аудит, но действует гораздо более целенаправленно. Когда происходит инцидент в области безопасности, различные стороны захотят узнать, что произошло. В старом мире физических лиц и виртуальных машин было много мер предосторожности для содействия расследованию после инцидента. Агенты и процессы-наблюдатели всех мастей могли работать в ОС, а «прослушки» (taps) могли быть размещены на сетевом или даже аппаратном уровне.

Вот некоторые из вопросов, на которые экспертная группа могла бы получить ответы, после того как произошел инцидент в области безопасности:

- Можете ли вы сказать, кто запустил контейнер?
- Можете ли вы сказать, кто собрал контейнер?
- Можете ли определить, что сделал контейнер, когда он был удален?
- Можете ли определить, что контейнер мог сделать, после того как он был удален?

В этом контексте вам потребуется использование определенных решений для ведения журналов, чтобы гарантировать, что информация о системной активности сохраняется в инстанцированиях контейнеров.

Sysdig и ее инструмент Falco (с открытым исходным кодом) – еще один интересный и многообещающий продукт в этой области. Если вы знакомы с tcpdump, этот инструмент выглядит очень похоже, позволяя запрашивать системные вызовы в полете. Вот пример такого правила:

```
container.id != host and proc.name = bash
```

Соответствует, если оболочка bash запущена в контейнере.

Коммерческое предложение Sysdig выходит за рамки мониторинга и позволяет вам предпринимать действия, основанные на отслеживаемом поведении, в отношении ваших определенных наборов правил.

13.2.2. Создание и доставка образов

Разобравшись с безопасностью, мы переходим к сборке и доставке. В этом разделе рассказывается о том, что может понадобиться при создании и распространении ваших образов.

Когда дело доходит до сборки образов, есть несколько областей, которые можно рассмотреть.

Во-первых, хотя файлы Dockerfile являются стандартом, существуют и другие методы создания образов (см. главу 7), поэтому потребуется установить стандарт, если различные способы могут вызвать путаницу или несовместимы друг с другом. У вас также может быть инструмент стратегического управления конфигурацией, который вы захотите интегрировать в стандартное развертывание ОС.

Наш реальный опыт показывает, что подход с использованием файлов Dockerfile глубоко укоренился и популярен среди разработчиков. Затраты на изучение более сложного инструментального средства конфигурационного управления, чтобы соответствовать стандартам компании для виртуальных машин, зачастую не то, на что у разработчиков есть время или желание. Для удобства или повторного использования кода более широко применяются такие методы, как S2I или Chef/Puppet/Ansible. Поддержка файлов Dockerfile гарантирует, что вы получите меньше вопросов и откликов от сообщества разработчиков.

Во-вторых, в чувствительных средах вы вряд ли захотите, чтобы сборка образов была открыта для всех пользователей, поскольку образам могут доверять другие команды внутри или снаружи.

Сборка может быть ограничена соответствующей маркировкой или продвижением образов (см. ниже) или контролем доступа на основе ролей.

В-третьих, стоит подумать об опыте разработчика. По соображениям безопасности не всегда возможно предоставить пользователям открытый доступ для загрузки образов Docker из общедоступных репозиториев или даже возможность запуска инструментов Docker в их локальной среде (см. главу 14). Если это так, есть несколько вариантов, которые вы можете использовать:

1. Получение одобрения на стандартный инструментарий. Это может дорого стоить, а иногда и слишком дорого из-за проблем безопасности и требований бизнеса.
2. Создание одноразовой песочницы, в которой можно собирать образы Docker. Если виртуальная машина временная, заблокирована и подвергнута строгому аудиту, многие проблемы безопасности значительно облегчаются.
3. Предложение удаленного доступа к вышеупомянутой изолированной программной среде посредством любого клиента Docker (но учтите, что это необязательно значительно уменьшает многие поверхности атаки).

В-четвертых, также стоит задуматься о совместимости опыта разработчика при развертывании приложения. Например, если разработчики используют docker-compose на своих ноутбуках или в тестовых средах, они могут отказаться от перехода к развертыванию Kubernetes в рабочей среде. (Со временем этот последний момент оказывается все более спорным, по мере того как Kubernetes становится стандартом.)

Реестр

Теперь должно быть очевидно, что вам понадобится реестр. Существует пример с открытым исходным кодом, Docker Distribution, но он больше не является доминирующим выбором, главным образом потому, что реестр Docker – это реализация хорошо известного API. Сейчас есть множество предложений, если вы хотите платить за корпоративный реестр или запустить открытый исходный код самостоятельно.

Docker Distribution входит в состав продукта Data Center, который обладает рядом привлекательных функций (таких как Content Trust).

Какой бы продукт вы ни выбрали, есть несколько менее очевидных моментов, которые следует учитывать:

- Этот реестр хорошо ведет себя с вашей системой аутентификации?
- Обладает ли он управлением доступом на основе ролей (RBAC)?

Аутентификация и авторизация очень важны для предприятий. Быстрое и дешевое бесплатное решение для реестра будет работать в процессе разработки, но, если у вас есть стандарты безопасности или стандарты RBAC, которые необходимо поддерживать, эти требования окажутся на вершине вашего списка.

Некоторые инструментальные средства имеют менее детализированные функции RBAC, и это может оказаться серьезным пробелом.

- *Есть ли у него средства для продвижения (promotion) образов?* – Образы не созданы одинаковыми. Некоторые из них «сляпаны» наспех в ходе экспериментов разработчиков, в которых корректность не требуется, тогда как другие предназначены для надежной эксплуатации. Рабочие процессы вашей организации могут потребовать, чтобы вы различали два этих типа, и реестр может помочь вам в этом, управляя процессом через отдельные экземпляры или через ворота, навязанные метками;
- *Хорошо ли это согласуется с другими вашими хранилищами артефактов?* – У вас, вероятно, уже есть хранилище артефактов для TAR-файлов, внутренних пакетов и тому подобного. В идеальном мире ваш реестр будет просто функцией внутри него. Если это не вариант, затраты на интеграцию или управление обойдутся вам в круглую сумму, о чем нужно знать.

Базовые образы

Если вы думаете о стандартах, базовый образ (или образы), который используют команды, может потребовать некоторого рассмотрения.

Во-первых, какой корневой образ вы хотите использовать и что в него должно входить? Обычно организации имеют стандартный дистрибутив Linux, который они предпочитают использовать. Если так, то это, вероятно, будет указано в качестве базы.

Во-вторых, как вы будете создавать и поддерживать эти образы? В случае обнаружения уязвимости, кто (или что) несет ответственность за определение того, затронуты ли вы или какие образы затронуты? Кто несет ответственность за исправление пострадавшего пространства?

В-третьих, что должно войти в этот базовый образ? Есть ли общий набор инструментов, который понадобится всем пользователям, или вы хотите оставить это на усмотрение отдельных команд? Хотите разделить эти требования на отдельные подобразы?

В-четвертых, как будут восстановлены эти образы и подобразы? Обычно необходимо создать какой-либо конвейер. Как правило, для автоматического построения базового образа (и, как следствие этого, любых подобразов) при срабатывании какого-либо пускового механизма будет использоваться средство непрерывной интеграции, например Jenkins.

Если вы несете ответственность за базовый образ, приходится задумываться над его размером. Часто утверждают, что тонкие образы лучше. В некоторых отношениях (например, в отношении безопасности) это допустимо, но данная «проблема» чаще воображаемая, нежели реальная, особенно в отношении производительности. Парадоксальная природа этой ситуации обсуждается в методе 60.

Жизненный цикл разработки программного обеспечения

Жизненный цикл разработки программного обеспечения – это определенный процесс получения, создания, тестирования, развертывания и утилизации программного обеспечения. В своем идеальном состоянии он помогает снизить неэффективность, обеспечивая постоянную оценку, закупку и использование программного обеспечения в рамках группы с общим интересом к объединению ресурсов.

Если у вас уже есть процессы жизненного цикла разработки программного обеспечения, как туда вписывается Docker? Можно вступить в философские дискуссии о том, является ли контейнер Docker пакетом (например, rpm) или целым дистрибутивом Linux (поскольку его содержимое, вероятно, находится под контролем разработчика). В любом случае ключевым предметом спора обычно является право собственности. Кто за что отвечает в образе? Именно здесь вступает в действие многоуровневая файловая система Docker (см. главу 1). Поскольку кто что создал в конечном образе – полностью подконтрольно (при условии, что контент является доверенным), то отследить, кто за какую часть программного стека отвечает, относительно просто.

Как только ответственность определена, вы можете рассмотреть, как будут обрабатываться исправления:

- *Как определить, какие образы необходимо обновить?* – Здесь может помочь сканер или любое инструментальное средство, которое способно идентифицировать файлы по артефактам, представляющим интерес.
- *Как вы их обновляете?* – Некоторые платформы позволяют запускать повторные сборки и развертывания контейнеров (такие как OpenShift или, возможно, ваш конвейер, созданный вручную).
- *Как вы сообщаете командам о необходимости обновления?* – Достаточно ли сообщения по электронной почте, или вам нужно идентифицируемое лицо в качестве владельца? Опять же, ваша корпоративная политика, вероятно, будет здесь вашим гидом. Существующие политики должны существовать для более традиционного развернутого программного обеспечения.

Ключевым моментом в этом новом мире является то, что число команд, ответственных за контейнеры, может быть выше, чем в прошлом, и количество контейнеров, которые нужно оценить или обновить, также может быть значительно выше. Все это станет тяжелым бременем для ваших групп обслуживания инфраструктуры, если в поставках программного обеспечения не будет процессов для решения этой проблемы. В крайнем случае вы будете вынуждены заставить пользователей выполнить обновления, добавляя слои к их образам, если они не встают в очередь. Это особенно важно, если вы используете общую платформу. Можно было бы даже рассмотреть возможность использования инструментов оркестровки для размещения «непослушных» контейнеров на конкретных изолированных хостах, чтобы снизить риск. Обычно эти вещи считаются слишком запоздалыми, и ответ должен быть импровизированным.

13.2.3. Запуск контейнеров

Теперь посмотрим на запуск контейнеров. Во многих отношениях он мало чем отличается от запуска отдельных процессов, но внедрение Docker может принести свои собственные проблемы, а изменения поведения, которые обеспечивает Docker, могут также заставить задуматься о других аспектах вашей инфраструктуры.

Операционная система

Операционная система, которую вы используете, может стать важной при запуске платформы Docker. Корпоративные операционные системы могут отставать от последних и лучших версий ядра, и, как вы увидите в главе 16, используемая версия ядра может иметь большое значение для вашего приложения. Традиционно Docker был очень быстро меняющейся кодовой базой, и не все рекомендованные ОС были в состоянии поддерживать его (версия 1.10 была для нас особенно болезненным переходом со значительными изменениями в формате хранения образов). Стоит проверить, какие версии Docker (и связанные с ним технологии, такие как Kubernetes) доступны вам в менеджерах пакетов, прежде чем вы пообещаете поставщикам, что их приложения будут работать в вашем кластере Kubernetes.

Общее хранилище

Как только ваши пользователи начнут развертывать приложения, первое, о чем они будут беспокоиться, – это то, куда уходят их данные. В основе Docker лежит использование томов (см. главу 5), которые не зависят от работающих контейнеров.

Эти тома могут поддерживаться многочисленными типами хранилищ, монтируемыми локально или удаленно, но ключевой момент заключается в том, что хранилище может совместно использоваться несколькими контейнерами, что делает его идеальным для запуска баз данных, которые сохраняются в циклах контейнеров.

- *Легко ли предоставить общее хранилище?* – Общее хранилище может быть дорогим в обслуживании и предоставлении как с точки зрения необходимой инфраструктуры, так и с учетом почасовой оплаты. Во многих организациях предоставление хранилища – это не просто вызов API и ожидание в течение нескольких секунд, как это может быть у облачных провайдеров, таких как AWS.
- *Готова ли поддержка общего хранилища к повышенному спросу?* – Поскольку развертывание контейнеров Docker и новых сред для разработки или тестирования настолько просто, спрос на общее хранилище может значительно возрасти. Стоит подумать, готовы ли вы к этому.
- *Доступно ли общее хранилище в местах развертывания?* – У вас может быть несколько центров обработки данных или облачных провайдеров или даже их сочетание. Могут ли все эти места без проблем общаться друг с другом? Выполняют ли они это требование или нет? Нормативные

ограничения и желание предоставить возможности вашим разработчикам могут добавить вам работы.

Сетевая среда

Что касается работы сетевой среды, то при реализации платформы Docker вам, возможно, придется подумать о нескольких вещах.

Как видно из главы 10, по умолчанию у каждого контейнера Docker есть свой собственный IP-адрес, выделенный из зарезервированного набора IP-адресов. При введении продукта, который управляет работой контейнеров в вашей сети, могут быть зарезервированы другие наборы сетевых адресов. Например, сервисный слой Kubernetes использует набор сетевых адресов для поддержки и маршрутизации к стабильным конечным точкам в кластере узлов.

Некоторые организации резервируют диапазоны IP-адресов для своих собственных целей, поэтому вам следует опасаться конфликтов. Если диапазон IP-адресов зарезервирован для определенного набора баз данных, например, приложения, которые используют диапазон IP-адресов в вашем кластере для своих контейнеров или служб, могут захватить эти IP-адреса и запретить другим приложениям в кластере получать доступ к этому набору баз данных. Трафик, предназначенный для баз данных, в конечном итоге будет направляться внутри кластера на IP-адреса контейнера или службы.

Производительность сети также может стать значительной. Если у вас уже есть программно-определяемые сети (такие как Nuage или Calico), расположенные поверх вашей сети, добавление дополнительных сетей для платформ Docker (таких как OpenVSwitch или даже еще один слой Calico) может заметно снизить производительность.

Контейнеры также могут влиять на работу сети так, как вы этого не ожидаете. Многие приложения традиционно используют стабильный исходный IP-адрес как часть аутентификации для внешних служб. Однако в мире контейнеров IP-адрес источника, представленный из контейнера, может быть либо IP-адресом контейнера, либо IP-адресом хоста, на котором работает контейнер (выполняющий преобразование сетевых адресов обратно в контейнер). Кроме того, если он поступает из кластера хостов, нет гарантии, что представленный IP-адрес стабилен. Существуют способы обеспечения стабильности представления IP-адресов, но обычно они требуют определенных усилий по разработке и внедрению.

Балансировка нагрузки – это еще одна область, которая потенциально требует больших усилий. Здесь так много есть что рассказать, что она может стать темой для другой книги, но вот краткий список:

- Какой продукт является предпочтительным/стандартным (например, NGinx, F5s, HAProxy, HTTPD)?
- Как и где вы будете обрабатывать SSL-терминацию?
- Вам нужно решение для взаимной аутентификации TLS?
- Как будут генерироваться и управляться сертификаты на вашей платформе?

- Влияет ли ваш балансировщик нагрузки на заголовки таким образом, чтобы это соответствовало другим приложениям в организации (будьте готовы выполнить здесь множество отладок, если это не так)?

Наконец, если вы используете облачного провайдера в дополнение к каким-либо центрам обработки данных, которыми вы уже владеете или пользуетесь, возможно, придется подумать о том, будут ли пользователи подключаться к локальным услугам, предоставляемым облачным провайдером, и каким образом.

Ведение журналов

Практически каждое приложение будет иметь связанные с ним файлы журналов. Большинству приложений требуется постоянный доступ к этим журналам (особенно в рабочей среде), поэтому обычно необходима какая-либо централизованная служба ведения журналов. Поскольку контейнеры эфемерны (в то время как виртуальные машины и выделенные серверы обычно таковыми не являются), такие данные могут быть потеряны, если контейнер умирает, и журналы хранятся в его файловой системе. По этим причинам переход в контейнеризованный мир может вывести проблему журналирования на первый план.

Поскольку ведение журналов – это настолько основной и распространенный элемент функциональности приложения, часто имеет смысл централизовать и стандартизировать его. Контейнеры могут предоставить возможность сделать это.

Мониторинг

Большинство приложений необходимо будет контролировать в большей или меньшей степени, и существует множество поставщиков и продуктов, связанных с мониторингом контейнеров. Это все еще развивающаяся область.

Prometheus – один из продуктов, который имеет большой потенциал в области Docker. Первоначально разработанный компанией SoundCloud, он со временем приобрел популярность, особенно с тех пор, как стал частью Cloud Native Computing Foundation.

Поскольку контейнеры и виртуальные или физические машины не одно и то же, не факт, что традиционные средства мониторинга будут хорошо работать внутри контейнеров в качестве расширений или на хосте, если они не знают о контейнерах.

Тем не менее, если вы используете кластер хостов и необходимо их обслуживать, вам пригодятся традиционные, проверенные, зрелые средства мониторинга. Скорее всего, на них в значительной степени будут полагаться, когда вы попытаетесь выжать максимальную производительность из вашего кластера для конечных пользователей. Это предполагает, что платформа является успешной. Наш опыт подсказывает, что часто спрос намного превышает предложение.

13.3. Поставщики, организации и продукты

Недостатка в компаниях и организациях, желающих заработать с Docker, нет. Здесь мы рассмотрим самых крупных и значимых игроков в 2018 году и попытаемся описать, на что направлены их усилия и как их продукты могут работать на вас.

13.3.1. Cloud Native Computing Foundation (CNCF)

Первая из этих организаций отличается тем, что это не компания, а, пожалуй, самый влиятельный игрок в этой сфере. Фонд CNCF был создан в 2015 году для продвижения общих стандартов в области контейнерных технологий. Среди членов-учредителей:

- Google;
- IBM;
- Twitter;
- Docker;
- Intel;
- VMWare;
- Cisco.

Его создание совпало с выходом Kubernetes версии 1.0, который Google передал в дар CNCF (хотя он уже был предоставлен в качестве ПО с открытым исходным кодом компанией Google).

Роль CNCF в контейнерном пространстве – это роль влиятельного лица. Поскольку коллективная мощь различных вовлеченных игроков настолько велика, то, когда CNCF отстает от какой-либо технологии, вы знаете о ней две вещи: она будет иметь инвестиции и поддержку, и маловероятно, что предпочтению будет отдано одному поставщику вместо другого. Последний фактор особенно важен для потребителя платформы Docker, поскольку это означает, что ваш выбор технологии вряд ли устареет в обозримом будущем.

Существует длинный (и растущий) список технологий, которые были одобрены CNCF. Мы рассмотрим некоторые наиболее значимые:

- Kubernetes;
- Envoy;
- CNI;
- Notary;
- Containerd;
- Prometheus.

Kubernetes

Kubernetes был основополагающей и наиболее значимой технологией, которая является частью CNCF. Он был пожертвован Google сообществу, сначала как ПО с открытым исходным кодом, а затем CNCF.

Несмотря на то что это ПО с открытым исходным кодом, его пожертвование сообществу является частью стратегии Google по коммерциализации

облачных технологий и упрощению перехода потребителей от других облачных провайдеров, наиболее доминирующим из которых является AWS.

Kubernetes – это базовая технология большинства платформ Docker, прежде всего OpenShift, но также и Rancher, и даже Docker Datacenter от компании Docker Inc., потому что они поддерживают Kubernetes в дополнение к Swarm.

CNI

CNI расшифровывается как Container Network Interface. Этот проект предоставляет стандартный интерфейс для управления сетевыми интерфейсами для контейнеров. Как вы видели в главе 10, сетевая среда может быть сложной областью для управления контейнерами, и данный проект является попыткой сделать это управление проще.

Вот (очень) простой пример, который определяет loopback-интерфейс:

```
{
    "cniVersion": "0.2.0",
    "type": "loopback"
}
```

Этот файл может быть помещен в файл /etc/cni/net.d/99-loopback.conf и использован для настройки сетевого интерфейса loopback.

Более сложные примеры доступны в репозитории Git здесь: https://github.com/containernetworking/cni.

Containerd

Containerd – это версия сообщества демона Docker. Она управляет жизненным циклом контейнеров. Runc – его родственный проект. Это среда выполнения, ответственная за запуск самого контейнера.

Envoy

Первоначально созданный в компании Lyft, чтобы перевести свою архитектуру с монолитной на микросервисы, Envoy – это высокопроизводительный открытый и служебный прокси-сервер с открытым исходным кодом, который делает сеть прозрачной для приложений.

Он позволяет осуществлять непосредственное управление ключевыми сетевыми и интеграционными задачами, такими как балансировка нагрузки, проксирование и распределенная трассировка.

Notary

Notary – это инструментальное средство, изначально разработанное и созданное компанией Docker Inc. для подписи и проверки целостности образов контейнеров. (Пожалуйста, обратитесь к стр. 317, раздел «Целостность образов».)

Prometheus

Prometheus – это средство мониторинга, которое прекрасно работает с контейнерами. Оно набирает обороты в сообществе, например, благодаря переходу Red Hat с Hawkular на Prometheus на их платформе OpenShift.

13.3.2. Docker, Inc

Docker, Inc. – это коммерческая организация, которая стремится получить прибыль от проекта с открытым исходным кодом Docker.

> **ПРИМЕЧАНИЕ.** Этот проект был переименован в Moby компанией Docker Inc. в попытке зарезервировать имя Docker для получения прибыли. Пока это имя не завоевало популярность, поэтому в этой книге Moby практически не упоминается.

Как и следовало ожидать, Docker Inc. была ранним лидером в области продуктов Docker. Компания собрала несколько своих продуктов в монолитный продукт под названием Docker Datacenter. Он включал в себя поддержку, интеграцию и функции Notary, реестра, Swarm и нескольких других проектов, исходные коды которых открыл Docker. Сейчас и поддержка Kubernetes не за горами.

Из-за того, что Docker рано пришел на вечеринку, а его техническая репутация была сильной на раннем этапе, их продукт был очень убедительным по методу «быстро добраться до эксплуатации». Со временем продукт Docker утратил свои позиции, так как его догнали другие. Бизнес-модель Docker было трудно продать из-за стратегии «бери все или уходи» и модели «цена за сервер», из-за чего клиенты попадают в сильную зависимость от одного поставщика, что может заставить их выкупать платформу Docker целиком.

13.3.3. Google

Kubernetes был создан компанией Google в 2014 году после того, как Docker стал популярен. Он был призван донести принципы внутренней контейнерной платформы Google (Borg) до более широкой аудитории.

Примерно в то же время появился сервис Google Cloud. Продвижение Kubernetes было частью их облачной стратегии. (Пожалуйста, обратитесь к стр. 327, раздел «Kubernetes».)

У Google есть платный сервис для управления кластерами Kubernetes, под названием Google Kubernetes Engine (GKE), аналогичный EKS AWS.

Облачное предложение Google является для них ключевым бизнес-приоритетом, а поддержка Kubernetes – центральная часть этой стратегии.

13.3.4. Microsoft

Microsoft сотрудничает с Docker по нескольким направлениям, и все это с целью расширения облачного предложения Azure.

Во-первых, Microsoft реализовала Docker API для контейнеров изначально на платформе Windows, начиная с Windows 10 и далее. Это позволяет создавать и запускать контейнеры Windows. Планируется поддержка Kubernetes для узлов Windows, но на момент написания этой главы она все еще находится на ранней стадии.

Во-вторых, Microsoft работала над предложением своей платформы .NET под названием Dotnet Core (или .NET Core, если предпочитаете), которая

обеспечивает поддержку кодовых баз .NET в Linux. Поддерживаются не все библиотеки .NET, поэтому перемещение вашего приложения Windows далеко не тривиально (пока), но многие организации будут заинтересованы в возможности запуска своего кода Windows на платформе Linux и даже в возможности сборки с нуля для запуска на любой платформе.

В-третьих, существует предложение Azure для Kubernetes (AKS), также похожее на AWS EKS и Google Cloud GKE.

Все эти усилия можно рассматривать как предназначенные для поощрения пользователей переходить в облако Azure. Возможность запуска аналогичных рабочих нагрузок в Windows или Linux (или даже в обеих ОС) привлекательна для многих организаций. Это особенно верно, если данные уже находятся в центрах обработки данных. Кроме того, Microsoft может предложить привлекательные пакеты лицензий организациям, которые уже вложили значительные средства в технологии Microsoft, стремящиеся перейти в облако.

13.3.5. Amazon

У Amazon сейчас есть несколько контейнерных предложений, но, возможно, он немного опоздал на вечеринку. Его первым предложением была служба Elastic Container Service (ECS), которая использовала Mesos для управления развертыванием контейнеров и их хостов.

Поначалу было интересно, но вскоре по популярности ее обошел Kubernetes. В конце 2017 года в ответ Amazon объявил о сервисе Elastic Kubernetes Service (EKS), который (подобно упомянутым ранее сервисам GKE и AKS) является рекомендуемым сервисом Kubernetes. ECS все еще поддерживается, но кажется естественным думать, что EKS будет более стратегическим для них. В конце 2017 года также было объявлено о Fargate, сервисе, который запускает контейнеры без необходимости управлять какими-либо экземплярами EC2.

Все эти сервисы обеспечивают тесную интеграцию с другими сервисами AWS, что очень удобно, если вы рассматриваете AWS как долгосрочную платформу для вашего программного обеспечения. Очевидно, что коммерческая цель AWS состоит в том, чтобы вы продолжали оплачивать их услуги, но их широкая поддержка API Kubernetes может дать потребителям некоторое утешение, что связи с платформой AWS могут быть менее сильными по сравнению с другими сервисами.

13.3.6. Red Hat

Коммерческая стратегия Red Hat заключается в том, чтобы курировать, поддерживать и управлять основным программным обеспечением для своих клиентов, так называемая стратегия «сомелье с открытым исходным кодом». Red Hat отличается от других коммерческих игроков тем, что у них нет универсального облачного сервиса для пользователей (хотя OpenShift online можно рассматривать как облачное предложение, поскольку это внешний сервис).

Контейнер Red Hat сосредоточен на двух областях. Первая – OpenShift, представляет собой продукт, охватывающий Kubernetes, который можно запускать

и поддерживать в различных средах, таких как локальная работа с поставщиками облачных услуг, упомянутыми здесь (а также некоторыми другими), и как сервис с OpenShift Online от компании Red Hat.

Разработка OpenShift представила различные корпоративные функции (такие как управление доступом на основе ролей, встроенное хранилище образов и триггеры развертывания модулей), которые нашли свое отражение в основе Kubernetes.

Резюме

- Некоторые основные определяющие факторы, от которых зависит ваш выбор платформы Docker, могут включать в себя позицию «покупка по сравнению со сборкой», позицию в области безопасности, вашу облачную стратегию и то, склонна ли ваша организация решать технические проблемы с помощью «монолитных» или «частичных» продуктов.
- Эти факторы, в свою очередь, могут зависеть от технических драйверов вашего программного обеспечения, требований времени выхода на рынок, уровня независимости потребителей, вашей стратегии открытого исходного кода и вашей организационной структуры.
- В более крупной организации мультиплатформенный подход может иметь смысл, но, возможно, следует позаботиться о том, как обеспечить согласованность подхода на этих платформах, чтобы снизить в дальнейшем организационную неэффективность.
- Основные функциональные области, которые могут учитываться при реализации платформы Docker, включают в себя то, как будут создаваться образы, их сканирование и целостность, управление секретами, реестры образов и базовая ОС.
- Важными игроками в области платформы Docker являются Docker Inc., три крупных облачных провайдера (AWS, Google Cloud Platform и Microsoft Azure) и Cloud Native Computing Foundation (CNCF).
- CNCF – очень влиятельная организация, которая инкубирует и поддерживает ключевые технические компоненты с открытым исходным кодом платформ Docker. Полное признание CNCF является сигналом того, что эта технология будет устойчивой.

Часть 5

Docker в рабочем окружении

Наконец-то мы готовы рассмотреть запуск Docker в рабочем окружении. В этой части мы изучим ключевые операционные аспекты при работе Docker в реальных средах.

Безопасность находится в центре внимания главы 14. Благодаря практическим методам вы получите реальное понимание проблем безопасности, которые ставит Docker, и того, как можно их решить.

Резервное копирование, ведение журнала и управление ресурсами рассматриваются в главе 15, где мы покажем, как можно управлять этими традиционными задачами системного администратора в контексте Docker.

Наконец, в главе 16 мы рассмотрим, что можно сделать, если что-то пойдет не так, охватывая некоторые общие области, в которых Docker может столкнуться с проблемами, а также способы отладки контейнеров в рабочем окружении.

Глава 14

Docker и безопасность

О чем рассказывается в этой главе:

- безопасность из коробки, которую предлагает Docker;
- что сделал Docker, чтобы стать более безопасным;
- что другие делают по поводу этого;
- какие еще шаги можно предпринять для улучшения проблем безопасности;
- как управлять правами пользователя в Docker с помощью aPaaS, потенциально в мультиарендной среде.

Как следует из документации к Docker, доступ к API Docker подразумевает доступ к привилегиям пользователя root, поэтому Docker часто должен запускаться с помощью sudo, или пользователь должен быть добавлен в группу пользователей (которую можно назвать «docker», или «dockerroot»), позволяющую получить доступ к API Docker.

В этой главе мы рассмотрим вопрос безопасности в Docker.

14.1. Получение доступа к Docker, и что это значит

Вам, наверное, интересно, какой ущерб может нанести пользователь, если он сможет запустить Docker. В качестве простого примера приведенная ниже команда (не выполняйте ее!) удалит все двоичные файлы в /sbin на вашем хост-компьютере (если вы убрали фиктивный флаг `--donotrunme`):

```
docker run --donotrunme -v /sbin:/sbin busybox rm -rf /sbin
```

Стоит отметить, что это действительно так, даже если вы не являетесь пользователем root.

Следующая команда покажет вам содержимое файла безопасного теневого пароля из хост-системы:

```
docker run -v /etc/shadow:/etc/shadow busybox cat /etc/shadow
```

Небезопасность Docker часто неправильно толкуют отчасти из-за неправильного понимания преимуществ пространств имен в ядре. Пространства имен Linux обеспечивают изоляцию от других частей системы, но уровень изоляции у вас в Docker на ваше усмотрение (как видно из предыдущих примеров с командой `docker run`). Кроме того, не все части ОС Linux имеют возможность пространства имен. Устройства и модули ядра – примеры основных функций Linux, которые не имеют пространства имен.

> **ПОДСКАЗКА.** Пространства имен Linux были разработаны, чтобы позволить процессам иметь представление о системе, отличное от других процессов. Например, *пространство имен процессов* означает, что контейнеры могут видеть только процессы, связанные с этим контейнером, – другие процессы, выполняющиеся на том же хосте, фактически невидимы для них. *Пространство имен сети* означает, что у контейнеров есть собственный сетевой стек, доступный для них. Пространства имен были частью ядра Linux в течение ряда лет.

Кроме того, поскольку у вас есть возможность взаимодействовать с ядром в качестве пользователя root из контейнера через системные вызовы, любая уязвимость ядра может быть использована пользователем root в контейнере Docker. Конечно, в случае с виртуальными машинами подобная атака возможна через доступ к гипервизору, потому что сообщения об уязвимостях безопасности в гипервизорах также имели место.

Еще один способ понять риски в данном случае – думать о том, что запуск контейнера Docker ничем не отличается (с точки зрения безопасности) от возможности установки любого пакета через менеджера пакетов. Ваши требования к безопасности при запуске контейнеров Docker должны быть такими же, как и для установки пакетов. Если у вас есть Docker, можете установить программное обеспечение от имени пользователя root. Отчасти поэтому некоторые утверждают, что Docker лучше всего понимать как систему упаковки программного обеспечения.

> **ПОДСКАЗКА.** Ведется работа по устранению этой опасности с помощью пространства имен пользователей, когда пользователь root отображается в контейнере в непривилегированного пользователя на хосте.

14.1.1. Вас это волнует?

Учитывая, что доступ к API Docker эквивалентен корневому доступу, возникает следующий вопрос: «Вам есть до этого дело?» Хотя это может показаться странным, но безопасность – это доверие и, если вы доверяете своим пользователям установку программного обеспечения в среде, в которой они работают, для них не должно быть никаких препятствий для запуска там контейнеров Docker. Проблемы с безопасностью возникают в первую очередь

при рассмотрении мультиарендных сред. Поскольку пользователь root внутри вашего контейнера в ключевых отношениях совпадает с пользователем root вне вашего контейнера, наличие большого количества разных пользователей с правами пользователя root в вашей системе вызывает потенциальную обеспокоенность.

> **ПОДСКАЗКА.** Мультиарендная среда – это среда, в которой много разных пользователей совместно используют одни и те же ресурсы. Например, две команды могут использовать один и тот же сервер на двух разных виртуальных машинах. Мультиарендность предлагает экономию средств за счет совместного использования оборудования, а не предоставления оборудования для конкретных приложений. Но это может привести к другим проблемам, связанным с надежностью обслуживания и изоляцией безопасности, и их может компенсировать снижение затрат.

Некоторые организации используют подход, при котором Docker работает на выделенной виртуальной машине для каждого пользователя. Виртуальная машина может применяться для обеспечения безопасности, эксплуатации или изоляции ресурсов. В рамках границы доверия виртуальной машины пользователи запускают контейнеры Docker ради повышения производительности и эксплуатационных преимуществ, которые они дают. Это подход, принятый Google Compute Engine, при котором виртуальная машина помещается между контейнером пользователя и базовой инфраструктурой, чтобы обеспечить дополнительный уровень безопасности и ряд эксплуатационных преимуществ. Google имеет в своем распоряжении более чем немного вычислительных ресурсов, поэтому она не возражает против этого.

14.2. Меры безопасности в Docker

Специалисты Docker уже предприняли различные меры для снижения рисков безопасности при работе с контейнерами. Например:

- некоторые точки монтирования ядра (такие как /proc и /sys) теперь монтируются как точки только для чтения;
- мандаты Linux по умолчанию были ограничены;
- поддержка сторонних систем безопасности, таких как SELinux и AppArmor, уже существует.

В этом разделе мы более подробно рассмотрим эти и другие меры, которые вы можете предпринять, чтобы снизить риски запуска контейнеров в вашей системе.

МЕТОД 93 Ограничение мандатов

Как мы уже упоминали, пользователь root в контейнере – тот же, что и пользователь root на хосте. Но не все они созданы равными. Linux предоставляет

вам возможность назначать более детальные привилегии пользователю root внутри процесса.

Эти детальные привилегии называются *мандатами* и позволяют ограничить ущерб, который может нанести пользователь, даже если это root. Данный метод показывает, как управлять этими мандатами при запуске контейнеров Docker, особенно если вы не полностью доверяете их содержимому.

ПРОБЛЕМА

Вы хотите уменьшить способность контейнеров выполнять вредоносные действия на вашем хост-компьютере.

РЕШЕНИЕ

Сбросьте мандаты, доступные для контейнера, используя флаг `--drop-cap`.

Модель доверия UNIX

Чтобы понять, что означает термин «сбрасывание мандатов», требуется немного предыстории. Когда была разработана система Unix, модель доверия не являлась сложной. У вас были администраторы, которым доверяли (пользователи root) и пользователи, которым не доверяли. Пользователи root могут делать что угодно, тогда как обычные пользователи могут влиять только на свои файлы. Поскольку система обычно использовалась в университетской лаборатории и была небольшой, эта модель имела смысл.

По мере роста модели Unix и появления интернета эта модель приобретала все меньше смысла. Такие программы, как веб-серверы, нуждались в правах доступа пользователя root для обслуживания контента через порт 80, но они также эффективно действовали в качестве прокси для выполнения команд на хосте. Для этого были созданы стандартные шаблоны, такие как привязка к порту 80 и сброс эффективного идентификатора пользователя до пользователя без полномочий root. Пользователи, выполняющие все виды ролей, от системных администраторов до администраторов баз данных и инженеров службы поддержки приложений и разработчиков, возможно, могут нуждаться в детальном доступе к различным ресурсам в системе. Группы Unix в некоторой степени облегчают это, но моделирование этих требований – как скажет вам любой системный администратор – неочевидная проблема.

Мандаты Linux

В попытке поддержать более детальный подход к управлению привилегированными пользователями инженеры-программисты Linux Kernel разработали *мандаты*. Это была попытка разбить монолитную корневую привилегию на куски функциональности, которые можно было предоставить дискретно. Вы можете прочитать о них более подробно, выполнив команду `man 7 capabilities` (при условии, что у вас установлена страница руководства).

Docker по умолчанию отключил некоторые мандаты. Это означает, что даже если у вас есть root в контейнере, некоторые вещи вы не сможете сделать. Например, мандат `CAP_NET_ADMIN`, который позволяет влиять на сетевой стек хоста, по умолчанию отключен.

В табл. 14.1 перечислены мандаты Linux, дано краткое описание того, что они разрешают, и указано, разрешены ли они по умолчанию в контейнерах Docker.

Таблица 14.1 ❖ Мандаты Linux в контейнерах Docker

Мандат	Описание	Включен?
CHOWN	Вносим изменения владельца в любые файлы	Да
DAC_OVERRIDE	Переопределяем проверки чтения, записи и выполнения	Да
FSETID	Не очищаем биты suid и guid при изменении файлов	Да
FOWNER	Переопределяем проверки владения при сохранении файлов	Да
KILL	Обходим проверки прав на основе сигналов	Да
MKNOD	Создаем специальные файлы с помощью mknod	Да
NET_RAW	Используем raw и пакетные сокеты и привязываем к портам для прозрачного проксирования	Да
SETGID	Вносим изменения в групповое владение процессами	Да
SETUID	Вносим изменения в пользовательское владение процессами	Да
SETFCAP	Устанавливаем мандаты файлов	Да
SETPCAP	Если мандаты файлов не поддерживаются, применяем ограничения мандатов по отношению к и из других процессов	Да
NET_BIND_SERVICE	Привязываем сокеты к портам до 1024	Да
SYS_CHROOT	Используем chroot	Да
AUDIT_WRITE	Выполняем запись в журналы ядра	Да
AUDIT_CONTROL	Включаем/отключаем ведение журнала ядра	Нет
BLOCK_SUSPEND	Используем функции, которые блокируют способность системы приостанавливаться	Нет
DAC_READ_SEARCH	Обходим проверку прав доступа к файлам при чтении файлов и каталогов	Нет

Мандат	Описание	Включен?
IPC_LOCK	Блокировка памяти	Нет
IPC_OWNER	Обходим права доступа для объектов межпроцессного взаимодействия	Нет
LEASE	Устанавливаем лизинг (отслеживание попыток открыть или обрезать) на обычных файлах	Нет
LINUX_IMMUTABLE	Устанавливаем флаги индексных дескрипторов FS_APPEND_FL и FS_IMMUTABLE_FL	Нет
MAC_ADMIN	Переопределяем обязательный контроль доступа (относится к модулю безопасности Smack)	Нет
MAC_OVERRIDE	Обязательные изменения контроля доступа (относится к модулю безопасности Smack)	Нет
NET_ADMIN	Различные операции, связанные с сетью, включая изменения брандмауэра или IP и конфигурацию интерфейса	Нет
NET_BROADCAST	Не используется	Нет
SYS_ADMIN	Диапазон административных функций - см. man capabilities для получения дополнительной информации	Нет
SYS_BOOT	Перезагрузка	Нет
SYS_MODULE	Загрузка/выгрузка модулей ядра	Нет
SYS_NICE	Манипулируем приоритетом nice процессов	Нет
SYS_PACCT	Включаем или выключаем учет процессов	Нет
SYS_PTRACE	Отслеживаем системные вызовы процессов и другие мандаты для манипуляций процессами	Нет
SYS_RAWIO	Выполняем ввод/вывод в различных основных частях системы, таких как память и устройства SCSI	Нет
SYS_RESOURCE	Контроль и переопределение лимитов различных ресурсов	Нет
SYS_TIME	Устанавливаем системные часы	Нет
SYS_TTY_CONFIG	Привилегированные операции на виртуальных терминалах	Нет

> **ПРИМЕЧАНИЕ.** Если вы не используете по умолчанию контейнерный движок Docker (libcontainer), эти мандаты в вашей установке могут быть другими. Если у вас есть системный администратор и вы хотите быть уверены, спросите у него.

К сожалению, авторы ядра выделили в системе только 32 мандата, поэтому масштабы мандатов выросли, по мере того как все больше и больше детальных привилегий пользователя root было выделено из ядра. В частности, смутно упомянутый мандат `CAP_SYS_ADMIN` охватывает такие действия, как изменение имени домена хоста до превышения общесистемного ограничения на количество открытых файлов.

Один из крайних подходов – удалить из контейнера все мандаты, которые по умолчанию включены в Docker, и посмотреть, что перестанет работать. Здесь мы запускаем оболочку bash с удаленными по умолчанию мандатами:

```
$ docker run -ti --cap-drop=CHOWN --cap-drop=DAC_OVERRIDE \
--cap-drop=FSETID --cap-drop=FOWNER --cap-drop=KILL --cap-drop=MKNOD \
--cap-drop=NET_RAW --cap-drop=SETGID --cap-drop=SETUID \
--cap-drop=SETFCAP --cap-drop=SETPCAP --cap-drop=NET_BIND_SERVICE \
--cap-drop=SYS_CHROOT --cap-drop=AUDIT_WRITE debian /bin/bash
```

Если вы запустите приложение из этой оболочки, то увидите, где оно не работает должным образом, и заново добавите необходимые мандаты. Например, вам может понадобиться мандат для изменения владельца файла, поэтому придется избавиться от сбрасывания мандата `FOWNER` в предыдущем коде для запуска приложения:

```
$ docker run -ti --cap-drop=CHOWN --cap-drop=DAC_OVERRIDE \
--cap-drop=FSETID --cap-drop=KILL --cap-drop=MKNOD \
--cap-drop=NET_RAW --cap-drop=SETGID --cap-drop=SETUID \
--cap-drop=SETFCAP --cap-drop=SETPCAP --cap-drop=NET_BIND_SERVICE \
--cap-drop=SYS_CHROOT --cap-drop=AUDIT_WRITE debian /bin/bash
```

> **ПОДСКАЗКА.** Если вы хотите включить или отключить все мандаты, то можете использовать `all` вместо определенного мандата, как, например, `docker run -ti --cap-drop=all ubuntu bash`.

ОБСУЖДЕНИЕ

Если вы выполните несколько основных команд в оболочке bash, где все мандаты отключены, то увидите, что она вполне пригодна для использования. Хотя ваш пробег может варьироваться при запуске более сложных приложений.

> **ПРЕДУПРЕЖДЕНИЕ.** Стоит пояснить, что многие из этих мандатов связаны с мандатами пользователя root, чтобы влиять на объекты *других* пользователей в системе, а не на собственные. Пользователь root

по-прежнему может изменять владельца файлов root на хосте с помощью утилиты chown, если они размещены в контейнере и имеют доступ к файлам хоста, например, через монтируемый том. Таким образом, все еще стоит сделать так, чтобы приложения переходили к обычному пользователю как можно скорее с целью защитить систему, даже если все эти мандаты отключены.

Эта возможность тонкой настройки мандатов вашего контейнера означает, что использование флага `--privileged` для команды `docker run` не требуется. Процессы, которым нужны мандаты, будут подлежать аудиту и контролироваться администратором хоста.

МЕТОД 94 «Плохой» образ Docker для сканирования

Одной из проблем, быстро распознаваемых в экосистеме Docker, была проблема уязвимостей – если у вас неизменный образ, вы также не получите никаких исправлений безопасности. Это может не быть проблемой, если вы следуете передовым рекомендациям Docker по минимализму образов, хотя трудно сказать.

Сканеры образов были созданы в качестве решения этой проблемы – как способ выявления неполадок с образом – однако вопрос о том, как их оценивать, все еще открыт.

ПРОБЛЕМА

Вы хотите определить, насколько эффективен сканер образов.

РЕШЕНИЕ

Создайте «заведомо плохой» образ для проверки своих сканеров.

Мы столкнулись с этой проблемой во время работы. Существует множество сканеров образов Docker (таких как Clair), но коммерческие предложения утверждают, что подробно исследуют образ, чтобы определить любые потенциальные проблемы, скрывающиеся в нем.

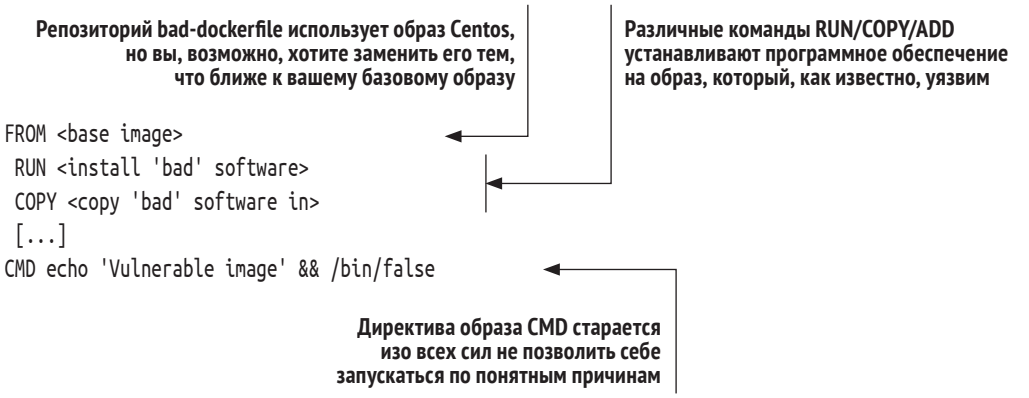
Однако не было образа, содержащего известные и задокументированные уязвимости, которые мы могли бы использовать для проверки эффективности этих сканеров. Это не удивительно, так как большинство образов не рекламируют свою собственную незащищенность!

Поэтому мы придумали заведомо плохой образ. Он доступен для скачивания:

```
$ docker pull imiell/bad-dockerfile
```

Принцип прост: создайте файл Dockerfile, чтобы собрать образ, пронизанный документированными уязвимостями, и направьте его на ваш потенциальный сканер.

Последняя версия Dockerfile доступна по адресу: https://github.com/ianmiell/bad-dockerfile. Он все еще в стадии изменения, поэтому здесь не приводится. Однако форма довольно проста:



Образ содержит спектр уязвимостей, предназначенных для того, чтобы развернуть сканер до предела.

В самом простом случае образ устанавливает программное обеспечение, о котором известно, что оно уязвимо, с помощью менеджера пакетов. В каждой категории образ Docker пытается содержать уязвимости различной степени серьезности.

Более сложное размещение уязвимостей выполняется, например, путем копирования незащищенного кода на JavaScript, используя менеджеры пакетов для конкретного языка (таких как npm для JavaScript, gem для Ruby и pip для Python) для установки уязвимого кода и даже компиляции определенной версии bash (с печально известной уязвимостью Shellshock), и размещение его в неожиданном месте, чтобы избежать множества методов сканирования.

ОБСУЖДЕНИЕ

Вы можете подумать, что лучшее решение для сканирования – это то, которое ловит большинство общеизвестных уязвимостей информационной безопасности. Но это необязательно так. Хорошо, если сканер обнаружит, что образ содержит уязвимость, и это очевидно. Помимо этого, однако, поиск уязвимостей может стать скорее искусством, чем наукой.

> **ПОДСКАЗКА.** Common Vulnerability Exposure (CVE) – это идентификатор конкретной уязвимости, обнаруженной в общедоступном программном обеспечении. Примером CVE может быть CVE-2001-0067, где первое четырехзначное число – это год обнаружения, а второе – счетчик выявленной уязвимости за этот год.

Например, уязвимость может быть очень серьезной (такой как получение прав суперпользователя на вашем хост-сервере), но чрезвычайно трудной для использования (например, требующей ресурсов национального государства).

Вы (или организация, за которую вы несете ответственность), возможно, меньше беспокоитесь об этом, чем об уязвимости, которая менее серьезна, но проста в использовании. Если, например, в вашей системе имеет место

DoS-атака, риска утечки данных или проникновения нет, но вы можете разориться из-за этого, поэтому вас больше беспокоит, как это исправить, нежели какая-то неясная атака на шифр, требующая вычислительной мощности на десятки тысяч долларов.

> **ЧТО ТАКОЕ DoS-АТАКА?** DoS означает «отказ в обслуживании». Это атака, которая приводит к снижению способности вашей системы справляться со спросом. DoS-атака способна привести к перегрузке вашего веб-сервера до такой степени, что он не сможет отвечать законным пользователям.

Стоит также рассмотреть вопрос о том, действительно ли уязвимость доступна в работающем контейнере. В образе может существовать старая версия веб-сервера Apache, но, если он в действительности никогда не запускается контейнером, уязвимость фактически игнорируется. Такое случается часто. Менеджеры пакетов регулярно вводят зависимости, которые на самом деле не нужны только потому, что это упрощает управление ими.

Если безопасность является серьезной проблемой, это может быть еще одной причиной наличия небольших образов (см. главу 7) – даже если какая-то часть программного обеспечения не используется, она все равно может отображаться при сканировании безопасности, что приводит к потере времени, пока ваша организация пытается разобраться, нужно ли это исправлять.

Надеемся, что этот метод дал вам пищу для размышлений, когда вы думали над тем, какой сканер вам подходит. Как всегда, это баланс между стоимостью, тем, что вам нужно, и тем, сколько вы готовы работать, чтобы найти правильное решение.

14.3. Обеспечение доступа к Docker

Лучший способ предотвратить небезопасное использование демона Docker – вообще запретить его использование.

Вероятно, вы впервые столкнулись с ограниченным доступом, когда устанавливали Docker, и вам нужно было использовать команду `sudo` для запуска самого Docker. В методе 41 описано, как избирательно разрешать пользователям на локальном компьютере использовать Docker без этого ограничения.

Но это не поможет, если у вас есть пользователи, подключающиеся к демону Docker с другого компьютера. Мы рассмотрим несколько способов, как обеспечить большую безопасность в таких ситуациях.

МЕТОД 95 HTTP-аутентификация на вашем экземпляре Docker

В методе 1 вы увидели, как открыть сети доступ к вашему демону, а в методе 4 – как выследить Docker API с помощью socat.

Этот метод сочетает в себе два предыдущих: вы сможете получить удаленный доступ к своему демону и просматривать ответы. Доступ ограничен

для тех, у кого есть комбинация имени пользователя и пароля, так что это несколько безопаснее. В качестве бонуса вам не нужно перезапускать демон Docker, чтобы добиться этого – запустите демон контейнера.

ПРОБЛЕМА

Вам нужна базовая аутентификация с доступом к сети на вашем демоне Docker.

РЕШЕНИЕ

Примените HTTP-аутентификацию, чтобы временно использовать своего демона Docker вместе с другими.

На рис. 14.1 показана окончательная архитектура этого метода.

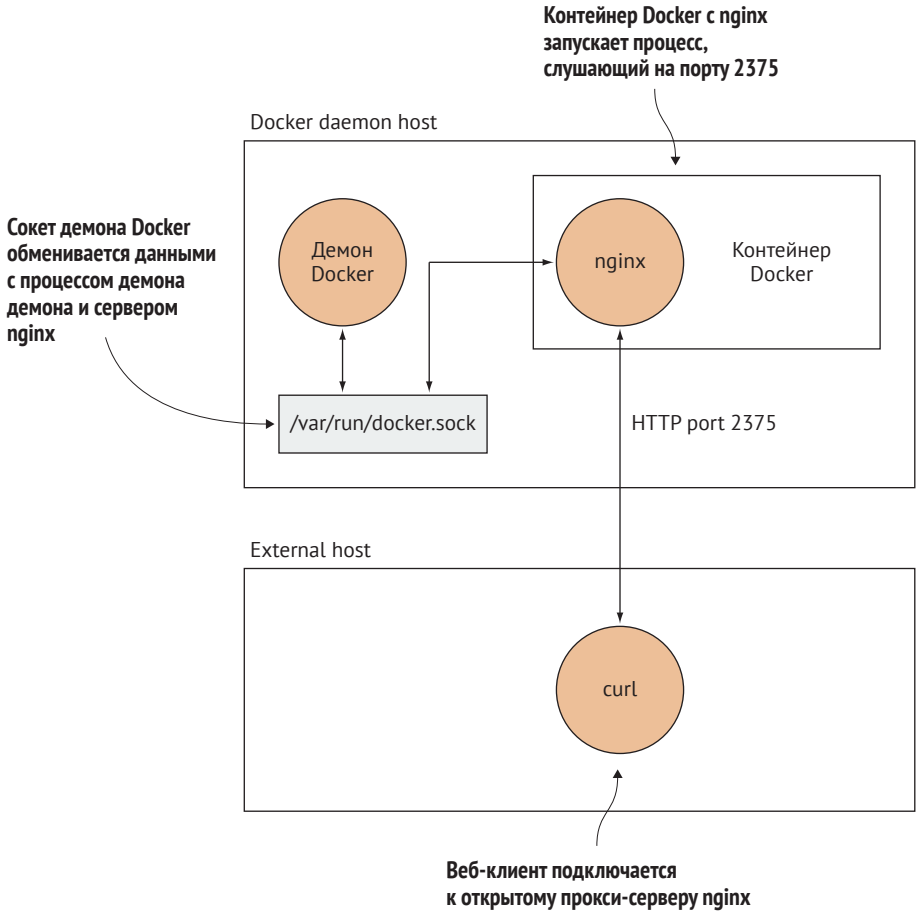
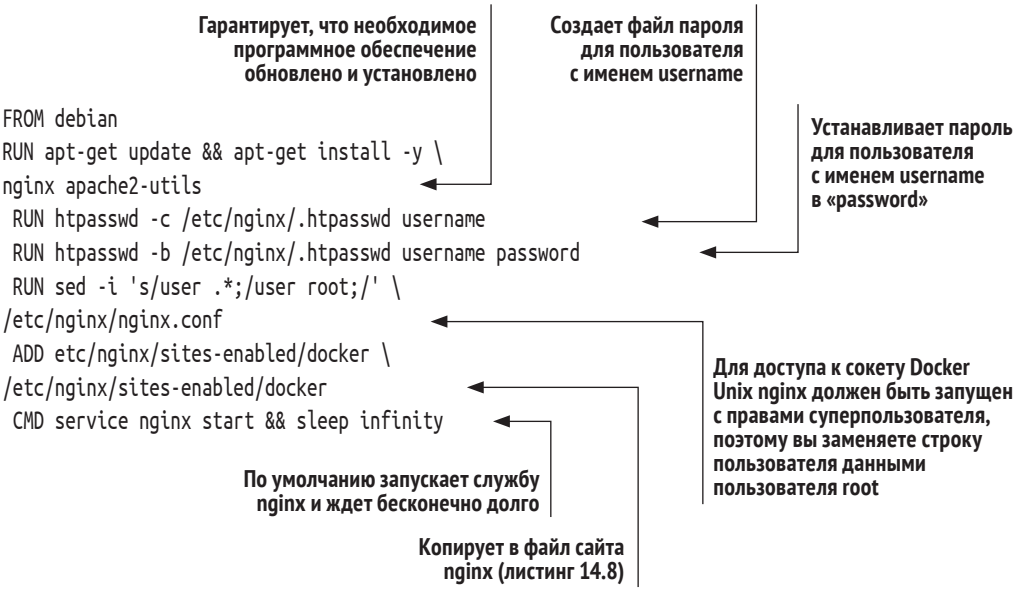


Рис. 14.1 ❖ Архитектура демона Docker с базовой аутентификацией

ПРИМЕЧАНИЕ. Предполагается, что ваш демон Docker использует метод доступа сокета Unix по умолчанию в /var/run/docker.sock.

Код, приведенный в этом методе доступен по адресу https://github.com/docker-in-practice/docker-authenticate. В следующем листинге показано содержимое файла Dockerfile в этом репозитории, использованном, чтобы создать образ для данного метода.

Листинг 14.1. Файл Dockerfile



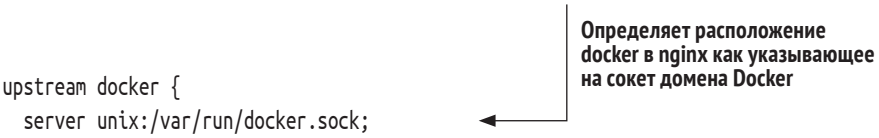

```
FROM debian
RUN apt-get update && apt-get install -y \
nginx apache2-utils
 RUN htpasswd -c /etc/nginx/.htpasswd username
 RUN htpasswd -b /etc/nginx/.htpasswd username password
 RUN sed -i 's/user .*;/user root;/' \
/etc/nginx/nginx.conf
 ADD etc/nginx/sites-enabled/docker \
/etc/nginx/sites-enabled/docker
 CMD service nginx start && sleep infinity
```

Файл .htpasswd, настроенный с помощью команды `htpasswd`, содержит учетные данные, которые необходимо проверить, перед тем как разрешить (или отклонить) доступ к сокету Docker. Создавая этот образ самостоятельно, вы, вероятно, захотите изменить имя пользователя и пароль на этих двух шагах, чтобы настроить учетные данные с доступом к сокету Docker.

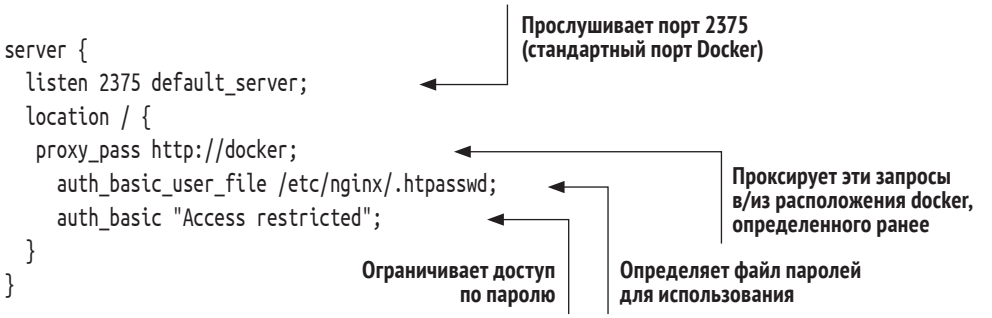
> **ВНИМАНИЕ!** Будьте осторожны, чтобы не делиться этим образом, так как он будет содержать установленный вами пароль!

Файл сайта nginx для Docker показан в следующем листинге.

Листинг 14.2. /etc/nginx/sites-enabled/docker




```
upstream docker {
  server unix:/var/run/docker.sock;
}
```

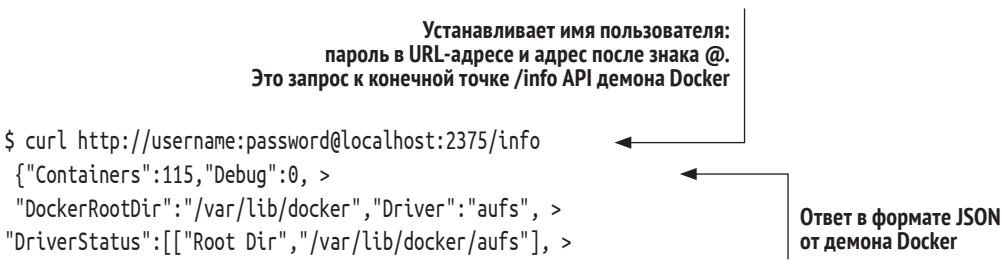


Теперь запустите образ в качестве контейнера демона, отображая необходимые ресурсы из хост-компьютера:

```
$ docker run -d --name docker-authenticate -p 2375:2375 \
  -v /var/run:/var/run dockerinpractice/docker-authenticate
```

Контейнер будет запущен в фоновом режиме с именем `docker-authenticate`, чтобы вы могли обратиться к нему позже. Порт контейнера 2375 открыт на хосте, и контейнеру предоставляется доступ к демону Docker путем монтирования каталога по умолчанию, содержащего сокет Docker в качестве тома. Если вы берете пользовательский образ со своим собственным именем пользователя и паролем, нужно заменить имя образа здесь своим собственным.

Веб-сервис теперь готов к работе. Если вы выполните обращение к сервису с помощью утилиты `curl`, используя заданные вами имя пользователя и пароль, то увидите ответ API:




```
$ curl http://username:password@localhost:2375/info
 {"Containers":115,"Debug":0, >
 "DockerRootDir":"/var/lib/docker","Driver":"aufs", >
"DriverStatus":[["Root Dir","/var/lib/docker/aufs"], >
["Backing Filesystem","extfs"],["Dirs","1033"]], >
"ExecutionDriver":"native-0.2", >
"ID":"QSCJ:NLPA:CRS7:WCOI:K23J:6Y2V:G35M:BF55:OA2W:MV3E:RG47:DG23", >
"IPv4Forwarding":1,"Images":792, >
"IndexServerAddress":"https://index.docker.io/v1/", >
"InitPath":"/usr/bin/docker","InitSha1":"", >
"KernelVersion":"3.13.0-45-generic", >
"Labels":null,"MemTotal":5939630080,"MemoryLimit":1, >
"NCPU":4,"NEventsListener":0,"NFd":31,"NGoroutines":30, >
"Name":"rothko","OperatingSystem":"Ubuntu 14.04.2 LTS", >
"RegistryConfig":{"IndexConfigs":{"docker.io": >
{"Mirrors":null,"Name":"docker.io", >
```

```
"Official":true,"Secure":true}}, >
"InsecureRegistryCIDRs":["127.0.0.0/8"]},"SwapLimit":0}
```

Когда вы закончите, удалите контейнер с помощью этой команды:

```
$ docker rm -f docker-authenticate
```

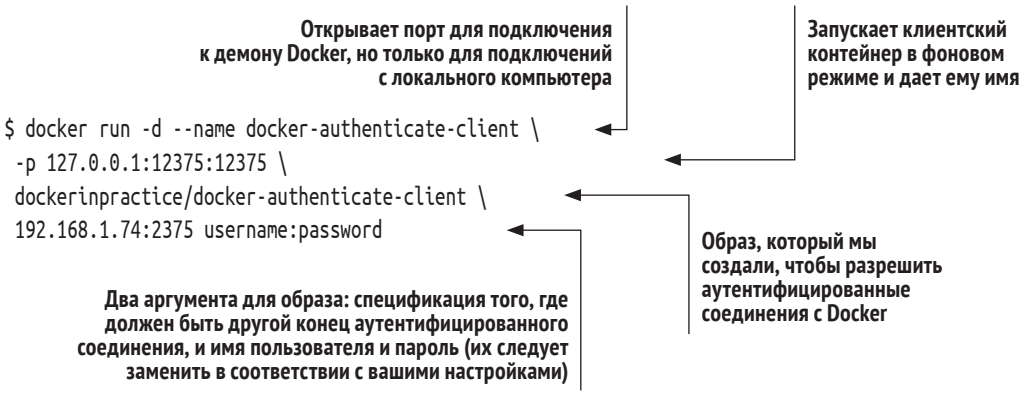
Доступ теперь отменен.

Используем команду docker?

Читатели могут задаться вопросом, смогут ли другие пользователи подключиться с помощью команды docker-, например, используя что-то вроде этого:

```
docker -H tcp://username:password@localhost:2375 ps
```

На момент написания этой главы функция аутентификации не была встроена в сам Docker. Но мы создали образ, который будет обрабатывать аутентификацию и позволять Docker подключаться к демону. Просто используйте образ:



Обратите внимание, что localhost или 127.0.0.1 не будут работать, если вам нужно указать другой конец аутентифицируемого соединения, – если вы хотите опробовать его на одном хосте, нужно использовать ip addr для определения внешнего IP-адреса своего компьютера.

Теперь можете применить аутентифицированное соединение с помощью следующей команды:

```
docker -H localhost:12375 ps
```

Имейте в виду, что интерактивные команды Docker (run и exec с аргументом -i) не будут работать с этим соединением из-за ограничений реализации.

ОБСУЖДЕНИЕ

В этом методе мы показали, как настроить базовую аутентификацию для вашего сервера Docker в доверенной сети. В следующем методе рассмотрим шифрование трафика, чтобы злоумышленники не могли увидеть, что вы делаете, или даже внедрить вредоносные данные или код.

ПРЕДУПРЕЖДЕНИЕ. Этот метод обеспечивает базовый уровень *аутентификации*, но не серьезный уровень *безопасности* (в частности, некто, способный прослушивать трафик сети, может перехватить ваше имя пользователя и пароль). Настройка сервера, защищенного с помощью TLS, довольно сложна и рассматривается в следующем методе.

МЕТОД 96 Защита API Docker

Здесь мы покажем, как можно открыть свой сервер Docker для посторонних через TCP-порт, в то же время гарантируя, что подключаться могут только доверенные клиенты. Это достигается путем создания секретного ключа, который будет передан только доверенным узлам. Пока этот доверенный ключ остается секретом между сервером и клиентским компьютером, сервер Docker должен быть в безопасности.

ПРОБЛЕМА

Вы хотите, чтобы ваш API Docker надежно обслуживался через порт.

РЕШЕНИЕ

Создайте самоподписанный сертификат и запустите демон Docker, используя флаг `--tls-verify`.

Данный метод безопасности зависит от так называемых *файлов ключей*, создаваемых на сервере.

Эти файлы создаются с использованием специальных средств, которые гарантируют, что их трудно скопировать, если у вас нет *серверного ключа*. На рис. 14.2 приводится обзор того, как это работает.

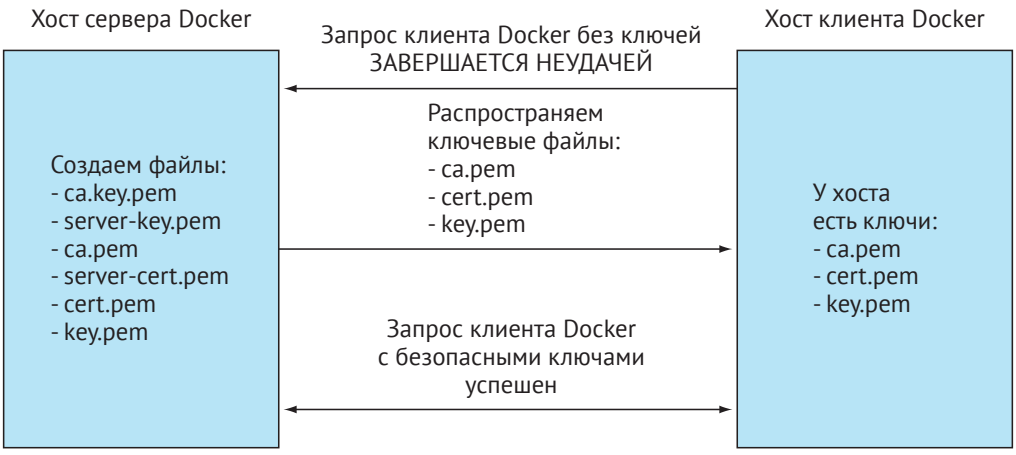


Рис. 14.2 ❖ Настройка и распределение ключей

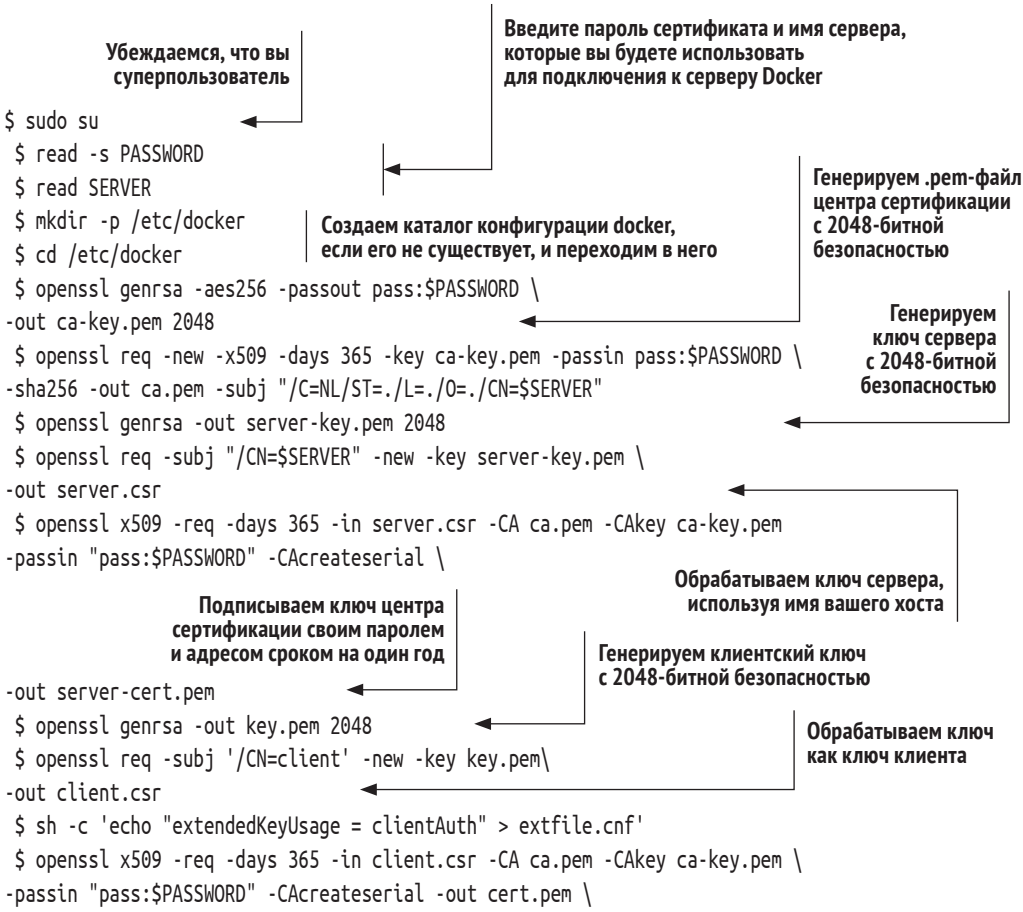
ПОДСКАЗКА. Серверный ключ – это файл, который содержит секретный номер, известный только серверу, и который требуется для чтения сообщений, зашифрованных с помощью файлов секретного ключа, выданных владельцем сервера (так называемые *клиентские ключи*). После создания и распределения ключей их можно использовать для обеспечения безопасности соединения между клиентом и сервером.

Установка сертификата сервера Docker

Сначала вы создаете сертификаты и ключи.

Для генерации ключей требуется пакет OpenSSL, и вы можете проверить, установлен ли он, выполнив в терминале команду `openssl`. Если нет, необходимо установить его, прежде чем приступить к созданию сертификатов и ключей, с помощью следующего кода.

Листинг 14.3. Создание сертификатов и ключей с помощью OpenSSL



```
-extfile extfile.cnf
 $ chmod 0400 ca-key.pem key.pem server-key.pem
 $ chmod 0444 ca.pem server-cert.pem cert.pem
 $ rm client.csr server.csr
```

Подписываем ключ своим паролем сроком на один год

Удаляем оставшиеся файлы

Изменяем права доступа для файлов клиента на read-only для всех

Изменяем для файлов сервера права доступа на read-only для пользователей root

СОВЕТ. В вашей системе может быть установлен сценарий с именем CA.pl, который упрощает этот процесс. Здесь мы представили необработанные команды `openssl`, потому что они более поучительны.

Настройка сервера Docker

Теперь вам нужно установить параметры Docker в файле конфигурации демона Docker, чтобы указать, какие ключи используются для шифрования сообщений (см. приложение B для получения рекомендаций по поводу того, как настроить и перезапустить демон Docker).

Листинг 14.4. Опции Docker для использования новых ключей и сертификатов

Сообщает демону Docker, что вы хотите использовать протокол TLS для защиты подключений к нему

Указывает файл центра сертификации для сервера Docker

```
DOCKER_OPTS="$DOCKER_OPTS --tlsverify"
 DOCKER_OPTS="$DOCKER_OPTS \
--tlscacert=/etc/docker/ca.pem"
 DOCKER_OPTS="$DOCKER_OPTS \
--tlscert=/etc/docker/server-cert.pem"
 DOCKER_OPTS="$DOCKER_OPTS \
--tlskey=/etc/docker/server-key.pem"
 DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2376"
 DOCKER_OPTS="$DOCKER_OPTS \
-H unix:///var/run/docker.sock"
```

Указывает сертификат для сервера

Определяет закрытый ключ, используемый сервером

Открывает демон Docker локально через сокет Unix обычным способом

Открывает демон Docker для внешних клиентов через TCP на порту 2376

Далее необходимо отправить ключи клиентскому хосту, чтобы он мог подключиться к серверу и обмениваться информацией. Вы не хотите показывать свои секретные ключи кому-либо еще, поэтому их необходимо безопасно передать клиенту. Относительно надежный способ сделать это – безопасно скопировать их напрямую с сервера на клиент с помощью утилиты scp. Утилита scp использует, по сути, тот же метод защиты передачи данных, демонстрируемый здесь, только с другими ключами, которые уже были настроены.

На клиентском хосте создайте папку конфигурации Docker в /etc, как делали ранее:

```
user@client:~$ sudo su
root@client:~$ mkdir -p /etc/docker
```

Затем скопируйте файлы с сервера на клиент. Убедитесь, что вы заменили слово «клиент» в следующих командах на имя хоста вашего клиентского компьютера. Также удостоверьтесь, что все файлы доступны для чтения пользователю, который будет выполнять команду `docker` на клиенте.

```
user@server:~$ sudo su
root@server:~$ scp /etc/docker/ca.pem client:/etc/docker
root@server:~$ scp /etc/docker/cert.pem client:/etc/docker
root@server:~$ scp /etc/docker/key.pem client:/etc/docker
```

Тестирование

Чтобы проверить настройки, сначала попробуйте сделать запрос к серверу Docker без каких-либо учетных данных.

Запрос должен быть отклонен:

```
root@client~: docker -H myserver.localdomain:2376 info
FATA[0000] Get http://myserver.localdomain:2376/v1.17/info: malformed HTTP >
response "\x15\x03\x01\x00\x02\x02". Are you trying to connect to a >
TLS-enabled daemon without TLS?
```

Затем соединитесь с учетными данными, которые должны вернуть полезный вывод:

```
root@client~: docker --tlsverify --tlscacert=/etc/docker/ca.pem \
--tlscert=/etc/docker/cert.pem --tlskey=/etc/docker/key.pem \
-H myserver.localdomain:2376 info
243 info
Containers: 3
Images: 86
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 92
Execution Driver: native-0.2
Kernel Version: 3.16.0-34-generic
Operating System: Ubuntu 14.04.2 LTS
CPUs: 4
Total Memory: 11.44 GiB
Name: rothko
ID: 4YQA:KK65:FXON:YVLT:BVVH:Y3KC:UATJ:I4GK:S3E2:UTA6:R43U:DX5T
WARNING: No swap limit support
```

ОБСУЖДЕНИЕ

Этот метод дает вам лучшее из обоих миров – демона Docker, открытого для использования другими, и демона, который доступен только для доверенных пользователей. Убедитесь, что вы храните эти ключи в безопасности!

Управление ключами – критический аспект процессов управления ИТ в более крупных организациях. Определенно это стоит денег, поэтому, когда дело доходит до реализации платформы Docker, эта тема может быть в центре внимания. Безопасное развертывание ключей в контейнерах – это задача, которая, возможно, должна быть рассмотрена при проектировании большинства платформ Docker.

14.4. Безопасность за пределами Docker

Безопасность на вашем хосте не ограничивается командой `docker`. В этом разделе вы увидите другие подходы к защите контейнеров Docker, на этот раз за его пределами.

Мы начнем с пары методов, которые изменяют образы, чтобы уменьшить площадь поверхности для внешней атаки, когда они запускаются. В двух последующих методах рассматривается, как запускать контейнеры ограниченно.

Из этих двух последних методов первый демонстрирует подход «платформа приложений как служба» (aPaaS), который гарантирует, что Docker работает в «смирительной рубашке», настроенной и контролируемой администратором. Например, мы запустим сервер OpenShift Origin (aPaaS, при котором контейнеры Docker развертываются управляемым способом), используя команды Docker. Вы увидите, что полномочия конечного пользователя могут быть ограничены и управляться администратором, а доступ к среде выполнения Docker может быть удален.

Второй подход выходит за рамки этого уровня безопасности, чтобы еще больше ограничить свободы, доступные в работающих контейнерах, используя SELinux, технологию безопасности, которая дает вам детальный контроль над тем, кто и что может делать.

> **ПОДСКАЗКА.** SELinux – это инструментальное средство с открытым исходным кодом, созданное Агентством национальной безопасности (АНБ) США, которое удовлетворяет его потребности в строгом контроле доступа. Оно было стандартом безопасности в течение некоторого времени, и оно очень мощное. К сожалению, многие люди просто выключают его, когда сталкиваются с проблемами, вместо того, чтобы потратить время на его изучение. Мы надеемся, что приведенный здесь метод поможет сделать такой подход менее заманчивым.

МЕТОД 97 Сокращение поверхности атаки контейнера с помощью DockerSlim

В разделе 7.3 мы обсудили несколько различных способов создания небольшого образа в ответ на разумную обеспокоенность по поводу объема данных, перемещаемых по сети. Но есть еще одна причина для этого – чем меньше содержит ваш образ, тем меньше злоумышленнику придется взламывать. Конкретный пример: нельзя получить оболочку в контейнере, если оболочка не установлена.

Создание профиля «ожидаемого поведения» для вашего контейнера и последующее его применение во время выполнения означает, что неожиданные действия имеют реальный шанс быть обнаруженными и предотвращенными.

ПРОБЛЕМА

Вы хотите уменьшить образ до самого необходимого, чтобы уменьшить его поверхность атаки.

РЕШЕНИЕ

Используйте DockerSlim, чтобы проанализировать свой образ и изменить его для уменьшения поверхности атаки.

Это средство предназначено для получения образа Docker и уменьшения его до минимума.

Оно доступно по адресу https://github.com/docker-slim/docker-slim.

DockerSlim уменьшает ваш образ Docker по крайней мере двумя различными способами. Во-первых, он сокращает его только до необходимых файлов и помещает эти файлы в один слой.

Конечным результатом является образ, который значительно меньше исходного, толстого аналога.

Во-вторых, он предоставляет профиль seccomp. Это достигается за счет динамического анализа вашего работающего образа. Говоря простым языком, это означает, что он запускает образ и отслеживает, какие файлы и системные вызовы используются. Пока DockerSlim анализирует ваш работающий контейнер, вы должны использовать приложение, как это делали бы все типичные пользователи, чтобы обеспечить сбор необходимых файлов и системных вызовов.

> **ВНИМАНИЕ!** Уменьшая образ с помощью такого инструмента динамического анализа, будьте абсолютно уверены, что вы достаточно проявили его на этапе анализа. В этом пошаговом руководстве используется тривиальный образ, но у вас может быть более сложный, детально профилировать который тяжелее.

В этом методе для демонстрации будет использоваться простое веб-приложение.

Вы:

- настроите DockerSlim;
- создадите образ;
- запустите его в качестве контейнера с помощью DockerSlim;
- обратитесь к конечной точке приложения;
- запустите уменьшенный образ, используя созданный профиль seccomp.

> **ПРИМЕЧАНИЕ.** Профиль seccomp – это, по сути, белый список, из которого системные вызовы могут быть сделаны из контейнера. При запуске контейнера вы можете указать профиль seccomp с уменьшенными или повышенными правами доступа в зависимости от потребностей вашего приложения. Профиль seccomp по умолчанию отключает около 45 системных вызовов из более чем 300. Большинству приложений требуется гораздо меньше.

Установка DockerSlim

Выполните эти команды, чтобы скачать и настроить двоичный файл docker-slim.

Листинг 14.5. Скачивание файла docker-slim и установка его в каталог

```
$ mkdir -p docker-slim/bin && cd docker-slim/bin
 $ wget https://github.com/docker-slim/docker-slim/releases/download/1.18
➥ /dist_linux.zip
 $ unzip dist_linux.zip
 $ cd ..
```

Создает папку docker-slim и подпапку bin

Получает ZIP-файл docker-slim из папки с релизами

Распаковывает полученный zip-файл

Перемещается в родительский каталог, docker-slim

Этот метод был протестирован с использованием предыдущей версией docker-slim. Вы можете посетить GitHub по адресу https://github.com/docker-slim/docker-slim/releases, чтобы увидеть, были ли какие-либо обновления. Этот проект не относится к числу стремительно развивающихся, поэтому обновления не должны быть особо важными.

Теперь у вас есть подпрограмма Docker-Slim в подпапке bin.

Создание толстого образа

Далее вы создадите пример приложения, использующего NodeJS. Это тривиальное приложение, которое просто обслуживает строку в формате JSON на порту 8000. Следующая команда клонирует репозиторий docker-slim,

переходит к коду приложения и встраивает свой файл Dockerfile в образ с именем sample-node-app.

Листинг 14.6. Создание примера приложения docker-slim

```
$ git clone https://github.com/docker-slim/docker-slim.git
$ cd docker-slim && git checkout 1.18
$ cd sample/apps/node
$ docker build -t sample-node-app .
$ cd -
```

Клонирует репозиторий docker-slim, который содержит образец приложения

Проверяет известную рабочую версию репозитория docker-slim

Перемещается в папку приложения NodeJS

Возвращается в предыдущий каталог, где находится двоичный файл docker-slim

Создает образ, дав ему имя sample-node-app

Запуск образа

Теперь, когда вы создали свой образ, следующий шаг включает в себя его запуск в качестве контейнера с оболочкой docker-slim. После инициализации приложения вы обращаетесь к конечной точке приложения для тренировки его кода. Наконец, выводите фоновое приложение docker-slim на передний план и дожидаетесь его завершения.

```
$ ./docker-slim build --http-probe sample-node-app &
$ sleep 10 && curl localhost:32770
{"status":"success","info":"yes!!!","service":"node"}
$ fg
./docker-slim build --http-probe sample-node-app
INFO[0014] docker-slim: HTTP probe started...
INFO[0014] docker-slim: http probe - GET http://127.0.0.1:32770/ => 200
INFO[0014] docker-slim: HTTP probe done.
INFO[0015] docker-slim: shutting down 'fat' container...
INFO[0015] docker-slim: processing instrumented 'fat' container info...
INFO[0015] docker-slim: generating AppArmor profile...
INFO[0015] docker-slim: building 'slim' image...
```

Запускает двоичный файл docker-slim для образа sample-node-app. Возобновляет работу процесса в фоновом режиме. http-probe будет вызывать приложение на всех открытых портах

«Спит» в течение 10 секунд, чтобы позволить приложению sample-node-app запуститься, а затем обращается к порту, на котором работает приложение

Отправляет ответ приложения в формате JSON на терминал

Первый раздел вывода docker-slim показывает его рабочие логи

Возобновляет работу процесса docker-slim и ждет, пока он не завершится

```
Step 1 : FROM scratch
--->
Step 2 : COPY files /
---> 0953a87c8e4f
Removing intermediate container 51e4e625017e
Step 3 : WORKDIR /opt/my/service
---> Running in a2851dce6df7
---> 2d82f368c130
Removing intermediate container a2851dce6df7
Step 4 : ENV PATH "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
➥ /bin"
---> Running in ae1d211f118e
---> 4ef6d57d3230
Removing intermediate container ae1d211f118e
Step 5 : EXPOSE 8000/tcp
---> Running in 36e2ced2a1b6
---> 2616067ec78d
Removing intermediate container 36e2ced2a1b6
Step 6 : ENTRYPOINT node /opt/my/service/server.js
---> Running in 16a35fd2fb1c
---> 7451554aa807
Removing intermediate container 16a35fd2fb1c
Successfully built 7451554aa807
INFO[0016] docker-slim: created new image: sample-node-app.slim
$
$
```

Docker-slim создает «тонкий» контейнер

Когда он завершит работу, вам может понадобиться нажать Return, чтобы получить приглашение

В этом случае «тренировка кода» просто включает в себя обращение к одному URL-адресу и получение ответа. Более сложные приложения будут нуждаться в более разнообразных способах тыкать и понукать, чтобы удостовериться, что они были полностью задействованы.

Обратите внимание, что, согласно документам, нам не нужно самим подключаться к приложению через порт 32770, потому что мы использовали аргумент http-probe. Если вы задействуете его, по умолчанию на каждом открытом порту для корневого URL-адреса («/») будут запускаться HTTP- и HTTPS-запросы с методом GET. Мы вручную используем утилиту `curl` просто для демонстрационных целей.

На данный момент вы создали версию вашего образа sample-node-app.slim. Если вы посмотрите на вывод команды `docker images`, то увидите, что его размер значительно уменьшился.

Размер образа sample-node-app.slim составляет чуть более 14 Мб

```
$ docker images
REPOSITORY            TAG       IMAGE ID        CREATED              SIZE
sample-node-app.slim  latest    7451554aa807    About an hour ago    14.02 MB
sample-node-app       latest    78776db92c2a    About an hour ago    418.5 MB
```

Размер исходного образа sample-node-app составлял более 400 Мб

Сравнив вывод команды `docker history` из приложения с «толстым» образцом с его «тонким» аналогом, вы увидите, что они совершенно разные по структуре.

Команда docker history выполняется в образе sample-node-app

История этого образа показывает каждую команду в своем первозданном виде

```
$ docker history sample-node-app
IMAGE          CREATED        CREATED BY                                      SIZE
78776db92c2a   42 hours ago   /bin/sh -c #(nop) ENTRYPOINT ["node"            0 B
0f044b6540cd   42 hours ago   /bin/sh -c #(nop) EXPOSE 8000/tcp               0 B
555cf79f13e8   42 hours ago   /bin/sh -c npm install                          14.71 MB
6c62e6b40d47   42 hours ago   /bin/sh -c #(nop) WORKDIR /opt/my/ser           0 B
7871fb6df03b   42 hours ago   /bin/sh -c #(nop) COPY dir:298f558c6f2          656 B
618020744734   42 hours ago   /bin/sh -c apt-get update && apt-get            215.8 MB
dea1945146b9   7 weeks ago    /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>      7 weeks ago    /bin/sh -c mkdir -p /run/systemd && ec          7 B
<missing>      7 weeks ago    /bin/sh -c sed -i 's/^#\s*\(deb.*unive          2.753 kB
<missing>      7 weeks ago    /bin/sh -c rm -rf /var/lib/apt/lists/*          0 B
<missing>      7 weeks ago    /bin/sh -c set -xe && echo '#!/bin/s            194.6 kB
<missing>      7 weeks ago    /bin/sh -c #(nop) ADD file:8f997234193          187.8 MB
$ docker history sample-node-app.slim
IMAGE          CREATED        CREATED BY                                      SIZE
7451554aa807   42 hours ago   /bin/sh -c #(nop) ENTRYPOINT ["node"            0 B
2616067ec78d   42 hours ago   /bin/sh -c #(nop) EXPOSE 8000/tcp               0 B
4ef6d57d3230   42 hours ago   /bin/sh -c #(nop) ENV PATH=/usr/local           0 B
2d82f368c130   42 hours ago   /bin/sh -c #(nop) WORKDIR /opt/my/ser           0 B
0953a87c8e4f   42 hours ago   /bin/sh -c #(nop) COPY dir:36323da1e97          14.02 MB
```

История тонкого контейнера состоит из меньшего количества команд, включая команду COPY, отсутствующую в оригинальном толстом образе

Команда docker history выполняется в образе sample-node-app.slim

Предыдущий вывод дает представление о том, что делает DockerSlim. Ему удается уменьшить размер образа до (эффективно) одного слоя в 14 Мб, приняв конечное состояние файловой системы и скопировав этот каталог в качестве последнего слоя образа.

Другой артефакт, созданный DockerSlim, относится ко второй цели, как было описано в начале этого метода. Создается файл seccomp.json (в данном случае sample-node-app-seccomp.json), который можно использовать для ограничения действий работающего контейнера. Давайте посмотрим на содержимое этого файла (здесь он отредактирован, так как довольно длинный).

Листинг 14.7. Профиль seccomp

Определяет местоположение файла seccomp в переменной SECCOMPFILE

```
$ SECCOMPFILE=$(ls $(pwd)/.images/*/artifacts/sample-node-app-seccomp.json)
 $ cat ${SECCOMPFILE}
 {
"defaultAction": "SCMP_ACT_ERRNO",
   "architectures": [
   "SCMP_ARCH_X86_64"
   ],
   "syscalls": [
     {
       "name": "capset",
       "action": "SCMP_ACT_ALLOW"
     },
     {
       "name": "rt_sigaction",
       "action": "SCMP_ACT_ALLOW"
     },
     {
       "name": "write",
       "action": "SCMP_ACT_ALLOW"
     },
[...]
     {
       "name": "execve",
       "action": "SCMP_ACT_ALLOW"
     },
     {
       "name": "getcwd",
       "action": "SCMP_ACT_ALLOW"
     }
   ]
}
```

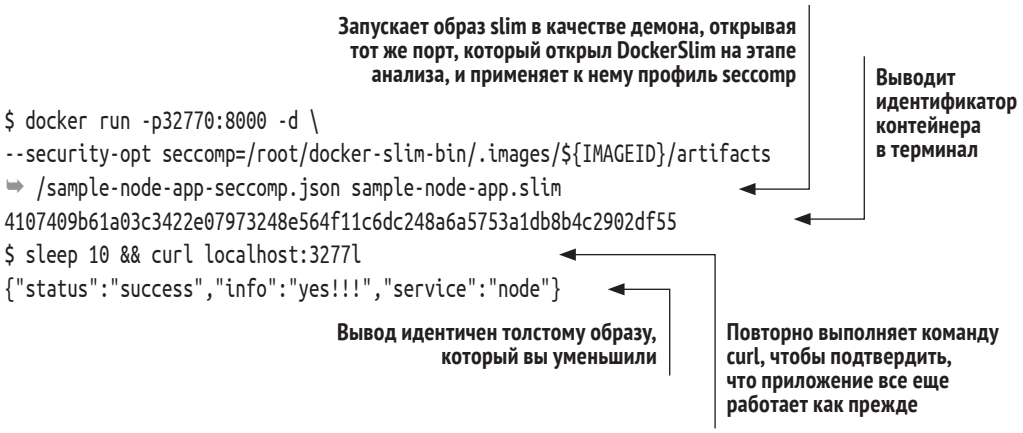
Выводит содержимое этого файла для просмотра

Определяет аппаратную архитектуру, к которой должен применяться этот профиль

Указывает код завершения для процесса, который пытается вызвать любой запрещенный системный вызов

Управляемые системные вызовы здесь заносятся в белый список, определяя для них действие SCMP_ACT_ALLOW

Наконец, вы снова запускаете тонкий образ с профилем seccomp и проверяете, что он работает, как и ожидалось:



ОБСУЖДЕНИЕ

Этот простой пример показал, как можно уменьшить не только размер образа, но и объем действий, которые он может выполнять. Это достигается путем удаления несущественных файлов (также обсуждаемых в методе 59) и сокращения доступных системных вызовов только до тех, которые необходимы для запуска приложения.

Средства «тренировки» приложения здесь были простыми (один запрос `curl` к конечной точке по умолчанию). Для реального приложения существует ряд подходов, которые вы можете использовать, чтобы убедиться, что охватили все возможности. Один из способов – разработка набора тестов с известными конечными точками, а другой – использование средства фаззинга для выброса в приложение большого количества входных данных в автоматическом режиме (это один из способов найти ошибки и уязвимости в вашем программном обеспечении). Самый простой способ – оставить ваше приложение работающим на более длительный период времени, ожидая обращений ко всем необходимым файлам и системным вызовам.

Многие корпоративные средства защиты Docker работают по этому принципу, но более автоматизированным способом. Обычно они позволяют приложению работать некоторое время и отслеживают, какие системные вызовы сделаны, к каким файлам обращаются, а также (возможно), какие мандаты операционной системы используются. Основываясь на этом – и настраиваемом периоде обучения – они могут определить ожидаемое поведение приложения и сообщить о любом поведении, которое, как кажется, выходит за рамки. Например, если злоумышленник получает доступ к работающему контейнеру и запускает двоичный файл bash или открывает неожиданные порты, система может поднять тревогу. DockerSlim позволяет вам заранее контролировать этот процесс, уменьшая возможности злоумышленника, даже если он получил доступ.

Еще один способ уменьшить поверхность атаки вашего приложения – ограничить его мандаты. Это описывается в методе 93.

МЕТОД 98 Удаление секретов, добавленных во время сборки

Когда вы создаете образы в корпоративной среде, часто необходимо использовать ключи и учетные данные для извлечения данных. Если вы используете файл Dockerfile для создания приложения, эти секреты обычно будут присутствовать в истории, даже если вы удалите ее после использования.

Это может быть проблемой безопасности: если кто-то завладеет образом, он также сможет получить секрет в предыдущих слоях.

ПРОБЛЕМА

Вы хотите удалить файл из истории образа.

РЕШЕНИЕ

Используйте команду `docker-squash` для удаления слоев из образа.

Есть простые способы решения этой проблемы, которые работают в теории. Например, вы можете удалить секрет во время его использования следующим образом.

Листинг 14.8. Грубый способ не оставлять секрет в слое

```
FROM ubuntu
RUN echo mysecret > secretfile && command_using_secret && rm secretfile
```

У данного подхода имеется ряд недостатков. Он требует, чтобы секрет был вставлен в код в Dockerfile, поэтому в системе контроля версий он может быть в виде простого текста.

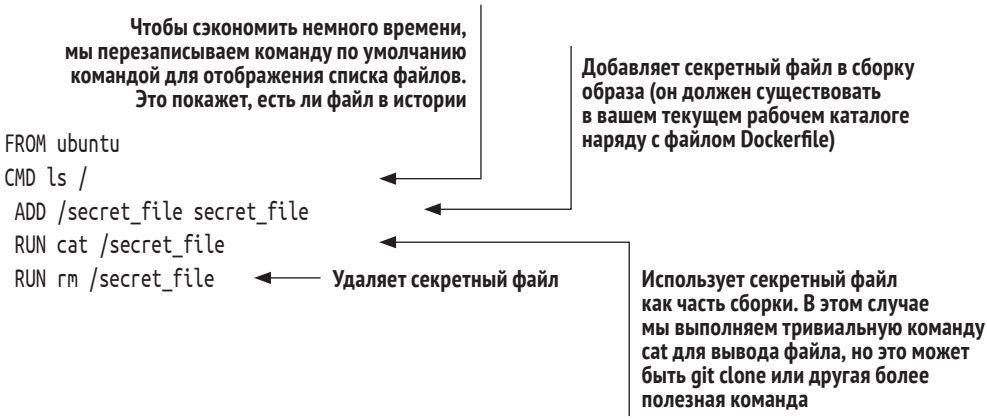
Чтобы избежать этой проблемы, вы можете добавить файл в свой файл .gitignore (или аналогичный) в системе контроля версий и затем – к образу во время сборки. Файл будет добавлен в отдельный слой, который непросто удалить из полученного образа.

Наконец, вы можете использовать переменные среды для хранения секретов, но это также создает риски для безопасности, поскольку эти переменные легко устанавливаются в незащищенных постоянных хранилищах, таких как задания Jenkins. В любом случае вам может быть предложен образ с просьбой стереть из него секрет. Сначала мы продемонстрируем эту проблему с помощью простого примера, а затем покажем вам способ, как удалить секрет из базового слоя.

Образ с секретом

Приведенный ниже файл Dockerfile создаст образ, используя файл secret_file в качестве заполнителя для секретных данных, которые вы поместили в свой образ.

Листинг 14.9. Простой файл Dockerfile с секретом



Теперь можете собрать этот образ, вызвав полученный образ secret_build.

Листинг 14.10. Создание простого образа Docker с секретом

```
$ echo mysecret > secret_file
$ docker build -t secret_build .
Sending build context to Docker daemon 5.12 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
 ---> 08881219da4a
Step 1 : CMD ls /
 ---> Running in 7864e2311699
 ---> 5b39a3cba0b0
Removing intermediate container 7864e2311699
Step 2 : ADD /secret_file secret_file
 ---> a00886ff1240
Removing intermediate container 4f279a2af398
Step 3 : RUN cat /secret_file
 ---> Running in 601fdf2659dd
My secret
 ---> 2a4238c53408
Removing intermediate container 601fdf2659dd
Step 4 : RUN rm /secret_file
 ---> Running in 240a4e57153b
 ---> b8a62a826ddf
Removing intermediate container
```

Как только образ собран, можете продемонстрировать, что у него есть секретный файл, используя метод 27.

Листинг 14.11. Тегирование каждого шага и демонстрация слоя с секретом

```
$ x=0; for id in $(docker history -q secret_build:latest);
➥ do ((x++)); docker tag $id secret_build:step_$x; done
 $ docker run secret_build:step_3 cat /secret_file'
 mysecret
```

Помечает каждый шаг сборки в числовом порядке

Демонстрирует, что секретный файл находится в этом теге образа

Сжатие образов для удаления секретов

Вы видели, что секреты могут оставаться в истории, даже если их нет в конечном образе. Вот тут вступает в действие команда `docker-squash` – она удаляет промежуточные слои, но сохраняет команды Dockerfile (такие как `CMD`, `PORT`, `ENV` и т. д.) и исходный базовый слой в вашей истории.

В следующем листинге мы скачиваем, устанавливаем и используем команду `docker-squash` для сравнения образов до и после сжатия.

Листинг 14.12. Использование команды docker_squash для уменьшения слоев образа

Устанавливает docker-squash. (Возможно, вам придется обратиться к https://github.com/jwilder/docker-squash для получения последних инструкций по установке.)

Сохраняет образ в файл формата TAR, с которым работает docker-squash, а затем загружает полученный образ, помечая его как «secret_build_squashed»

```
$ wget -qO- https://github.com/jwilder/docker-squash/releases/download
➥ /v0.2.0/docker-squash-linux-amd64-v0.2.0.tar.gz | \
  tar -zxvf - && mv docker-squash /usr/local/bin
$ docker save secret_build:latest | \
  docker-squash -t secret_build_squashed | \
  docker load
$ docker history secret_build_squashed
IMAGE          CREATED         CREATED BY                                      SIZE
ee41518cca25   2 seconds ago   /bin/sh -c #(nop) CMD ["/bin/sh" "              0 B
b1c283b3b20a   2 seconds ago   /bin/sh -c #(nop) CMD ["/bin/bash               0 B
f443d173e026   2 seconds ago   /bin/sh -c #(squash) from 93c22f56              2.647 kB
93c22f563196   2 weeks ago     /bin/sh -c #(nop) ADD file:7529d28              128.9 MB
$ docker history secret_build
IMAGE          CREATED         CREATED BY                                      SIZE
b8a62a826ddf   3 seconds ago   /bin/sh -c rm /secret_file                      0 B
2a4238c53408   3 seconds ago   /bin/sh -c cat /secret_file                     0 B
a00886ff1240   9 seconds ago   /bin/sh -c #(nop) ADD file:69e77f6              10 B
5b39a3cba0b0   9 seconds ago   /bin/sh -c #(nop) CMD ["/bin/sh" "              0 B
08881219da4a   2 weeks ago     /bin/sh -c #(nop) CMD ["/bin/bash               0 B
```

В истории сжатого образа нет записи secret_file

В исходном образе все еще есть secret_file

```
6a4ec4bddc58   2 weeks ago     /bin/sh -c mkdir -p /run/systemd &   7 B
98697477f76a   2 weeks ago     /bin/sh -c sed -i 's/^#\s*\(deb.*u   1.895 kB
495ec797e6ba   2 weeks ago     /bin/sh -c rm -rf /var/lib/apt/lis   0 B
e3aa81f716f6   2 weeks ago     /bin/sh -c set -xe && echo '#!/bin   745 B
93c22f563196   2 weeks ago     /bin/sh -c #(nop) ADD file:7529d28   128.9 MB
$ docker run secret_build_squashed ls /secret_file
 ls: cannot access '/secret_file': No such file or directory
$ docker run f443d173e026 ls /secret_file
 ls: cannot access '/secret_file': No such file or directory
```

Демонстрирует, что в «сжатом» слое сжатого образа нет secret_file

Демонстрирует, что файл secret_file отсутствует в сжатом образе

Примечание относительно «отсутствующих» слоев образа

Docker изменил характер концепции слоев в Docker 1.10. С этого момента скачанные образы отображаются в истории как «отсутствующие». Это ожидаемо и связано с изменениями, внесенными Docker для повышения безопасности историй образов.

Вы по-прежнему можете получить содержимое загруженных слоев с помощью команды `docker save`, а затем извлекая TAR-файлы из этого файла TAR. Вот пример сеанса, который делает это для уже скачанного образа Ubuntu.

Листинг 14.13. «Отсутствующие» слои в загруженных образах

Использует команду docker history, чтобы показать слой истории образа Ubuntu

Использует команду docker save для вывода TAR-файла слоев образа, который передается прямо в tar и извлекается

```
$ docker history ubuntu
IMAGE          CREATED        CREATED BY                                      SIZE
104bec311bcd   2 weeks ago    /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>      2 weeks ago    /bin/sh -c mkdir -p /run/systemd && ech         7 B
<missing>      2 weeks ago    /bin/sh -c sed -i 's/^#\s*\(deb.*univer         1.9 kB
<missing>      2 weeks ago    /bin/sh -c rm -rf /var/lib/apt/lists/*          0 B
<missing>      2 weeks ago    /bin/sh -c set -xe && echo '#!/bin/sh           745 B
<missing>      2 weeks ago    /bin/sh -c #(nop) ADD file:7529d28035b4         129 MB
$ docker save ubuntu | tar -xf -
$ find . | grep tar$
./042e55060780206b2ceabe277a8beb9b10f48262a876fd21b495af318f2f2352/layer.tar
./1037e0a8442d212d5cc63d1bc706e0e82da0eaafd62a2033959cfc629f874b28/layer.tar
./25f649b30070b739bc2aa3dd877986bee4de30e43d6260b8872836cdf549fcfc/layer.tar
./3094e87864d918dfdb2502e3f5dc61ae40974cd957d5759b80f6df37e0e467e4/layer.tar
./41b8111724ab7cb6246c929857b0983a016f11346dcb25a551a778ef0cd8af20/layer.tar
./4c3b7294fe004590676fa2c27a9a952def0b71553cab4305aeed4d06c3b308ea/layer.tar
./5d1be8e6ec27a897e8b732c40911dcc799b6c043a8437149ab021ff713e1044f/layer.tar
./a594214bea5ead6d6774f7a09dbd7410d652f39cc4eba5c8571d5de3bcbe0057/layer.tar
```

```
./b18fcc335f7aeefd87c9d43db2888bf6ea0ac12645b7d2c33300744c770bcec7/layer.tar
./d899797a09bfcc6cb8e8a427bb358af546e7c2b18bf8e2f7b743ec36837b42f2/layer.tar
./ubuntu.tar
$ tar -tvf
➥ ./4c3b7294fe004590676fa2c27a9a952def0b71553cab4305aeed4d06c3b308ea
➥ /layer.tar
drwxr-xr-x 0 0        0           0 15 Dec 17:45 etc/
drwxr-xr-x 0 0        0           0 15 Dec 17:45 etc/apt/
-rw-r--r-- 0 0        0        1895 15 Dec 17:45 etc/apt/sources.list
```

Демонстрирует, что TAR-файлы содержат только изменения файла в этом слое

Хотя это и несколько похоже на метод 52, использование специализированного средства имеет некоторые заметные различия в конечном результате. В предыдущем решении видно, что слои метаданных, такие как `CMD`, были сохранены, в то время как в предыдущем методе они полностью отбрасывались, поэтому вам нужно было вручную воссоздавать эти слои метаданных через другой Dockerfile.

Это означает, что утилита docker-squash может использоваться для автоматической очистки образов по мере их поступления в реестр, если вы склонны не доверять своим пользователям правильное использование секретных данных в сборках образов – все они должны работать нормально.

Тем не менее следует опасаться, что ваши пользователи будут размещать секреты в любых слоях метаданных, – переменные среды, в частности, представляют собой угрозу и вполне могут быть сохранены в окончательном образе.

МЕТОД 99 OpenShift: платформа приложений как сервис

OpenShift – это продукт под управлением компании Red Hat, который позволяет организации запускать платформу приложений в качестве службы (aPaaS). Он предлагает группам разработчиков приложений платформу для запуска кода, не заботясь о деталях аппаратного обеспечения. Версия 3 этого продукта была полностью переписана на языке Go, используя Docker в качестве технологии контейнеров, а Kubernetes и etcd – для оркестровки. Помимо этого, Red Hat добавила корпоративные функции, которые позволяют легче развертывать его в корпоративной среде, ориентированной на безопасность.

Хотя в OpenShift есть много функций, о которых можно было бы рассказать, мы будем использовать его здесь как средство управления безопасностью, лишив пользователя возможности прямого запуска Docker, но сохранив при этом преимущества использования Docker.

OpenShift доступен как в виде продукта, поддерживаемого предприятием, так и в виде проекта с открытым исходным кодом под названием Origin, по адресу https://github.com/openshift/origin.

ПРОБЛЕМА

Вы хотите контролировать угрозу безопасности со стороны ненадежных пользователей, вызывающих команду `docker run`.

РЕШЕНИЕ

Используйте средство aPaaS для управления и обеспечения взаимодействия с Docker через интерфейс прокси.

У aPaaS есть много преимуществ, но здесь мы сосредоточимся на его способности управлять правами доступа пользователей и запускать контейнеры Docker от имени пользователя, обеспечивая безопасную точку аудита для тех, кто запускает контейнеры Docker.

Почему это важно? Пользователи, использующие этот aPaaS, не имеют прямого доступа к команде `docker`, поэтому они не могут нанести ущерб, не нарушив безопасность, которую обеспечивает OpenShift. Например, контейнеры развертываются обычными пользователями по умолчанию, а для того, чтобы преодолеть это, требуются права, которые должны быть предоставлены администратором. Если вы не можете доверять своим пользователям, применение aPaaS – это эффективный способ предоставить им доступ к Docker.

> **СОВЕТ.** APaaS предоставляет пользователям возможность запускать приложения по требованию для разработки, тестирования или эксплуатации. Docker естественным образом подходит для этих сервисов, так как он обеспечивает надежный и изолированный формат доставки приложений, позволяя рабочей группе позаботиться о деталях развертывания.

Говоря кратко, OpenShift опирается на Kubernetes (см. метод 88), но добавляет функции для предоставления полноценного aPaaS. Эти дополнительные функции включают в себя:

- управление пользователями;
- предоставление прав;
- квоты;
- контексты безопасности;
- маршрутизация.

Полный обзор установки OpenShift выходит за рамки этой книги. Если вам нужна автоматическая установка с использованием Vagrant, которую мы поддерживаем, см. https://github.com/docker-in-practice/shutit-openshift-origin. Если нужна помощь в установке Vagrant, см. приложение C.

Другие параметры, такие как установка только для Docker (только для одного узла) или полная ручная сборка, доступны и документированы в базе кода OpenShift Origin на странице https://github.com/openshift/origin.

> **ПОДСКАЗКА.** OpenShift Origin – это upstream-версия OpenShift. *Upstream* означает, что это кодовая база, из которой Red Hat берет изменения для OpenShift, своего поддерживаемого предложения. Origin –

ПО с открытым исходным кодом и может использоваться любым желающим, но рекомендуемая версия Red Hat продается и поддерживается как OpenShift. Upstream-версия обычно более продвинутая, но менее стабильная.

Приложение OpenShift

В этом методе мы покажем простой пример создания, сборки, запуска и доступа к приложению с помощью веб-интерфейса OpenShift. Это будет базовое приложение на NodeJS, которое обслуживает простую веб-страницу.

Приложение будет использовать Docker, Kubernetes и S2I. Docker используется для инкапсуляции сред сборки и развертывания. Метод сборки Source to Image (S2I) – это метод, применяемый Red Hat в OpenShift для сборки контейнера Docker, а Kubernetes используется для запуска приложения в кластере OpenShift.

Вход

Для начала запустите `./run.sh` из папки shutit-openshift-origin, а затем перейдите по адресу https: //localhost:8443, минуя все предупреждения безопасности. Вы увидите страницу входа, показанную на рис. 14.3. Обратите внимание, что, если вы используете Vagrant, необходимо запустить веб-браузер на вашей виртуальной машине (см. приложение C для получения справки о получении графического интерфейса пользователя к вашей виртуальной машине).

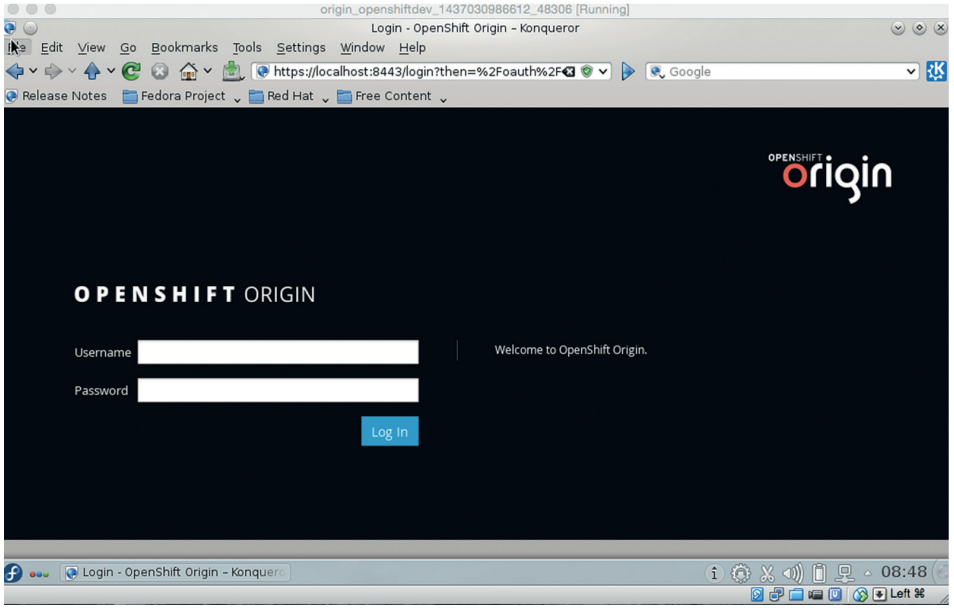


Рис. 14.3 ❖ Страница входа в OpenShift

Войдите с именем пользователя `hal-1` и любым паролем.

Создание приложения на NodeJS

Вы вошли в OpenShift как разработчик (см. рис. 14.4).

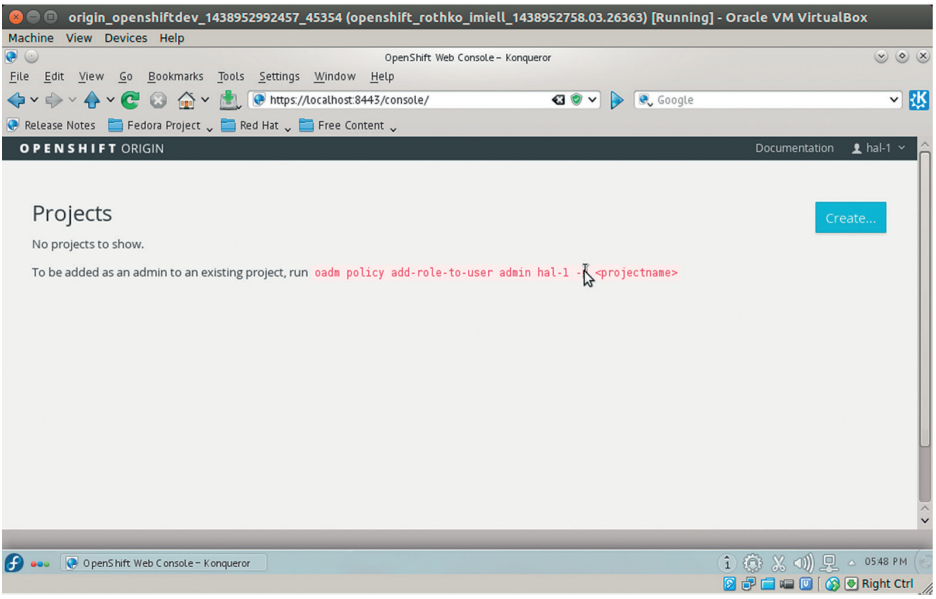


Рис. 14.4 ❖ Страница Projects

Создайте проект, нажав кнопку **Create** (Создать). Заполните форму, как показано на рис. 14.5. После этого нажмите **Создать** еще раз.

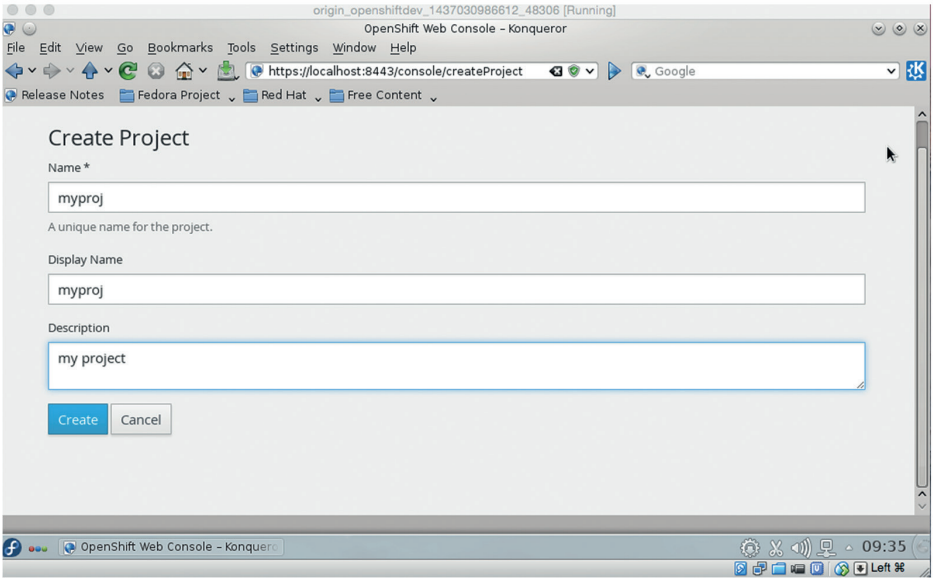


Рис. 14.5 ❖ Страница создания проекта

Как только проект будет настроен, снова нажмите кнопку **Создать** и введите предложенный репозиторий GitHub (https://github.com/openshift/nodejs-ex), как на рис. 14.6.

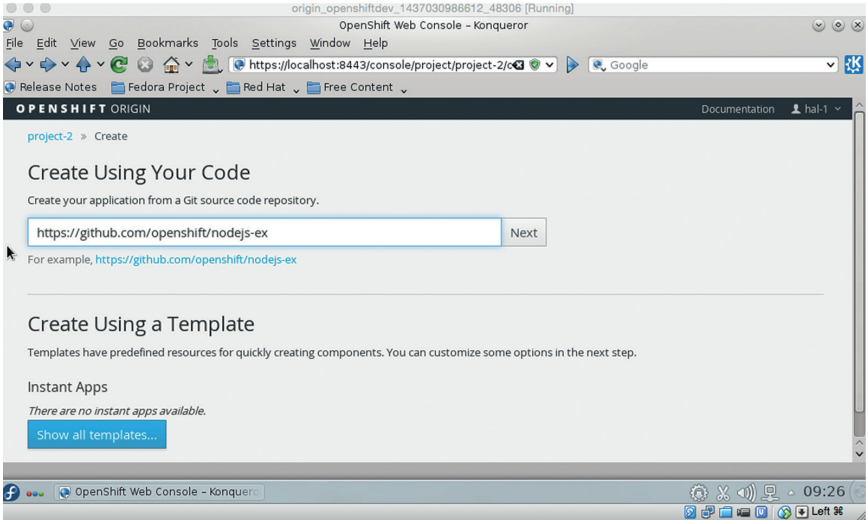


Рис. 14.6 ❖ Страница исходного кода

Нажмите кнопку **Next**, и вам будет предоставлен выбор образов-сборщиков, как на рис. 14.7. Образ сборки определяет контекст, в котором будет создан код. Выберите образ NodeJS.

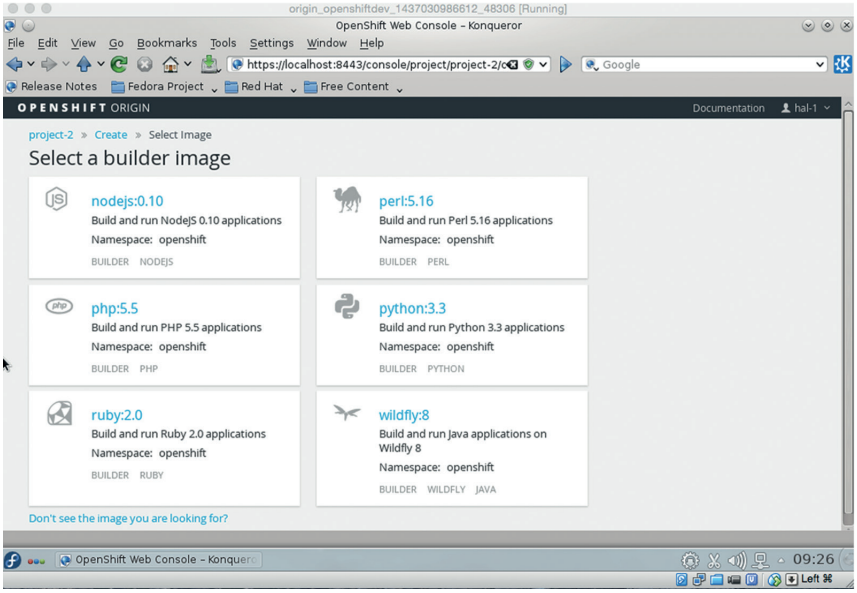


Рис. 14.7 ❖ Страница выбора сборщиков образов

Теперь заполните форму, как изображено на рис. 14.8. Нажмите кнопку **Создать** на NodeJS в нижней части страницы при прокрутке формы вниз.

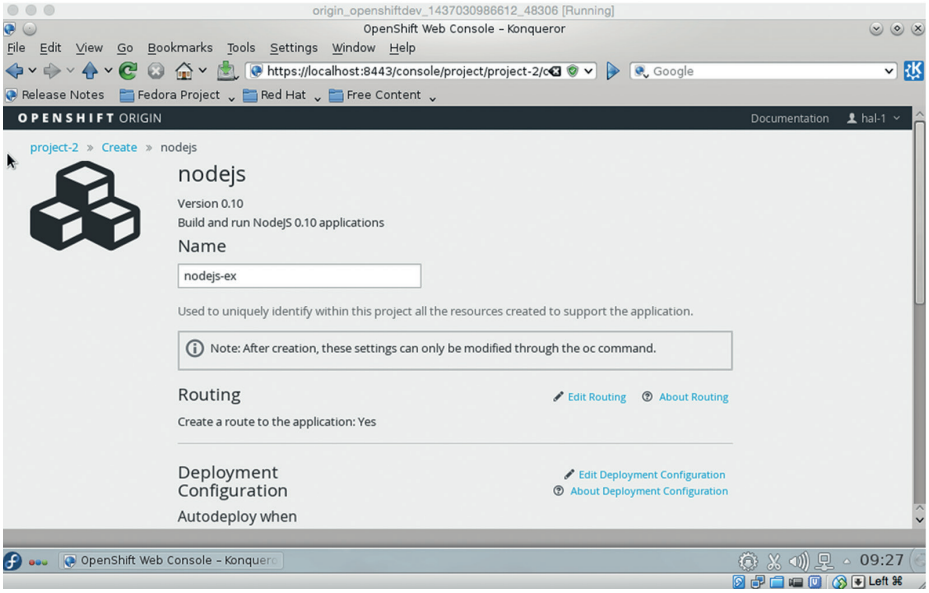


Рис. 14.8 ❖ Форма шаблона NodeJS

Через несколько минут вы должны увидеть экран, подобный тому, что показан на рис. 14.9.

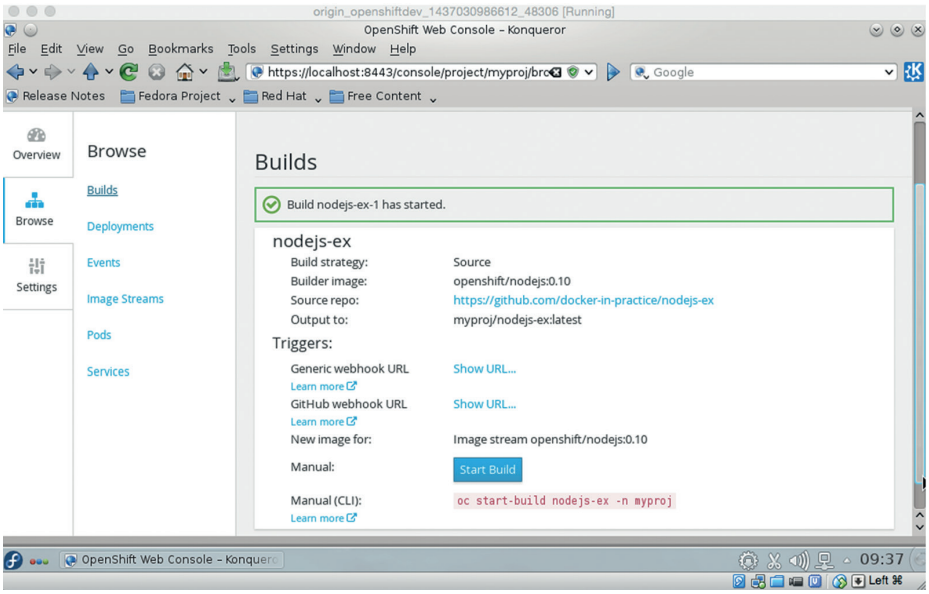


Рис. 14.9 ❖ Страница запуска сборки

Спустя несколько секунд, пролистав страницу вниз, вы увидите, что сборка началась (рис. 14.10).

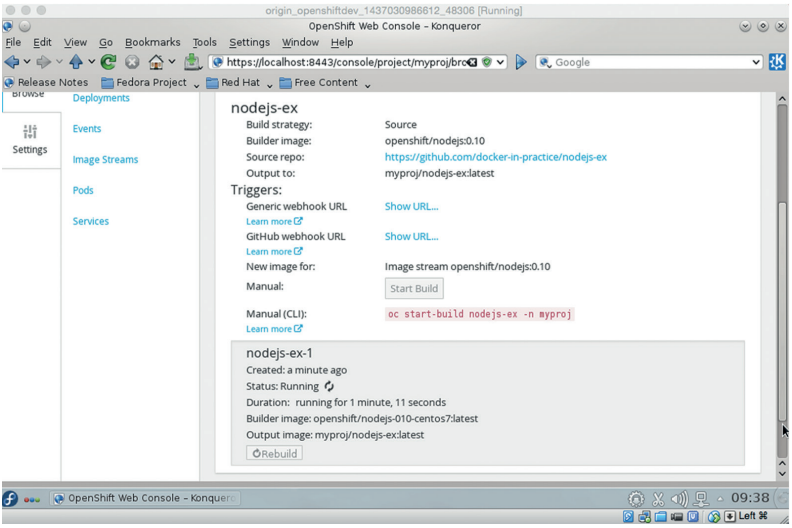


Рис. 14.10 ❖ Окно сведений о сборке

ПОДСКАЗКА. В ранних версиях OpenShift сборка иногда не начиналась автоматически. Если такое произошло, через несколько минут нажмите кнопку **Start Build** (Начать сборку).

Через некоторое время вы увидите, что приложение работает, как на рис. 14.11.

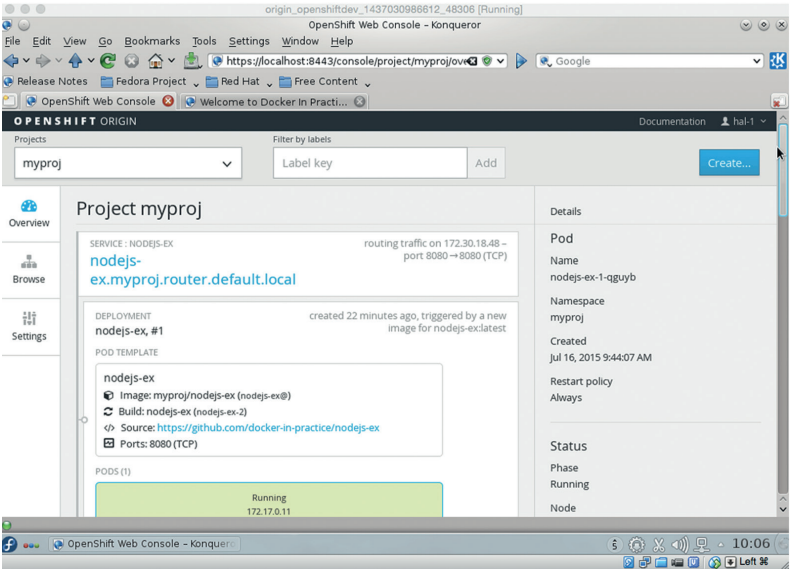


Рис. 14.11 ❖ Страница запуска приложения

Нажав **Browse** и **Pods**, вы увидите, что модуль был развернут(рис. 14.12).

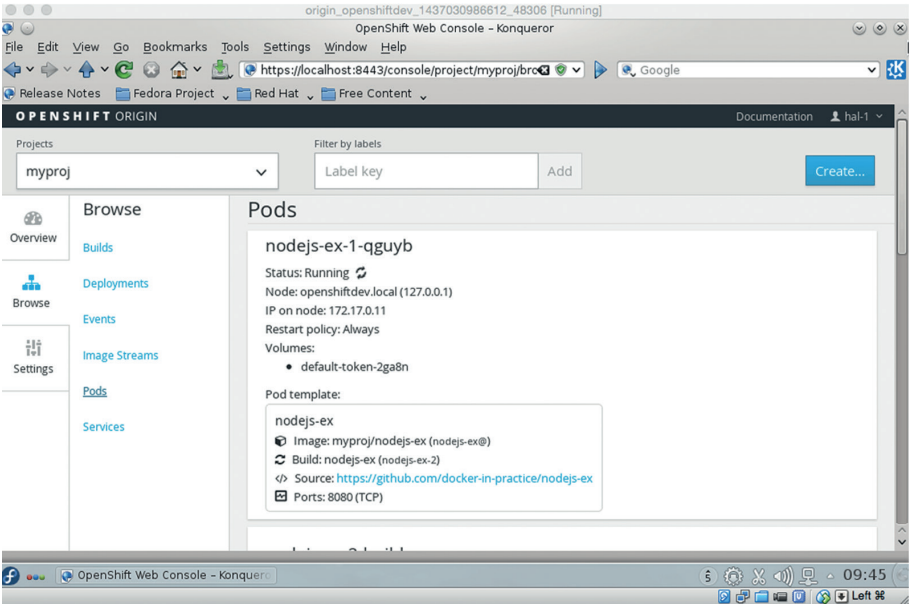


Рис. 14.12 ❖ Список модулей OpenShift

СОВЕТ. Смотрите метод 88, где объясняется, что такое модуль.

Как получить доступ к своему модулю? Если вы посмотрите на вкладку **Services** (Службы) (см. рис. 14.13), то увидите IP-адрес и номер порта для доступа.

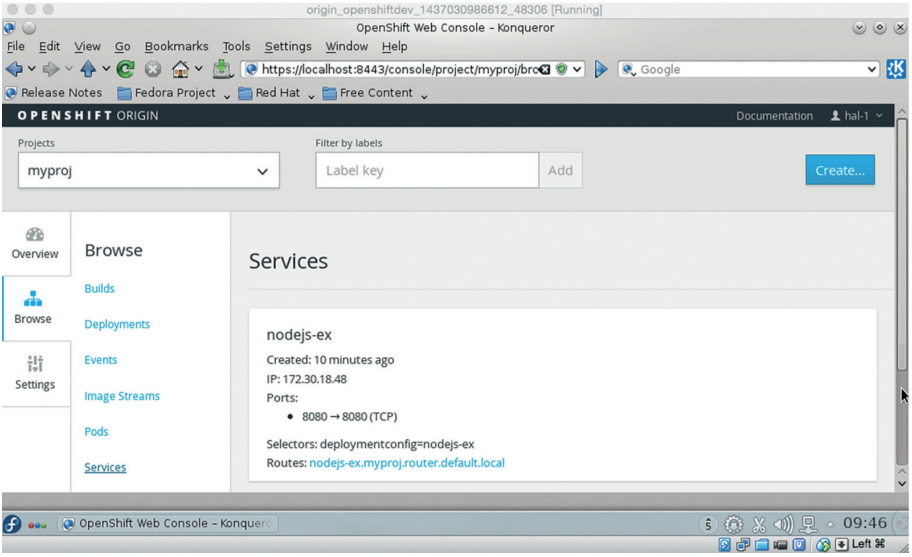


Рис. 14.13 ❖ Подробности службы приложения NodeJS

Введите этот адрес в строке браузера и – вуаля! – у вас имеется приложение NodeJS, как на рис. 14.14.

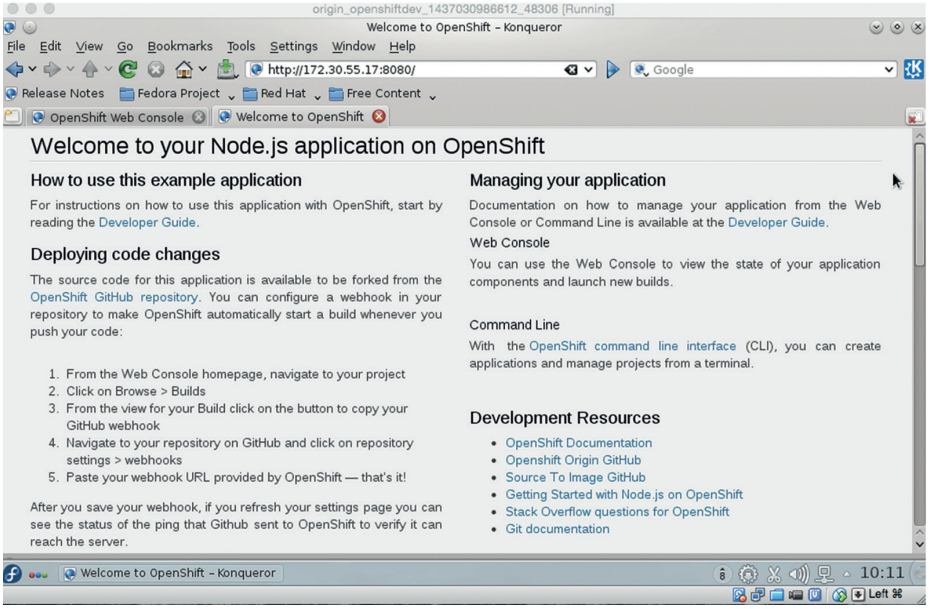


Рис. 14.14 ❖ Целевая страница приложения NodeJS

ОБСУЖДЕНИЕ

Давайте вспомним, чего мы здесь добились и почему это важно для безопасности.

С точки зрения пользователя, они вошли в веб-приложение и развернули приложение, используя технологии на основе Docker, не подходя к Dockerfile или команде `docker run`.

Администратор OpenShift может:

- контролировать доступ пользователей;
- ограничить использование ресурсов проектом;
- централизованно предоставить ресурсы;
- убедиться, что код запускается с непривилегированным статусом по умолчанию.

Это гораздо более безопасно, чем предоставление пользователям прямого доступа к команде `docker run`.

Если вы хотите использовать это приложение и увидеть, как aPaaS облегчает итеративный подход, можете разветвить Git-репозиторий, изменить в нем код, а затем создать новое приложение. Мы сделали это здесь: https://github.com/docker-in-practice/nodejs-ex.

Чтобы узнать больше об OpenShift, посетите страницу http://www.openshift.org.

МЕТОД 100 Использование параметров безопасности

Вы уже видели в предыдущих методах, что по умолчанию вы получаете права суперпользователя в контейнере Docker, и что этот пользователь тот же, что и пользователь root на хосте. Чтобы было проще, мы показали вам, что мандаты этого пользователя могут быть уменьшены, так что, даже если он выходит из контейнера, все равно есть действия, которые ядро не разрешит этому пользователю выполнить.

Но вы можете пойти дальше. Используя флаг параметров безопасности Docker, можно защитить ресурсы на хосте от воздействия того, что происходит внутри контейнера. При данном ограничении контейнер будет воздействовать только на ресурсы, на которые он получил разрешение от хоста.

ПРОБЛЕМА

Вы хотите обезопасить свой хост от действий контейнеров.

РЕШЕНИЕ

Используйте SELinux, чтобы наложить ограничения на контейнеры.

Здесь мы будем применять SELinux в качестве средства мандатного управления доступом (MAC), поддерживаемого ядром. SELinux – более или менее отраслевой стандарт и, скорее всего, будет использоваться организациями, которые особенно заботятся о безопасности. Первоначально он был разработан АНБ для защиты своих систем, а впоследствии был раскрыт его исходный код. Он используется в системах на базе Red Hat в качестве стандарта.

SELinux – обширная тема, поэтому мы не можем подробно рассказать о ней в этой книге, но покажем вам, как написать и применять простую политику, чтобы вы поняли, как она работает. Вы можете пойти дальше и поэкспериментировать, если вам нужно.

> **ПОДСКАЗКА.** Средства мандатного управления доступом (MAC) в Linux обеспечивают соблюдение правил безопасности помимо стандартных, к которым вы, возможно, привыкли. Говоря кратко, они обеспечивают соблюдение не только *обычных* правил чтения-записи-выполнения файлов и процессов, но и более тонких, которые можно применять к процессам на уровне ядра. Например, процессу MySQL может быть разрешено записывать файлы только в определенных каталогах, таких как /var/lib/mysql. Эквивалентный стандарт для систем на основе Debian – AppArmor.

Данный метод предполагает, что у вас есть хост с поддержкой SELinux. Это означает, что вы должны сначала установить SELinux (при условии, что он еще не установлен). Если вы используете Fedora или другую систему на базе Red Hat, скорее всего, он у вас уже есть.

Чтобы определить, включен ли SELinux, выполните команду `sestatus`:

```
# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   permissive
Mode from config file:          permissive
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
```

Первая строка вывода сообщит вам, включен ли SELinux. Если команда недоступна, на вашем хосте SELinux не установлен.

Также необходимо иметь соответствующие средства для создания политики SELinux. Например, на компьютере с поддержкой yum вам нужно будет выполнить команду `yum -y install selinux-policy-devel`.

SELinux на машине с Vagrant

Если у вас нет SELinux и вы хотите, чтобы он был создан для вас, можете использовать скрипт ShutIt для создания виртуальной машины на вашем хост-компьютере с предварительно установленными Docker и SELinux. Что он делает, объяснено на высоком уровне на рис. 14.15.

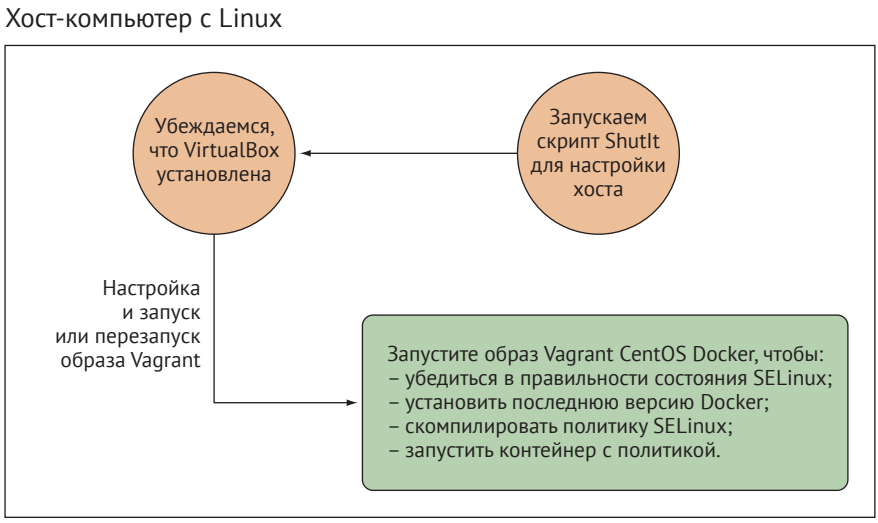


Рис. 14.15 ❖ Скрипт для подготовки виртуальной машины SELinux

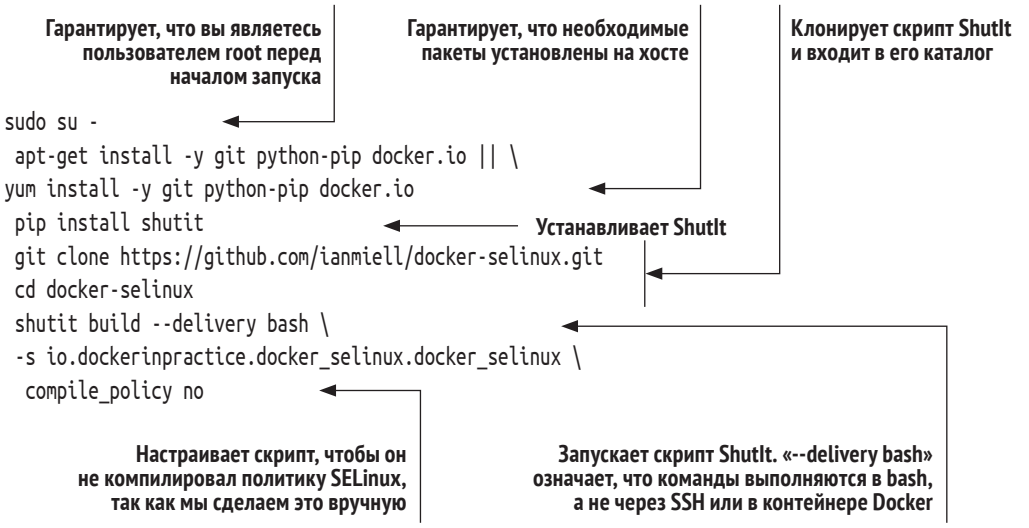
ПОДСКАЗКА. ShutIt – это универсальный инструмент автоматизации оболочки, который мы создали, чтобы преодолеть ограничения файлов Dockerfile. Если вы хотите узнать больше о нем, посетите страницу на GitHub http://ianmiell.github.io/shutit.

На рис. 14.5 показаны шаги, необходимые для настройки политики. Вот что сделает скрипт:

1. Настроит VirtualBox.
2. Запустит соответствующий образ Vagrant.
3. Войдет в виртуальную машину.
4. Убедится, что состояние SELinux правильное.
5. Установит последнюю версию Docker.
6. Установит инструменты разработки политики SELinux.
7. Даст вам оболочку.

Вот команды для его настройки и запуска (протестировано на дистрибутивах Debian и Red Hat):

Листинг 14.14. Установка ShutIt



После запуска этого скрипта вы должны в конечном итоге увидеть результат, как тот, что приведен ниже:

```
Pause point:
Have a shell:
You can now type in commands and alter the state of the target.
Hit return to see the prompt
Hit CTRL and ] at the same time to continue with build
Hit CTRL and u to save the state
```

Теперь у вас есть оболочка, работающая внутри виртуальной машины с установленным на ней SELinux. Если вы наберете `sestatus`, то увидите, что SELinux включен в разрешающем режиме (как показано в листинге 14.14). Чтобы вернуться в оболочку вашего хоста, нажмите **Ctrl-]**.

Компиляция политики SELinux

Независимо от того, использовали ли вы скрипт ShutIt или нет, мы предполагаем, что теперь у вас есть хост с включенным SELinux. Наберите `sestatus`, чтобы получить сводку состояния.

Листинг 14.15. Состояние SELinux после установки и включения

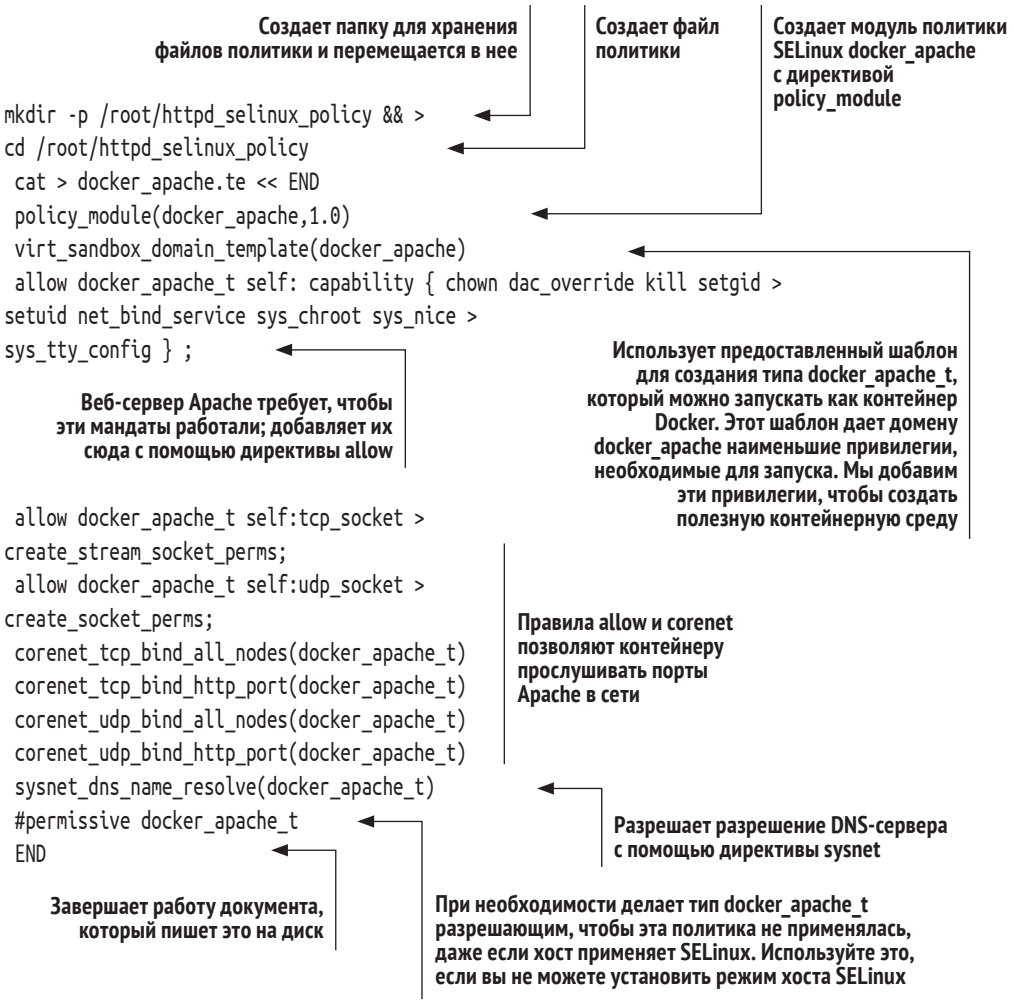
```
# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   permissive
Mode from config file:          permissive
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
```

В данном случае мы находимся в разрешающем режиме. Это означает, что SELinux регистрирует нарушения безопасности в журналах, но не применяет их. Это хорошо для безопасного тестирования новых политик без превращения вашей системы в непригодную для использования. Чтобы перевести статус SELinux в разрешающий, наберите `setenforce Permissive` от имени пользователя root. Если вы не можете сделать это на своем хосте из соображений безопасности, не беспокойтесь: есть возможность установить политику как разрешающую, которая описана в листинге 14.15.

> **ПРИМЕЧАНИЕ.** Если вы устанавливаете SELinux и Docker на хосте самостоятельно, убедитесь, что для демона Docker в качестве флага задан параметр `--selinux-enabled`. Это можно проверить с помощью команды `ps -ef | grep 'docker -d. * - selinux-enabled`, которая должна возвращать процесс проверки соответствия на выходе.

Создайте папку для вашей политики и перейдите к ней. Затем создайте файл политики со следующим содержимым от имени пользователя root с именем docker_apache.te. Этот файл содержит политику, которую мы попытаемся применить.

Листинг 14.16. Создание политики SELinux



СОВЕТ. Для получения дополнительной информации о предыдущих правах доступа, а также для изучения других вы можете установить пакет selinux-policy-doc и использовать браузер для просмотра документации по файлу: ///usr/share/doc-base/selinux-policy-doc/html/index.html. Эти документы также доступны онлайн на странице http://oss.tresys.com/docs/refpolicy/api/.

Теперь вы скомпилируете политику и увидите, что ваше приложение не запускается при этой политике в принудительном режиме. Затем перезапустите его в разрешающем режиме, чтобы проверить нарушения и исправить их позже:

```
$ make -f /usr/share/selinux/devel/Makefile \
docker_apache.te
Compiling targeted docker_apache module
/usr/bin/checkmodule: loading policy configuration from >
tmp/docker_apache.tmp
/usr/bin/checkmodule: policy configuration loaded
/usr/bin/checkmodule: writing binary representation (version 17) >
to tmp/docker_apache.mod
Creating targeted docker_apache.pp policy package
rm tmp/docker_apache.mod tmp/docker_apache.mod.fc
$ semodule -i docker_apache.pp
$ setenforce Enforcing
$ docker run -ti --name selinuxdock >
--security-opt label:type:docker_apache_t httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
2a341c7141bd: Pull complete
[...]
Status: Downloaded newer image for httpd:latest
permission denied
Error response from daemon: Cannot start container >
650c446b20da6867e6e13bdd6ab53f3ba3c3c565abb56c4490b487b9e8868985: >
[8] System error: permission denied
$ docker rm -f selinuxdock
selinuxdock
$ setenforce Permissive
$ docker run -d --name selinuxdock >
--security-opt label:type:docker_apache_t httpd
```

Компилирует файл docker_apache.te в двоичный модуль SELinux с суффиксом .pp

Устанавливает модуль

Устанавливает режим SELinux в значение «принудительный»

Запускает образ httpd в качестве демона, применяя тип метки безопасности docker_apache_t, который вы определили в модуле SELinux. Эта команда должна завершиться неудачно, потому что это нарушает конфигурацию безопасности SELinux

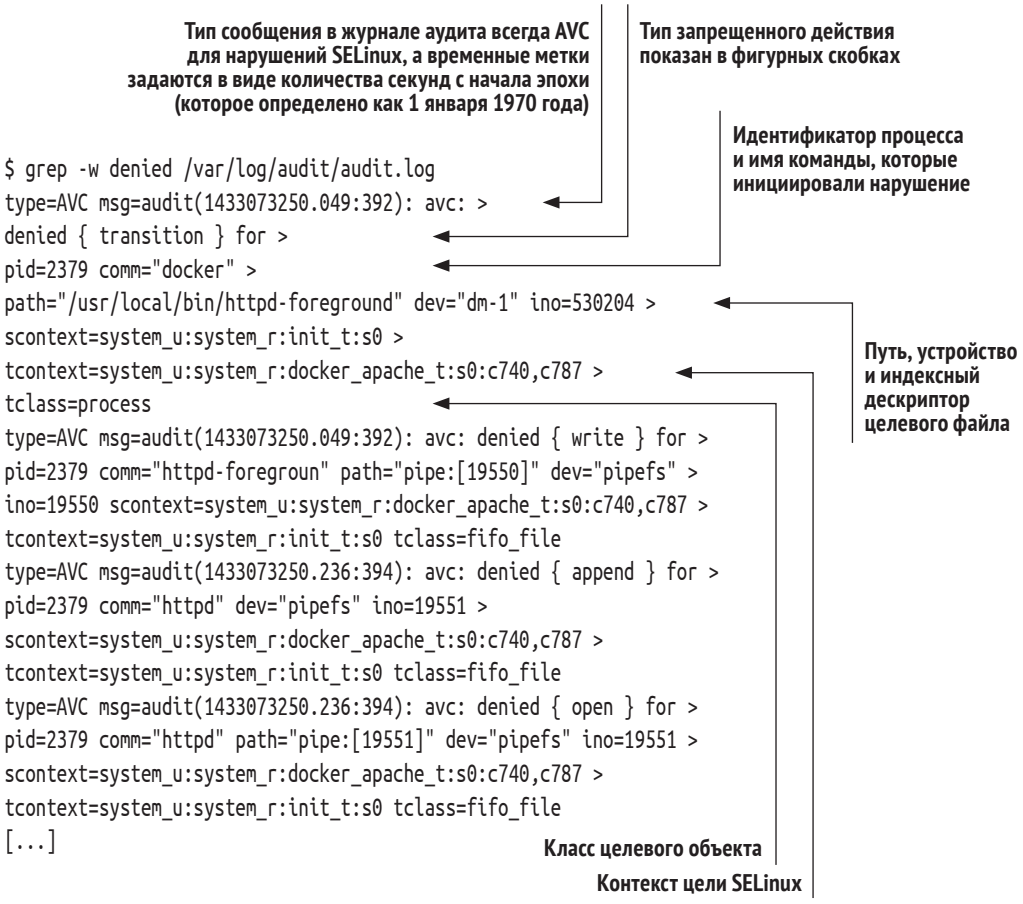
Удаляет вновь созданный контейнер

Устанавливает режим SELinux в «разрешающий», чтобы разрешить запуск приложения

Запускает образ httpd в качестве демона, применяя тип метки безопасности docker_apache_t, который вы определили в модуле SELinux. Эта команда должна работать успешно

Проверка нарушений

До этого момента вы создали модуль SELinux и применили его к своему хосту. Поскольку принудительный режим SELinux на этом хосте установлен как разрешающий, действия, которые будут запрещены в принудительном режиме, разрешены в строке в журнале аудита. Вы можете проверить эти сообщения, выполнив следующую команду:



Уф! Здесь много жаргона, и у нас нет времени, чтобы научить вас всему, что может понадобиться знать о SELinux. При желании узнать больше лучше начать с документации Red Hat по SELinux: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/deployment_guide/ch-selinux.

На данный момент вам нужно проверить, что в нарушениях нет ничего плохого. Что может выглядеть плохо? Если приложение пытается открыть порт или файл, который вы не ожидали, вы дважды подумаете о том, что будет дальше: мы исправим эти нарушения с помощью нового модуля SELinux.

В данном случае хорошо, что httpd может выполнять запись в каналы. Мы выяснили, что именно это и не давал делать SELinux, поскольку упомянутые «запрещенные» действия – это `append`, `write` и `open` для файлов pipefs на виртуальной машине.

Исправление нарушений в SELinux

После того как вы решили, что замеченные нарушения приемлемы, можно использовать инструменты, способные автоматически генерировать файл политики, который вам нужно применить, так что не придется проходить через боль и риск и писать его самостоятельно. В следующем примере для этого используется audit2allow.

Листинг 14.17. Создание новой политики SELinux

```
mkdir -p /root/selinux_policy_httpd_auto
cd /root/selinux_policy_httpd_auto
audit2allow -a -w
audit2allow -a -M newmodname create policy
semodule -i newmodname.pp
```

Создает новую папку для хранения нового модуля SELinux

Использует audit2allow для отображения политики, которая будет генерироваться из чтения журналов аудита. Просмотрите это снова, чтобы быть уверенными, что это выглядит разумно

Создает ваш модуль с флагом -M и имя для выбранного вами модуля

Устанавливает модуль из вновь созданного файла .pp

Важно понимать, что этот новый созданный нами модуль SELinux «включает в себя» (или «требует») и изменяет тот, который мы создали ранее, путем обращения и добавления полномочий к типу docker_apache_t. Вы можете объединить их в полную и дискретную политику в одном файле .te, если захотите.

Тестирование нового модуля

Теперь, когда у вас установлен новый модуль, можете попробовать снова включить SELinux и перезапустить контейнер.

> **СОВЕТ.** Если вы не смогли настроить свой хост как разрешающий ранее (и добавили строку хеширования в исходный файл docker_apache.te), перекомпилируйте и переустановите файл docker_apache.te (с хешированной разрешающей строкой), прежде чем продолжить.

Листинг 14.18. Запуск контейнера с ограничениями SELinux

```
docker rm -f selinuxdock
setenforce Enforcing
docker run -d --name selinuxdock \
--security-opt label:type:docker_apache_t httpd
docker logs selinuxdock
grep -w denied /var/log/audit/audit.log
```

В журнале аудита не должно быть новых ошибок. Ваше приложение было запущено в контексте этого режима SELinux.

ОБСУЖДЕНИЕ

SELinux имеет репутацию сложной и трудной в управлении системы, и нередко слышны жалобы на то, что его чаще выключают, чем отлаживают.

Вряд ли это абсолютно безопасно. Хотя тонкости SELinux требуют серьезных усилий для их освоения, мы надеемся, что этот метод показал вам, как создать что-то, что эксперт по безопасности может проверить, – и в идеале подписать, – если Docker не приемлем из коробки.

Резюме

- Вы можете детально контролировать возможности пользователя root в своих контейнерах с помощью мандатов.
- Вы можете аутентифицировать людей, используя API Docker через HTTP.
- Docker обладает встроенной поддержкой шифрования API с использованием сертификатов.
- SELinux – это проверенный способ снижения опасности контейнеров, работающих от имени пользователя root.
- Платформа приложения в качестве службы (aPaaS) может использоваться для управления доступом к среде исполнения Docker.

Глава 15

Как по маслу: запуск Docker в рабочем окружении

О чем рассказывается в этой главе:

- варианты записи вывода контейнера в журнал;
- мониторинг работающих контейнеров;
- управление использованием ресурсов контейнеров;
- использование возможностей Docker для управления традиционными задачами системного администратора.

В этой главе мы рассмотрим некоторые вопросы, возникающие при запуске в рабочем окружении. Запуск Docker в рабочем окружении – обширная тема, и эксплуатация Docker – область, которая еще развивается. Многие основные инструментальные средства находятся на ранней стадии разработки, и они менялись, когда мы писали первое и второе издания этой книги.

В этой главе мы сосредоточимся на том, чтобы показать вам те ключевые моменты, которые следует учитывать при переходе от нестабильных сред к стабильным.

15.1. Мониторинг

Запуская Docker в рабочем окружении, первое, что вы захотите рассмотреть, – это как отследить и оценить, что задумали ваши контейнеры. В данном разделе вы узнаете, как получить оперативный обзор активности ваших активных контейнеров и их производительности.

Это все еще развивающийся аспект экосистемы Docker, но некоторые средства и методы становятся все более распространенными по сравнению с другими. Мы рассмотрим перенаправление журналов приложений в системный журнал хоста, перенаправление вывода команды docker logs в одно место и cAdvisor, средство мониторинга производительности, ориентированное на контейнеры от компании Google.

МЕТОД 101 Логирование контейнеров в системный журнал хоста

В дистрибутивах Linux обычно запускается демон syslog. Он является серверной частью функции регистрации системы – приложения отправляют сообщения этому демону вместе с метаданными, такими как важность сообщения, и демон решает, где сохранить сообщение (если оно вообще есть). Эта функциональность используется целым рядом приложений, от менеджеров сетевых подключений до самого ядра, сбрасывающего информацию при обнаружении ошибки.

Поскольку он настолько надежен и широко используется, целесообразно, чтобы приложения, которые вы пишете сами, регистрировались в системном журнале. К сожалению, это не будет работать после того, как вы контейнеризируете свое приложение (поскольку по умолчанию в контейнерах нет демона syslog). Если вы решили запустить демона во всех своих контейнерах, нужно будет перейти к каждому отдельному контейнеру для извлечения журналов.

ПРОБЛЕМА

Вы хотите осуществлять сбор системных журналов централизованно на хосте Docker.

РЕШЕНИЕ

Запустите сервисный контейнер, который действует как демон syslog для контейнеров Docker.

Основная идея данного метода состоит в том, чтобы запустить сервисный контейнер, который запускает демон syslog, и совместно использовать точку взаимодействия логирования (/dev/log) через файловую систему хоста. Сам журнал можно получить, запросив контейнер Docker системного журнала, и он хранится в томе.

На рис. 15.1 показано, как /tmp/syslogdev в файловой системе хоста можно использовать в качестве точки взаимодействия для ведения всех системных журналов в контейнерах на хосте. Контейнеры, которые ведут журналы, монтируют и записывают свой системный журнал в это место, а контейнер syslogger разбирает все эти входы.

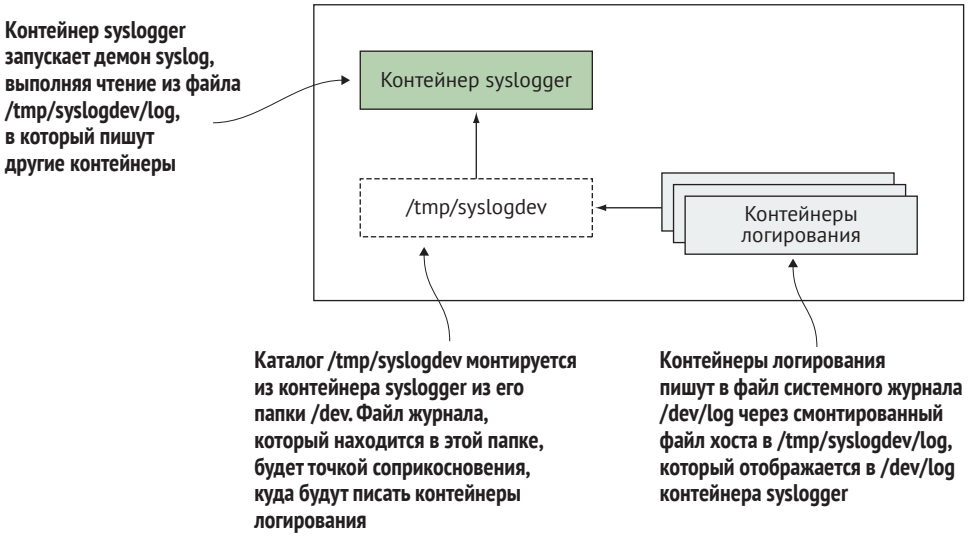


Рис. 15.1 ❖ Обзор централизованного ведения системных журналов контейнеров Docker

> **ПОДСКАЗКА.** Демон syslog – это процесс, который выполняется на сервере. Он собирает и осуществляет управление сообщениями, отправляемыми в центральный файл, который обычно является сокетом домена Unix. Обычно он использует /dev/log в качестве файла для получения сообщений файлов журнала.

Контейнер syslogger можно создать с помощью этого простого файла Dockerfile.

Листинг 15.1. Создание контейнера системного журнала

```
FROM ubuntu:14.043
RUN apt-get update && apt-get install rsyslog
 VOLUME /dev
 VOLUME /var/log
 CMD rsyslogd -n
```

Устанавливает пакет rsyslog, который делает доступной программу-демон rsyslogd. «r» означает «reliable»

Создает том /dev для совместного использования с другими контейнерами

Запускает процесс rsyslogd при запуске

Создает том /var/log для сохранения файла syslog

Затем вы создаете контейнер, помечая его тегом syslogger, и запускаете его:

```
docker build -t syslogger .
docker run --name syslogger -d -v /tmp/syslogdev:/dev syslogger
```

Вы смонтировали папку контейнера /dev на папку хоста /tmp/syslogdev, чтобы иметь возможность монтировать сокет /dev/log в каждый контейнер в качестве тома, как вы вскоре увидите. Контейнер продолжит работать в фоновом режиме, читая любые сообщения из файла /dev/log и обрабатывая их.

На хосте вы теперь увидите, что папка /dev контейнера syslog была смонтирована на папку хоста /tmp/syslogdev:

```
$ ls -1 /tmp/syslogdev/
fd
full
fuse
kcore
log
null
ptmx
random
stderr
stdin
stdout
tty
urandom
zero
```

Для демонстрации мы запустим 100 контейнеров демонов, которые записывают свой собственный порядок запуска от 0 до 100 в системный журнал, используя команду `logger`. Потом вы сможете увидеть эти сообщения, выполнив на хосте команду `docker exec`, чтобы просмотреть файл системного журнала контейнера syslogger.

Сначала запустите контейнеры.

Листинг 15.2. Запуск контейнеров

```
for d in {1..100}
do
    docker run -d -v /tmp/syslogdev/log:/dev/log ubuntu logger hello_$d
done
```

Предыдущий монтируемый том связывает конечную точку системного журнала контейнера (/dev/log) с файлом хоста /tmp/syslogdev/log, который, в свою очередь, отображается в файл /dev/log контейнера syslogger. После этого все выходные данные системного журнала отправляются в один и тот же файл.

Когда это будет завершено, вы увидите нечто похожее на этот (отредактированный) вывод:

```
$ docker exec -ti syslogger tail -f /var/log/syslog
May 2511:51:25 f4fb5d829699 logger: hello
May 2511:55:15 f4fb5d829699 logger: hello_1
May 2511:55:15 f4fb5d829699 logger: hello_2
May 2511:55:16 f4fb5d829699 logger: hello_3
 [...]
May 2511:57:38 f4fb5d829699 logger: hello_97
May 2511:57:38 f4fb5d829699 logger: hello_98
May 2511:57:39 f4fb5d829699 logger: hello_99
```

Вы можете использовать измененную команду exec для архивирования этих системных журналов, если хотите. Например, можно выполнить следующую команду, чтобы получить все журналы на 11 часов 25 мая в виде сжатого файла:

```
$ docker exec syslogger bash -c "cat /var/log/syslog | \
grep '^May 2511'" | xz - > /var/log/archive/May25_11.log.xz
```

> **ПРИМЕЧАНИЕ.** Чтобы сообщения отображались в центральном контейнере системного журнала, ваши программы должны войти в системный журнал. Здесь мы гарантируем это, выполняя команду `logger`, но вашим приложениям нужно сделать то же самое, чтобы это работало. Большинство современных методов ведения журналов обладают средствами для записи в локально видимый системный журнал.

ОБСУЖДЕНИЕ

Вам может быть интересно, как различать сообщения журналов разных контейнеров с помощью этого метода. Здесь у вас есть несколько вариантов. Вы можете изменить логирование приложения, чтобы вывести имя хоста контейнера, или можете посмотреть следующий метод, чтобы Docker сделал эту тяжелую работу за вас.

> **ПРИМЕЧАНИЕ.** Данный метод похож на следующий, в котором используется драйвер системного журнала Docker, но есть отличие. Здесь выходные данные запущенных процессов контейнеров сохраняются как выходные данные команды `docker logs`, тогда как в следующем методе используется команда `logs`, что делает этот метод избыточным.

МЕТОД 102 Логирование вывода журналов Docker

Как вы уже видели, Docker предлагает базовую систему ведения журналов, которая собирает выходные данные команды запуска вашего контейнера. Если вы – системный администратор и запускаете множество служб с одного хоста,

оперативно отслеживать и собирать журналы, используя команду `docker logs` для каждого контейнера по очереди, может быть утомительно.

В этом методе мы рассмотрим функцию драйвера журнала Docker. Она позволяет использовать стандартные системы ведения журналов для отслеживания множества служб на одном хосте или даже на нескольких.

ПРОБЛЕМА

Вы хотите централизованно собирать выходные данные команды `docker logs` на вашем хосте Docker.

РЕШЕНИЕ

Используйте флаг `--log-driver`, чтобы перенаправить журналы в нужное место.

По умолчанию сбор журналов Docker осуществляется в демоне Docker, и вы можете получить к ним доступ с помощью команды `docker logs`. Как вы, вероятно, знаете, это показывает вывод основного процесса контейнера.

На момент написания Docker предлагает несколько вариантов перенаправления этого вывода на некоторые *драйверы журналов*, в том числе:

- syslog;
- journald;
- json-file.

Опция по умолчанию -json-file, а другие можно выбрать с использованием флага `--log-driver`. Параметры syslog и journald отправляют вывод журнала соответствующим демонам с теми же именами. Вы можете найти официальную документацию по всем доступным драйверам журнала на странице https://docs.docker.com/engine/reference/logging/.

> **ВНИМАНИЕ!** Для этого метода требуется Docker версии 1.6.1 или выше.

Демон syslog – это процесс, который выполняется на сервере. Он собирает и осуществляет управление сообщениями, отправляемыми в центральный файл, который обычно является сокетом домена Unix. Обычно он использует /dev/log в качестве файла для получения сообщений файлов журнала.

Journald – это системная служба, которая собирает и хранит данные логирования. Она создает и поддерживает структурированный индекс журналов, полученных из различных источников. Журналы могут быть запрошены с помощью команды `journalctl`.

Логирование в системный журнал

Чтобы направить вывод в системный журнал, используйте флаг `--log-driver`:

```
$ docker run --log-driver=syslog ubuntu echo 'outputting to syslog'
outputting to syslog
```

Вывод будет записан в файл системного журнала. Если у вас есть права доступа к файлу, можете просмотреть журналы, используя стандартные инструменты Unix:

```
$ grep 'outputting to syslog' /var/log/syslog
Jun 23 20:37:50 myhost docker/6239418882b6[2559]: outputting to syslog
```

Логирование в journald

Вывод в journald выглядит примерно так:

```
$ docker run --log-driver=journald ubuntu echo 'outputting to journald'
outputting to journald
$ journalctl | grep 'outputting to journald'
Jun 23 11:49:23 myhost docker[2993]: outputting to journald
```

> **ВНИМАНИЕ!** Перед выполнением предыдущей команды убедитесь, что на вашем хосте запущен journald.

Применение этого аргумента ко всем контейнерам на вашем хосте может быть трудоемким, поэтому можно изменить демон Docker, чтобы он по умолчанию регистрировал эти поддерживаемые механизмы.

Измените демон /etc/default/docker, или /etc/sysconfig/docker, или любой другой конфигурационный файл Docker, настроенный вашим дистрибутивом, так, чтобы активировалась строка `DOCKER_OPTS = ""` и включала в себя флаг драйвера журнала. Например, если строка выглядит так:

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
```

поменяйте ее на:

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 --log-driver syslog"
```

> **СОВЕТ.** Смотрите Приложение В для получения подробной информации о том, как изменить конфигурацию демона Docker на вашем хосте.

Если вы перезапустите своего демона Docker, контейнеры должны вести запись в соответствующую службу.

ОБСУЖДЕНИЕ

Еще один распространенный вариант, который стоит упомянуть в этом контексте (но не рассматриваемый здесь), заключается в том, что вы можете использовать контейнеры для реализации инфраструктуры ведения журналов ELK (Elasticsearch, Logstash, Kibana).

ВНИМАНИЕ! Изменение этого параметра демона на что-либо еще кроме `json-file` или `journald` будет означать, что стандартная команда `docker logs` больше не станет работать по умолчанию. Пользователи этого демона Docker могут не оценить это изменение, особенно потому, что файл /var/log/syslog (используемый драйвером `syslog`), как правило, недоступен для обычных пользователей.

МЕТОД 103 Мониторинг контейнеров с помощью cAdvisor

Если у вас есть приличное количество контейнеров, работающих в рабочем окружении, вы захотите контролировать их использование ресурсов и производительность точно так же, как и при работе нескольких процессов на хосте.

Сфера мониторинга (как в целом, так и в отношении Docker) – это обширная область, где есть много кандидатов. Здесь мы выбрали cAdvisor, так как это популярный вариант.

После того как компания Google раскрыла его исходный код, он быстро завоевал популярность. Если вы уже используете традиционное средство мониторинга хоста, такое как Zabbix или Sysdig, то стоит посмотреть, предлагает ли оно уже необходимую вам функциональность, – многие инструменты добавляют поддержку контейнеров на момент написания этой главы.

ПРОБЛЕМА

Вы хотите отслеживать производительность ваших контейнеров.

РЕШЕНИЕ

Используйте cAdvisor в качестве средства мониторинга.

cAdvisor – это инструмент, разработанный компанией Google для мониторинга контейнеров. Его отрытый исходный код есть на GitHub на странице https://github.com/google/cadvisor.

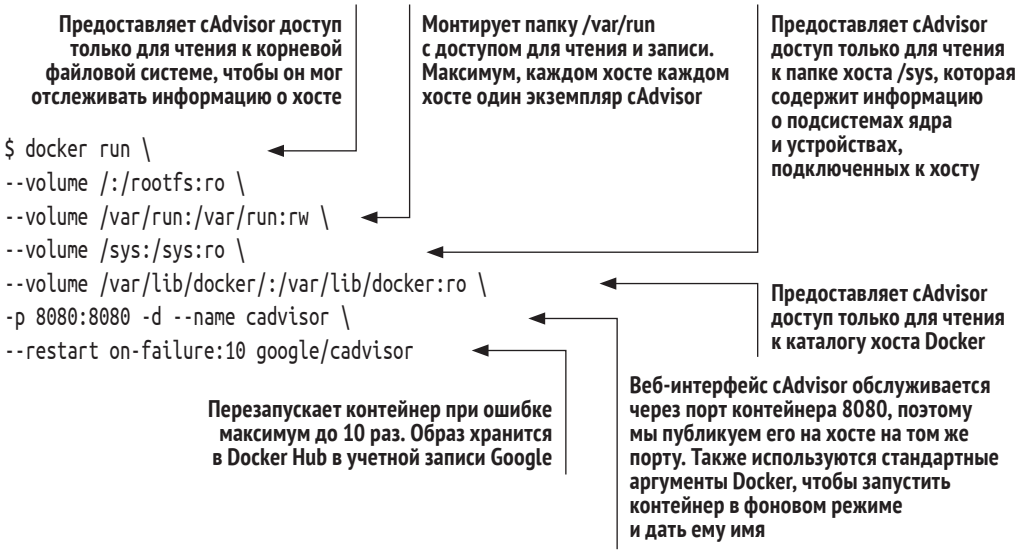
cAdvisor работает как демон, который собирает данные о производительности работающих контейнеров.

Среди прочего он отслеживает:

- параметры изоляции ресурсов;
- традиционное использование ресурсов;
- сетевую статистику.

cAdvisor можно установить непосредственно на хосте или запустить как контейнер Docker.

Листинг 15.3. Запуск cAdvisor



После того как вы запустили образ, можете перейти по адресу http://localhost:8080 в вашем браузере, чтобы приступить к изучению вывода данных. Есть информация о хосте, но нажав на ссылку Docker Containers в верхней части домашней страницы, вы сможете изучить графики процессора, памяти и другие традиционные данные. Просто нажмите на запущенные контейнеры, перечисленные под заголовком Subcontainers.

Данные собираются и сохраняются в памяти во время работы контейнера. На странице GitHub есть документация для сохранения данных в экземпляре InfluxDB.

GitHub-репозиторий также содержит подробную информацию о REST API и образце клиента, написанном на языке Go.

> **ПОДСКАЗКА.** InfluxDB – это база данных с открытым исходным кодом, предназначенная для обработки отслеживания временных рядов. Поэтому она идеально подходит для записи и анализа информации мониторинга, которая предоставляется в режиме реального времени.

ОБСУЖДЕНИЕ

Мониторинг – это преуспевающая область, и cAdvisor в настоящее время – это лишь один из множества компонентов. Например, Prometheus, быстро развивающийся стандарт для Docker, может получать и хранить данные, созданные cAdvisor, вместо того чтобы помещать их непосредственно в InfluxDB.

Мониторинг также является темой, которой разработчики могут увлечься. Это полезно для разработки стратегии мониторинга, достаточно гибкой, чтобы соответствовать меняющейся моде.

15.2. Управление ресурсами

Одной из основных проблем, связанных с запуском служб в рабочем окружении, является справедливое и функциональное распределение ресурсов. Внутри Docker применяет концепцию основной операционной системы cgroups для управления использованием ресурсов контейнеров. По умолчанию используется простой и равноправный алгоритм, когда контейнеры борются за ресурсы, но иногда этого недостаточно. Возможно, вы захотите зарезервировать или ограничить ресурсы для контейнера или класса контейнеров для эксплуатационных или служебных целей.

В этом разделе вы узнаете, как настроить использование процессора и памяти в контейнерах.

МЕТОД 104 Ограничение количества ядер для работы контейнеров

По умолчанию Docker позволяет запускать контейнеры в независимости от ядер, имеющихся на вашем компьютере. Контейнеры с одним процессом и потоком, очевидно, смогут максимально использовать только одно ядро, но многопоточные программы в контейнере (или несколько однопоточных) смогут использовать все ядра вашего процессора. Возможно, вы захотите изменить это поведение, если есть контейнер, который более важен, чем другие. Бороться за процессор каждый раз, когда запускаются ваши внутренние ежедневные отчеты – не лучший вариант для приложений, ориентированных на клиента. Также можете использовать этот метод, чтобы не позволить сбежавшим контейнерам запретить вам подключаться к серверу по протоколу SSH.

ПРОБЛЕМА

Вы хотите, чтобы у контейнера было минимальное распределение ЦП и жесткое ограничение на его потребление, в противном случае нужно ограничить количество ядер для работы контейнера.

РЕШЕНИЕ

Используйте опцию `--cpuset-cpus`, чтобы зарезервировать ядра ЦП для вашего контейнера.

Для правильного изучения этой опции вам нужно будет работать с этим методом на компьютере с несколькими ядрами. Эта схема может быть не рабочей, если вы используете облачный компьютер.

> **СОВЕТ.** В более старых версиях Docker использовался флаг `--cpuset`, который теперь устарел.

Если у вас не работает опция `--cpuset-cpus`, попробуйте использовать `--cpuset`.

Чтобы посмотреть на эффекты опции `--cpuset-cpus`, мы применим команду `htop`, которая дает полезное графическое представление об использовании ядра на вашем компьютере. Прежде чем продолжить, убедитесь, что она установлена – обычно она доступна в виде пакета htop в системном менеджере пакетов. Кроме того, вы можете установить ее в контейнере Ubuntu, запущенном с опцией `--pid = host` для предоставления информации о процессе от хоста контейнеру.

Если вы сейчас выполните команду `htop`, то, вероятно, увидите, что ни одно из ваших ядер не занято. Чтобы смоделировать некоторую нагрузку внутри пары контейнеров, выполните следующую команду в двух разных терминалах:

```
docker run ubuntu:14.04 sh -c 'cat /dev/zero >/dev/null'
```

Возвращаясь к `htop`, вы должны увидеть, что два ядра теперь работают на 100%. Чтобы ограничить это использование одним ядром, выполните команду `docker kill`, а затем эту команду в двух терминалах:

```
docker run --cpuset-cpus=0 ubuntu:14.04 sh -c 'cat /dev/zero >/dev/null'
```

Теперь `htop` покажет, что эти контейнеры используют только ваше первое ядро.

Опция `--cpuset-cpus` разрешает множественную спецификацию ядра в виде списка с запятыми (0,1,2), диапазона (0-2) или их комбинации (0-1,3). Поэтому резервирование процессора для хоста – это вопрос выбора диапазона для ваших контейнеров, который исключает ядро.

ОБСУЖДЕНИЕ

Вы можете использовать эту функцию различными способами. Например, можно зарезервировать определенные процессоры для хост-процессов, последовательно выделяя оставшиеся процессоры для работающих контейнеров. Или можно ограничить определенные контейнеры для запуска на их собственных выделенных процессорах, чтобы они не мешали вычислениям, используемым другими контейнерами.

В мультиарендной среде это может быть находкой, чтобы гарантировать, что рабочие нагрузки не мешают друг другу.

МЕТОД 105 Предоставление важным контейнерам больше ресурсов ЦП

Контейнеры на хосте, как правило, будут совместно использовать загрузку ЦП в равной степени, когда они конкурируют за него. Вы видели, как предоставлять абсолютные гарантии или ограничения, но они могут быть не совсем гибкими. Если вы хотите, чтобы процесс мог использовать больше ресурсов процессора по сравнению с другими, то постоянно резервировать для него целое ядро – пустая трата времени, и это может быть неудобно, если у вас небольшое количество ядер.

Docker облегчает мультиарендность для пользователей, которые хотят перенести свои приложения на общий сервер. Это может привести к проблеме *шумного соседа*, хорошо известной тем, кто имеет опыт работы с виртуальными машинами, когда один пользователь поглощает ресурсы и влияет на виртуальную машину другого пользователя, которая работает на том же оборудовании.

В качестве конкретного примера при написании книги нам пришлось использовать эту функциональность, чтобы уменьшить использование ресурсов особенно голодным приложением Postgres, которое потребляло циклы ЦП, лишая веб-сервер на машине способности обслуживать конечных пользователей.

ПРОБЛЕМА

Вы хотите иметь возможность предоставлять более важным контейнерам большую долю ресурсов ЦП или помечать некоторые контейнеры как менее важные.

РЕШЕНИЕ

Используйте аргумент `-c/-cpu-shares` в команде `docker run`, чтобы определить относительную долю использования ЦП.

Когда контейнер запускается, ему присваивается количество (по умолчанию 1024) *общих ресурсов процессора*.

Когда запущен только один процесс, он будет иметь доступ к 100 % ресурсов ЦП, если это необходимо, независимо от того, к каким ресурсам у него есть доступ. Это число используется, только когда вы конкурируете с другими контейнерами за процессор.

Представьте, что у нас есть три контейнера (A, B и C), и они пытаются использовать все доступные ресурсы ЦП:

- если им всем были предоставлены равные доли ресурсов ЦП, каждому из них будет выделена треть ЦП;
- если A и B даны 512, а C – 1024, C получит половину ЦП, а A и B – четверть;
- если A дается 10, B – 100, а C – 1000, A получит менее 1 % доступных ресурсов ЦП и сможет делать все, что требует ресурсов, только если B и C простаивают.

Все это предполагает, что ваши контейнеры могут использовать все ядра на компьютере (или что у вас есть только одно ядро). Docker распределяет нагрузку от контейнеров на все ядра, где это возможно. Если у вас два контейнера, на которых запущены однопоточные приложения на двухъядерном компьютере, очевидно, что нет способа применить относительное взвешивание при максимальном использовании доступных ресурсов. Каждому контейнеру будет предоставлено ядро для работы независимо от его веса.

Если вы хотите опробовать это, выполните следующие команды:

Листинг 15.4. Голодающая оболочка процессора

```
docker run --cpuset-cpus=0 -c 10000 ubuntu:14.04 \
sh -c 'cat /dev/zero > /dev/null' &
docker run --cpuset-cpus=0 -c 1 -it ubuntu:14.04 bash
```

Теперь вы видите, в приглашении bash все медлительно. Обратите внимание, что эти числа относительны – вы можете умножить их все на 10 (например), и они будут означать одно и то же. Но по умолчанию предоставляется 1024, поэтому, как только вы начнете менять эти цифры, стоит рассмотреть, что произойдет с процессами, которые запускаются без общей доли ЦП, указанной в команде, и работают с тем же набором ЦП.

> **СОВЕТ.** Поиск подходящего уровня общего ресурса ЦП для вашего случая использования – это искусство. Стоит посмотреть на вывод таких программ, как top и vmstat, чтобы определить, что использует процессорное время. При работе с top особенно полезно нажать клавишу **1**, чтобы отобразить действие каждого ядра процессора по отдельности.

ОБСУЖДЕНИЕ

Хотя мы не видели, чтобы этот метод непосредственно очень часто использовался в реальном мире, и его применение обычно наблюдается на базовой платформе, неплохо понять и поработать с базовым механизмом, чтобы знать, как он работает, когда арендаторы жалуются на отсутствие доступа (или явное отсутствие доступа) к ресурсам. Такое часто бывает в реальных условиях, особенно если рабочие нагрузки арендаторов чувствительны к колебаниям доступности инфраструктуры.

МЕТОД 106 Ограничение использования памяти контейнера

Когда вы запускаете контейнер, Docker позволяет ему выделять как можно больше памяти от хоста. Обычно это желательно (и это большое преимущество по сравнению с виртуальными машинами, у которых негибкий способ выделения памяти). Но иногда приложения могут выходить из-под контроля, выделять слишком много памяти и в итоге привести к остановке компьютера, когда он начнет подкачку. Это раздражает, и такое случалось с нами много раз в прошлом. Мы хотим ограничить потребление памяти контейнером во избежание подобного.

ПРОБЛЕМА

Вы хотите иметь возможность ограничить потребление памяти контейнером.

РЕШЕНИЕ

Используйте параметр `-m/--memory` для команды `docker run`.

Если вы работаете в Ubuntu, скорее всего, у вас не включена функция ограничения памяти по умолчанию. Чтобы проверить это, выполните команду `docker info`. Если одна из строк в выводе содержит предупреждение `No swap limit support`, к сожалению, вам необходимо выполнить некоторые настройки. Имейте в виду, что внесение этих изменений может повлиять на производительность вашего компьютера для всех приложений, – за дополнительной информацией обратитесь к документации по установке Ubuntu (https://docs.docker.com/install/linux/docker-ce/ubuntu/).

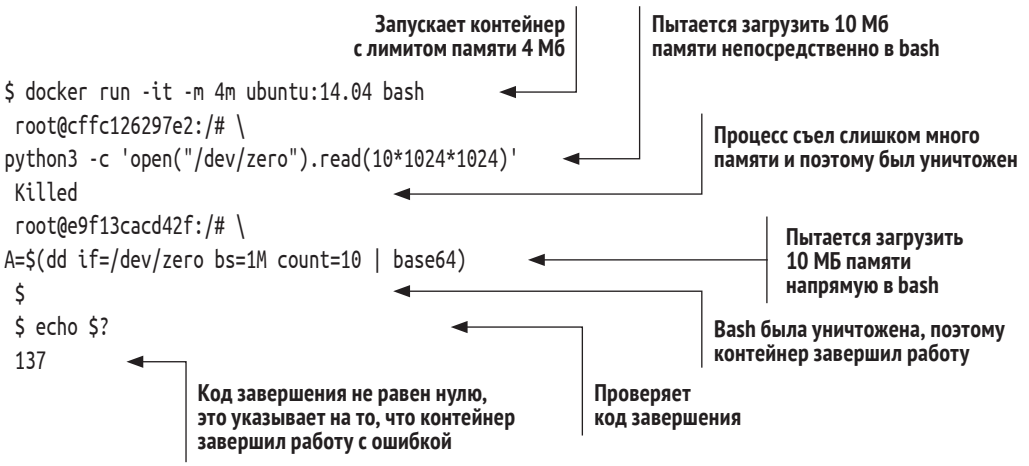
Говоря кратко, вам нужно указать ядру при загрузке, что вы хотите, чтобы эти ограничения были доступны. Для этого нужно изменить /etc/default/grub следующим образом. Если в `GRUB_CMDLINE _LINUX` уже есть значения, добавьте новые в конце:

```
-GRUB_CMDLINE_LINUX=""
+GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

Теперь вам нужно выполнить команду `sudo update-grub` и перезагрузить компьютер. При выполнении команды `docker run` больше не должно выскакивать предупреждение, и вы теперь готовы приступить к главному.

Во-первых, давайте вкратце продемонстрируем, что ограничение памяти работает, используя ограничение в 4 Мб, минимально возможное.

Листинг 15.5. Установка минимально возможного предела памяти для контейнера



При такого рода ограничениях есть ловушка. Чтобы продемонстрировать ее, будем использовать образ jess/stress, который содержит `stress`, инструмент, предназначенный для тестирования пределов системы.

> **СОВЕТ.** Jess/stress – полезный образ для тестирования любых ограничений ресурсов, которые вы накладываете на свой контейнер. Опробуйте предыдущие методы, используя этот образ, если хотите еще поэкспериментировать.

Если вы выполните следующую команду, то можете удивиться, увидев, что она не завершается сразу же:

```
docker run -m 100m jess/stress --vm 1 --vm-bytes 150M --vm-hang 0
```

Вы попросили Docker ограничить контейнер до 100 Мб и указали, что `stress` должен занимать 150 Мб. Можете проверить, что `stress` работает, как и ожидалось, выполнив эту команду:

```
docker top <container_id> -eo pid,size,args
```

Размер столбца в килобайтах показывает, что ваш контейнер действительно занимает около 150 Мб памяти, что ставит вопрос о том, почему он не был уничтожен. Оказывается, Docker дважды резервирует память – половину для физической памяти и половину для обмена. Если вы попробуете следующую команду, контейнер немедленно завершит работу:

```
docker run -m 100m jess/stress --vm 1 --vm-bytes 250M --vm-hang 0
```

Двойное резервирование – это просто значение по умолчанию, и им можно управлять с помощью аргумента `--memory-swap`, который определяет общий размер виртуальной памяти (memory + swap). Например, чтобы полностью исключить использование подкачки, вы должны установить для `--memory` и `--memory-swap` один и тот же размер. Можете найти больше примеров в справочнике по запуску Docker на странице https://docs.docker.com/engine/reference/run/#user-memory-constraints.

ОБСУЖДЕНИЕ

Ограничения памяти – одна из самых горячих тем для любой команды (или DevOps), работающей на платформе Docker. Неправильно сконфигурированные или плохо сконфигурированные контейнеры постоянно исчерпывают выделенную (или зарезервированную) память (я обращаюсь к *вам*, разработчики на Java!), требуя написания списка часто задаваемых вопросов и перечней задач, чтобы направлять пользователей, когда они возмущаются.

Знание того, что здесь происходит, очень помогает поддерживать такие платформы и дает пользователям контекст происходящего.

15.3. Варианты использования Docker для системного администратора

В этом разделе мы рассмотрим некоторые удивительные возможности использования Docker. На первый взгляд это может показаться странным,

но с помощью Docker можно упростить управление заданиями cron и использовать его в качестве инструмента резервного копирования.

> **ПОДСКАЗКА.** Задание cron – это регулярная команда, которая запускается демоном, включенным в качестве службы практически во все системы Linux. Каждый пользователь может указать свой собственный график команд для запуска. Оно активно используется системными администраторами для выполнения периодических задач, таких как очистка файлов журнала или запуск резервного копирования.

Это ни в коем случае не исчерпывающий список потенциальных применений, но он должен дать вам представление о гибкости Docker и некотором понимании того, как можно использовать его функции неожиданным образом.

МЕТОД 107 Использование Docker для запуска заданий cron

Если вам когда-либо приходилось управлять заданиями cron на нескольких хостах, вы, возможно, сталкивались с проблемами, связанными с необходимостью развертывания одного и того же программного обеспечения в нескольких местах, гарантируя что сам crontab правильно вызвал программу, которую вы хотите запустить.

Хотя существуют другие решения этой проблемы (например, использование Chef, Puppet, Ansible или какого-либо другого инструмента управления конфигурацией для управления развертыванием программного обеспечения на хостах), в качестве одного из вариантов можно использовать реестр Docker для хранения правильного вызова.

Это не всегда лучшее решение описанной проблемы, но это яркая иллюстрация преимуществ наличия изолированного и портативного хранилища конфигураций времени выполнения ваших приложений, а также это бесплатно, если вы уже используете Docker.

ПРОБЛЕМА

Вы хотите, чтобы ваши задания cron централизованно управлялись и автоматически обновлялись.

РЕШЕНИЕ

Извлеките и запустите сценарии задания cron в качестве контейнеров Docker.

Если у вас большое количество компьютеров, которым необходимо регулярно выполнять задания, вы обычно будете использовать crontab'ы и настраивать их вручную (да, такое все еще бывает) или использовать инструмент управления конфигурацией, такой как Puppet или Chef. Обновление их рецептов гарантирует, что при следующем запуске контроллера управления конфигурацией компьютера изменения применяются к crontab, готовому к запуску после этого.

ПОДСКАЗКА. Файл *crontab* – это специальный файл, поддерживаемый пользователем, в котором указано время запуска сценариев. Обычно это задачи обслуживания, такие как сжатие и архивирование файлов журнала, но это могут быть и критически важные приложения, такие как расчетник по кредитной карте.

В этом методе мы покажем вам, как заменить эту схему образами Docker, доставленными из реестра с помощью «docker pull».

В обычной ситуации, показанной на рис. 15.2, автор обновляет инструмент управления конфигурацией, который затем доставляется на серверы при запуске агента. Между тем задания cron выполняются со старым и новым кодом, пока системы обновляются.

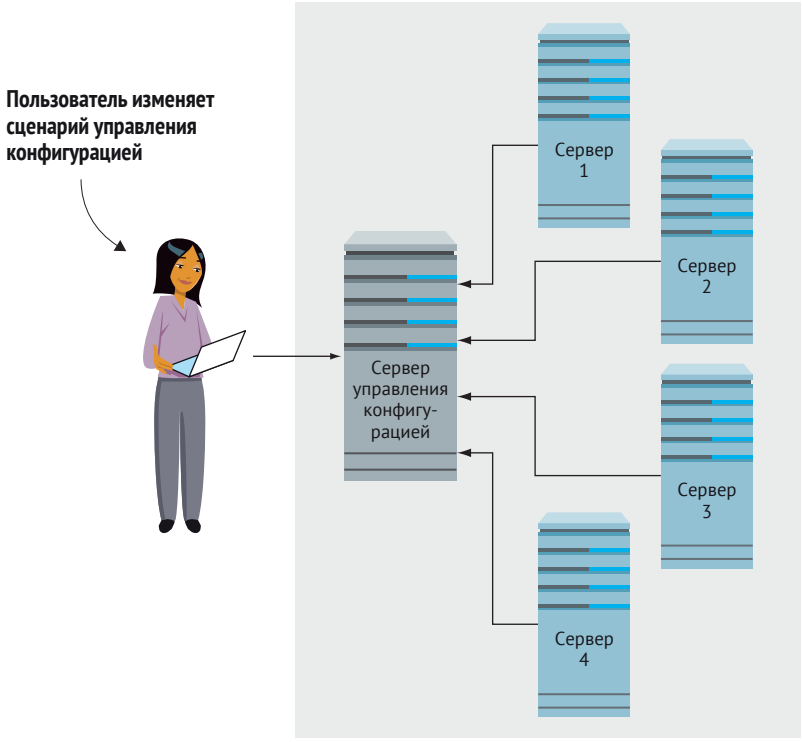


Рис. 15.2 ❖ Каждый сервер обновляет сценарии cron во время запланированного запуска агента управления конфигурацией

В сценарии Docker, показанном на рис. 15.3, серверы извлекают последнюю версию кода перед запуском заданий cron.

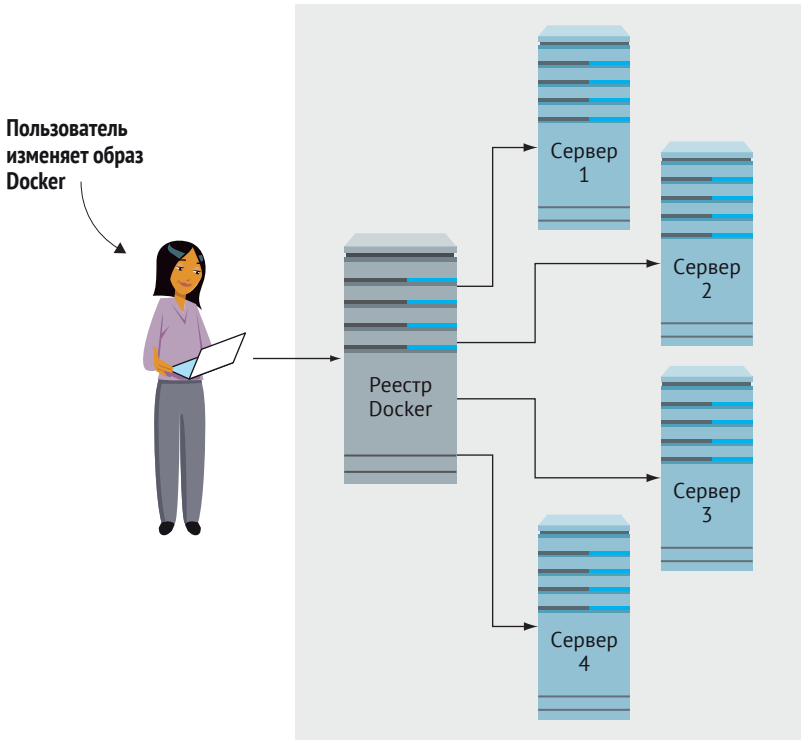


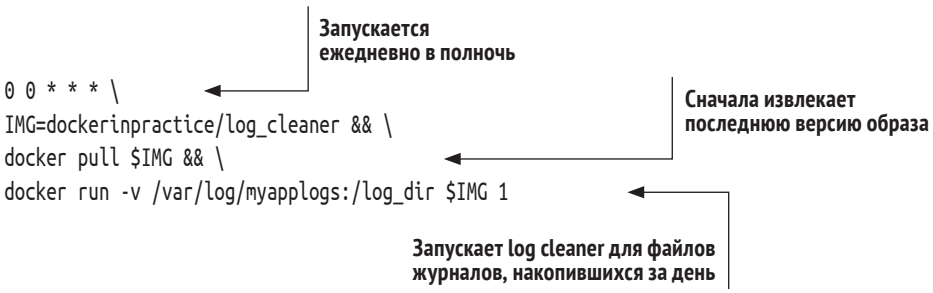
Рис. 15.3 ❖ Каждый сервер извлекает последний образ при каждом запуске задания cron

В этот момент вы можете задаться вопросом, зачем беспокоиться об этом, если у вас уже есть решение, которое работает. Вот некоторые преимущества использования Docker в качестве механизма доставки:

- каждый раз, когда задание запускается, оно обновляется до последней версии из центрального расположения;
- ваши файлы crontab становятся намного проще, потому что сценарий и код инкапсулированы в образ Docker;
- при более крупных или более сложных изменениях необходимо извлекать только дельты образа Docker, что ускоряет доставку и обновления;
- вам не нужно хранить код или двоичные файлы на самом компьютере;
- вы можете сочетать Docker с другими методами, такими как запись выходных данных в системный журнал, чтобы упростить и централизовать управление этими службами администрирования.

В этом примере мы будем использовать образ log_cleaner, который создали в методе 49. Вы, несомненно, помните, что этот образ инкапсулировал сценарий, который очищал файлы журналов на сервере и принимал параметр для количества дней очистки файлов журналов. Файл crontab, использующий Docker в качестве механизма доставки, будет выглядеть примерно так:

Листинг 15.6. Файл log cleaner



> **СОВЕТ.** Если вы не знакомы с cron, возможно, вы захотите узнать, что для редактирования crontab можно использовать `crontab -e`. В каждой строке указывается команда, которая должна быть запущена во время, указанное пятью пунктами в начале строки. Узнайте больше, заглянув на страницу справочника crontab.

Если происходит сбой, должен вступить в силу стандартный механизм отправки электронной почты cron. Если вы не полагаетесь на него, добавьте команду с оператором `or`. В следующем примере мы предполагаем, что ваша специальная команда оповещения – это `-my_alert_command`.

Листинг 15.7. Файл crontab с предупреждением об ошибке

```
0 0 * * * \
(IMG=dockerinpractice/log_cleaner && \
docker pull $IMG && \
docker run -v /var/log/myapplogs:/log_dir $IMG 1) \
|| my_alert_command 'log_cleaner failed'
```

Оператор `or` (в данном случае `||`) обеспечивает выполнение одной из команд с любой стороны. Если первая команда завершится неудачно (в этом случае любая из двух команд в скобках после cron-спецификации `0 0 * * *`, объединенной оператором and, `&&`), будет запущена вторая.

Оператор `||` гарантирует, что в случае сбоя какой-либо части задания очистки журнала запускается команда оповещения.

ОБСУЖДЕНИЕ

Нам очень нравится этот метод из-за его простоты и использования проверенных в действии технологий для оригинального решения проблемы.

Cron существует уже несколько десятилетий (с конца 1970-х годов, согласно Википедии), и его улучшение с помощью образа Docker – это метод, который мы используем дома для простого управления обычными заданиями.

МЕТОД 108 Подход «сохранить игру» по отношению к резервным копиям

Если вы когда-либо работали с транзакционной системой, вы будете знать, что, когда что-то идет не так, возможность анализа состояния системы на момент возникновения проблемы важна для анализа первопричины.

Обычно это делается с помощью комбинации средств:

- анализ журналов приложений;
- экспертиза базы данных (определение состояния данных в данный момент времени);
- анализ истории сборки (выяснение того, какой код и конфигурация выполнялись в службе в данный момент времени);
- анализ системы в реальном времени (например, кто-нибудь входил в систему и что-то менял?).

Для таких критически важных систем может потребоваться простой, но эффективный подход к резервному копированию сервисных контейнеров Docker. Хотя ваша база данных может быть отделена от инфраструктуры Docker, состояние конфигурации, кода и журналов можно хранить в реестре с помощью пары простых команд.

ПРОБЛЕМА

Вы хотите сохранить резервные копии контейнеров Docker.

РЕШЕНИЕ

Зафиксируйте контейнеры во время работы и разместите (push) полученный образ в качестве выделенного хранилища Docker.

Следуя рекомендациям Docker и используя некоторые функции, вы сможете избежать необходимости сохранять резервные копии контейнеров. Например, используя драйвер логирования, как описано в методе 102, вместо логирования в файловую систему контейнера означает, что журналы не нужно извлекать из резервных копий контейнера.

Но иногда реальность такова, что вы не можете делать все так, как вам хочется, и действительно нужно посмотреть, как выглядел контейнер. Приведенные ниже команды показывают весь процесс фиксации и размещения резервного контейнера.

Листинг 15.8. Фиксация и размещение резервного контейнера

Генерирует метку времени для гранулярности в секунду

Генерирует тег, который указывает на URL-адрес вашего реестра с тегом, который включает в себя имя хоста и дату

```
DATE=$(date +%Y%m%d_%H%M%S)
TAG="your_log_registry:5000/live_pmt_svr_backup:$(hostname -s)_${DATE}"
docker commit -m="$DATE" -a="Backup Admin" live_pmt_svr $TAG
docker push $TAG
```

Помещает контейнер в реестр

Фиксирует контейнер с датой в виде сообщения и «Администратором резервного копирования» в качестве автора

ВНИМАНИЕ! Этот метод приостановит контейнер, пока он работает, эффективно выводя его из эксплуатации. Ваша служба должна либо терпеть сбои, либо у вас должны быть запущены другие узлы, которые могут обслуживать запросы с балансировкой нагрузки.

Если выполнять это в шахматном порядке на всех хостах, у вас будет эффективная система резервного копирования и средство для восстановления состояния для инженеров службы поддержки с минимальной неопределенностью. На рис. 15.4 показан упрощенный вид такой установки.

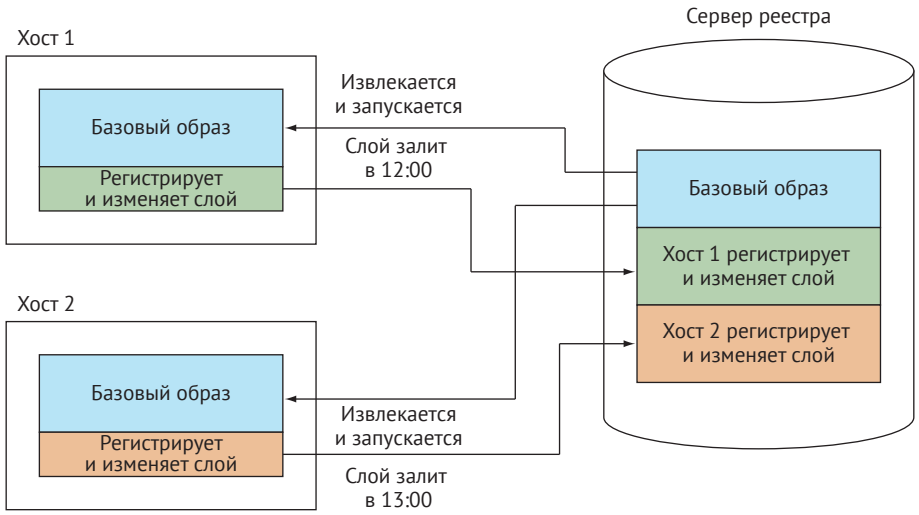


Рис. 15.4 ❖ Бэкап службы с двумя хостами

Резервные копии только увеличивают различия между базовым образом и состоянием контейнера во время его резервного копирования, а в шахматном порядке они расположены, чтобы гарантировать, что служба работает как минимум на одном хосте. Сервер реестра хранит только одну копию базового образа и различий в каждой точке фиксации, экономя место на диске.

ОБСУЖДЕНИЕ

Вы можете сделать шаг вперед, комбинируя этот метод с так называемой моделью «развертывания Феникса». Развертывание Феникса – это модель развертывания, в которой особое внимание уделяется замене как можно большей части системы, а не обновлению развертывания на месте. Это основной принцип многих инструментов Docker.

В этом случае вместо фиксации контейнера и продолжения его работы вы можете сделать следующее:

1. Извлеките свежую копию последнего образа из своего реестра.
2. Остановите работающий контейнер.
3. Запустите новый контейнер.
4. Зафиксируйте, пометьте и поместите старый контейнер в реестр.

Сочетание этих подходов дает вам еще большую уверенность в том, что действующая система не сместилась с исходного образа. Один из нас использует этот подход для управления действующей системой на домашнем сервере.

Резюме

- Вы можете направить логирование из ваших контейнеров в демон syslog вашего хоста.
- Выходные данные журнала Docker можно собирать в службу уровня хоста.
- cAdvisor можно использовать для контроля производительности ваших контейнеров.
- Использование контейнером процессора, ядра и памяти может быть ограничено и контролироваться.
- Docker имеет несколько неожиданных применений, таких как инструмент доставки cron и система резервного копирования.

Глава 16

Docker в рабочем окружении: решение проблем

О чем рассказывается в этой главе:

- обход пространства имен в Docker и непосредственное использование ресурсов хоста;
- убеждаемся, что ОС хоста не убивает процессы в контейнерах из-за нехватки памяти;
- непосредственная отладка сети контейнера с использованием инструментария хоста;
- отслеживание системных вызовов, чтобы определить, почему контейнер не работает на вашем хосте.

В этой главе мы обсудим, что можно сделать, когда абстракции Docker не работают. Эти темы непременно включают в себя знакомство с внутренним устройством Docker, чтобы понять, почему такие решения могут быть необходимы, и в процессе мы стремимся дать вам более глубокое понимание того, что может пойти не так и как это исправить.

16.1. Производительность: нельзя игнорировать хост

Хотя Docker стремится абстрагировать приложение от хоста, на котором оно запущено, нельзя полностью игнорировать хост. Чтобы предоставить свои абстракции, Docker должен добавить слои косвенности. Эти слои могут иметь последствия для вашей работающей системы, и в них иногда необходимо разбираться для решения проблем эксплуатации.

В данном разделе мы рассмотрим, как можно обойти некоторые из этих абстракций, заканчивая контейнером, в котором осталось немного от Docker. Мы также покажем, что, хотя Docker, по-видимому, абстрагируется от деталей хранилища, которое вы используете, иногда это может выйти вам боком.

МЕТОД 109 Получение доступа к ресурсам хоста из контейнера

В методе 34 мы рассматривали тома, наиболее часто используемый обход абстракции Docker. Они удобны для совместного использования файлов из хоста и хранения больших файлов вне слоев образов. Они также могут быть значительно быстрее для доступа к файловой системе, чем файловая система контейнера, так как некоторые хранилища накладывают значительные расходы на определенные рабочие нагрузки – это не полезно для всех приложений, но в некоторых случаях крайне важно.

В дополнение к расходам, налагаемым некоторыми хранилищами, происходит еще одно снижение производительности в результате сетевых интерфейсов, которые Docker настраивает, чтобы предоставить каждому контейнеру свою собственную сеть. Как и в случае с производительностью файловой системы, производительность сети определенно не является узким местом для всех, но это то, что вы можете захотеть оценить для себя (хотя тонкие детали настройки сети выходят далеко за рамки этой книги). В качестве альтернативы у вас могут быть другие причины для полного обхода сетевой среды Docker – сервер, открывающий случайные порты для прослушивания, может не очень хорошо обслуживаться при прослушивании диапазонов портов с помощью Docker, особенно потому, что, если вы откроете диапазон портов, они будут размещены на хосте независимо от того, используются они или нет.

Независимо от причины иногда абстракции Docker мешают, и Docker действительно предлагает возможность отказаться от них, если вам нужно.

ПРОБЛЕМА

Вы хотите разрешить доступ к ресурсам хоста из контейнера.

РЕШЕНИЕ

Используйте флаги, которые Docker предлагает для команды `docker run`, чтобы обойти пространства имен ядра, что использует Docker.

> **ПОДСКАЗКА.** Пространства имен ядра – это служба, которую ядро предлагает программам, позволяя им получать представления о глобальных ресурсах таким образом, что они, кажется, имеют свои отдельные экземпляры этого ресурса. Например, программа может запросить сетевое пространство имен, которое даст вам то, что кажется полным сетевым стеком. Docker использует и управляет этими пространствами имен для создания своих контейнеров.

В табл. 16.1 показано, как Docker использует пространства имен, и как можно эффективно отключить их.

Таблица 16.1 ❖ Мандаты Linux в контейнерах Docker

Пространство имен ядра	Описание	Используется в Docker?	Опция «Выключить»
Сеть	Подсистема сети	Да	`--net=host`
IPC	Межпроцессное взаимодействие: общая память, семафоры и т. д.	Да	`--ipc=host`
UTS	Имя хоста и домен NIS	Да	`--uts=host`
PID	Идентификаторы процессов	Да	`--pid=host`
Mount	Точки монтирования	Да	`--volume`, `--device`
User	Идентификаторы пользователей и групп	Нет	Отсутствует

ПРИМЕЧАНИЕ. Если какой-либо из этих флагов недоступен, скорее всего, это связано с тем, что ваша версия Docker устарела.

Например, если ваше приложение активно использует общую память, и вы хотите, чтобы ваши контейнеры использовали это пространство совместно с хостом, можете взять флаг `--ipc = host` для этого. Это относительно продвинутый вариант, поэтому мы сосредоточимся на других, более распространенных.

Сеть и имя хоста

Чтобы использовать сеть хоста, вы запускаете контейнер с флагом `--net`, установленным как `host`, например:

```
user@yourhostname:/$ docker run -ti --net=host ubuntu /bin/bash
root@yourhostname:/#
```

Вы заметите, что это сразу отличается от контейнера пространства имен сети тем, что имя хоста внутри контейнера совпадает с именем хоста. На практическом уровне это может вызвать путаницу, так как не очевидно, что вы находитесь в контейнере.

В изолированном от сети контейнере быстрая команда `netstat` покажет, что при запуске нет соединений:

```
host$ docker run -ti ubuntu
root@b1c4877a00cd:/# netstat
```

```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
root@b1c4877a00cd:/#
```

Аналогичный прогон с использованием сети хоста показывает обычную перегруженную сеть аналогично перегруженного технического писателя:

```
$ docker run -ti --net=host ubuntu
root@host:/# netstat -nap | head
Active Internet connections (servers and established)
Proto       Recv-Q   Send-Q    Local Address      Foreign Address    State        PID
➥ /Program name
tcp         0        0         127.0.0.1:47116    0.0.0.0:*          LISTEN       -
tcp         0        0         127.0.1.1:53       0.0.0.0:*          LISTEN       -
tcp         0        0         127.0.0.1:631      0.0.0.0:*          LISTEN       -
tcp         0        0         0.0.0.0:3000       0.0.0.0:*          LISTEN       -
tcp         0        0         127.0.0.1:54366    0.0.0.0:*          LISTEN       -
tcp         0        0         127.0.0.1:32888    127.0.0.1:47116    ESTABLISHED  -
tcp         0        0         127.0.0.1:32889    127.0.0.1:47116    ESTABLISHED  -
tcp         0        0         127.0.0.1:47116    127.0.0.1:32888    ESTABLISHED  -
root@host:/#
```

> **ПРИМЕЧАНИЕ.** `netstat` – это команда, которая позволяет просматривать информацию о сетевой среде в локальном сетевом стеке. Чаще всего она используется для определения состояния сетевых сокетов.

Флаг `net=host` применяют чаще всего по нескольким причинам. Во-первых, это может значительно облегчить подключение контейнеров. Но вы теряете преимущества проброса портов для своих контейнеров. Если есть два контейнера, которые прослушивают порт 80, например, вы не сможете запустить их на том же хосте таким образом. Вторая причина заключается в том, что при использовании этого флага производительность сети значительно улучшается по сравнению с Docker.

На рис. 16.1 на высоком уровне показаны уровни издержек, которые сетевой пакет должен проходить в Docker по сравнению с нативной сетью. В то время как нативная сеть должна проходить только через стек TCP/IP хоста к сетевой плате (NIC), Docker должен дополнительно поддерживать виртуальную пару Ethernet («veth-пара» – виртуальное представление физического соединения через кабель Ethernet), сетевой мост между этой veth-парой и сетью хоста и уровня преобразования сетевых адресов. Эти издержки могут привести к тому, что сеть Docker будет вдвое медленнее нативной сети хоста в обычных вариантах использования.

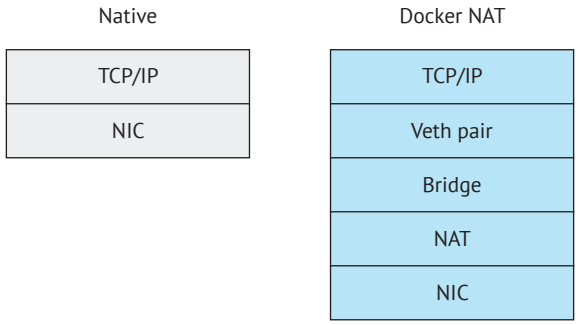


Рис. 16.1 ❖ Сетевая среда Docker в сравнении с нативной сетевой средой

PID

Флаг пространства имен PID похож на другие:

Выполняет команду ps в контейнеризированной среде, показывая только процесс с PID 1

ps, который мы запускаем, является единственным процессом в этом контейнере и имеет PID 1

```
imiell@host:/$ docker run ubuntu ps -p 1
  PID    TTY    TIME        CMD
  1      ?      00:00:00    ps
imiell@host:/$ docker run --pid=host ubuntu ps -p 1
  PID    TTY    TIME        CMD
  1      ?      00:00:27    systemd
```

На этот раз PID 1 – это команда systemd, которая является процессом запуска операционной системы хоста. В вашем случае она может выглядеть иначе в зависимости от вашего дистрибутива

Выполняет ту же команду ps с удаленным пространством имен PID, давая нам представление о процессах хоста

Предыдущий пример демонстрирует, что процесс хоста systemd имеет идентификатор процесса 1 в контейнере, у которого есть представление флагов PID, тогда как без этого представления единственным видимым процессом является сама команда `ps`.

Монтирование

Если вы хотите получить доступ к устройствам хоста, возьмите флаг `--device` для использования определенного устройства или смонтируйте файловую систему всего хоста с флагом `--volume`:

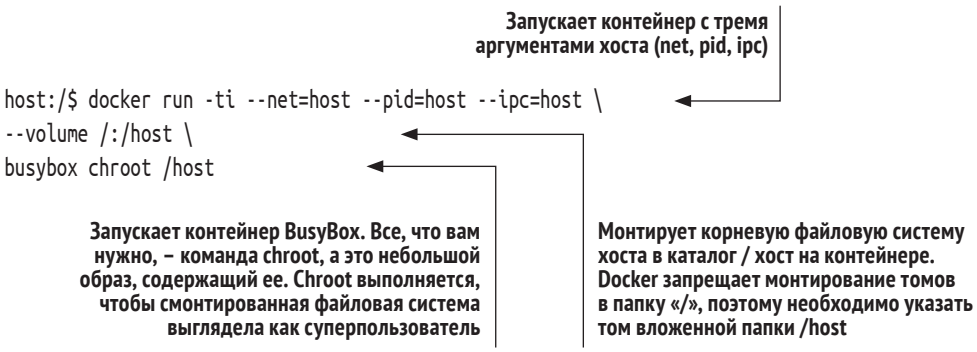
```
docker run -ti --volume /:/host ubuntu /bin/bash
```

Предыдущая команда монтирует каталог хоста / в каталог контейнера `/host`. Вам, наверное, интересно, почему нельзя смонтировать каталог хоста / в каталог контейнера /. Это явно запрещено командой `docker`.

Вам также может быть интересно, можно ли использовать эти флаги для создания контейнера, который практически неотличим от хоста. Это приводит нас к следующему разделу...

Контейнер, напоминающий хост

Вы можете использовать следующие флаги для создания контейнера, который имеет почти прозрачное представление хоста:



Ирония в том, что Docker был охарактеризован как «`chroot` на стероидах», и здесь мы используем нечто, характеризуемое как фреймворк для запуска `chroot` таким образом, что это подрывает одну из основных целей `chroot` – защиту файловой системы хоста. Обычно в этот момент мы стараемся не думать об этом слишком сильно.

В любом случае трудно представить реальное использование этой команды (поучительное, как здесь). Если вам что-то придет на ум, пожалуйста, напишите нам.

Тем не менее вы можете использовать это в качестве основы для более полезных команд, таких как эта:

```
$ docker run -ti --workdir /host \
  --volume /:/host:ro ubuntu /bin/bash
```

В этом примере `--workdir /host` устанавливает рабочий каталог при запуске контейнера как корень файловой системы хоста с аргументом `--volume`. Часть спецификации тома `:ro` означает, что файловая система хоста будет подключена только для чтения.

С помощью этой команды вы можете просматривать файловую систему в режиме «только для чтения», имея среду, в которой можно устанавливать инструментальные средства (с помощью стандартного менеджера пакетов Ubuntu), чтобы проверить это. Например, вы используете образ, который запускает отличный инструмент, сообщающий о проблемах безопасности в файловой системе хоста, и его не нужно устанавливать на вашем хосте.

> **ВНИМАНИЕ!** Как следует из предыдущего обсуждения, используя эти флаги, вы подвергаете себя большему количеству угроз безопасности. С точки зрения безопасности их использование следует рассматривать как то же самое, что запуск с флагом `--privileged`.

ОБСУЖДЕНИЕ

Из этого метода вы узнали, как обходить абстракции Docker внутри контейнера. Их отключение может привести к ускорению или другим удобствам, чтобы Docker лучше удовлетворял ваши потребности. Один из вариантов, который мы использовали в прошлом, – это установка сетевых утилит (возможно, таких как tcpflow, упомянутой в методе 112) внутри контейнера и предоставление сетевых интерфейсов хоста. Это позволяет вам экспериментировать с различными инструментами на временной основе без необходимости их установки.

В следующем методе рассматривается, как можно обойти ограничение основного дискового хранилища Docker.

МЕТОД 110 Отключение OOM killer

«OOM killer» звучит как плохой ужастик или серьезное заболевание, но на самом деле это поток в ядре операционной системы Linux, который решает, что делать, когда хосту не хватает памяти. После того, как операционная система исчерпала аппаратную память, израсходовала все доступное пространство подкачки и удалила все кешированные файлы из памяти, она вызывает OOM killer, чтобы решить, какие процессы должны быть уничтожены.

ПРОБЛЕМА

Вы хотите предотвратить уничтожение контейнеров OOM killer.

РЕШЕНИЕ

Используйте флаг `--oom-kill-disable` при запуске контейнера.

Решение этой проблемы так же просто, как добавление флага в ваш контейнер Docker. Но, как это часто бывает, полная история не так проста.

В следующем листинге показано, как отключить OOM killer для контейнера:

Листинг 16.1. --oom-kill-disable выводит предупреждение

```
$ docker run -ti --oom-kill-disable ubuntu sleep 1
 WARNING: Disabling the OOM killer on containers without setting a
➥ '-m/--memory' limit may be dangerous.
```

Флаг --oom-kill-disable добавляется в обычную команду docker run

Относительно другого флага, который может быть установлен, выводится предупреждение

Это важное предупреждение. Оно сообщает вам, что работать с этим параметром опасно, но не говорит почему. Опасно устанавливать эту опцию, потому что, если у вашего хоста не хватает памяти, операционная система уничтожит все другие пользовательские процессы раньше ваших.

Иногда это желательно, например, если у вас имеется критическая часть инфраструктуры, которую вы хотите защитить от сбоев – может быть, процесс аудита или ведения журнала, выполняемый на (или для) всех контейнерах на хосте. Даже тогда захочется дважды подумать о том, насколько это будет вредно для вашей среды. Например, контейнер может зависеть от другой работающей инфраструктуры на том же хосте. При работе на контейнерной платформе, такой как OpenShift, ваш контейнер выживет, даже если ключевые процессы платформы будут уничтожены. Скорее всего, вы захотите, чтобы эта ключевая инфраструктура работала до этого контейнера.

Листинг 16.2. --oom-kill-disable без предупреждения

```
$ docker run -ti --oom-kill-disable --memory 4M ubuntu sleep 1
 $
```

На этот раз предупреждения не видно

Флаг --memory добавляется в обычную команду docker run

> **ПРИМЕЧАНИЕ.** Минимальный объем памяти, который вы можете выделить, составляет 4M, где «M» означает мегабайт. Вы также можете использовать «G», что значит гигабайт.

Возможно, вам интересно, как определить, уничтожил ли OOM killer ваш контейнер. Это легко сделать с помощью команды `docker inspect`:

Листинг 16.3. Определяем, был ли ваш контейнер уничтожен OOM killer

```
$ docker inspect logger | grep OOMKilled
            "OOMKilled": false,
```

Эта команда выводит информацию о том, почему был уничтожен контейнер, в том числе сделал ли это OOM killer.

ОБСУЖДЕНИЕ

OOM killer не требует установки расширенных привилегий в контейнере и того, чтобы вы были пользователем root. Все, что вам нужно – это доступ к команде `docker`. Это еще одна причина опасаться предоставления непривилегированным пользователям доступа к команде `docker` (см. главу 14 о безопасности).

Это рискованно не только с точки зрения безопасности, но и с точки зрения стабильности. Если пользователь может выполнить команду `docker`, он запустит процесс, что приведет к постепенной утечки памяти (это распространенное явление во многих производственных средах).

Если для этой памяти не установлены границы, операционная система вступит в действие, как только ее параметры будут исчерпаны, и сначала

уничтожит пользовательский процесс с наибольшим использованием памяти (это упрощение алгоритма OOM-killer, который был проверен в действии и вырос годами). Однако, если контейнер был запущен, когда OOM killer отключен, это может уничтожить все контейнеры на хосте, вызывая гораздо большие разрушения и нестабильность для его пользователей.

Для более детального подхода к управлению памятью вы можете настроить «OOM Score» с помощью флага `--oom-score-adj`. Еще один подход, подходящий для ваших целей, – отключить чрезмерную загрузку памяти в ядре. Это приводит к глобальному отключению OOM-killer, поскольку память предоставляется только в том случае, если она определенно доступна.

Однако это может ограничить количество контейнеров, работающих на ваших хостах, что также может быть нежелательным.

Как всегда, управление производительностью – это искусство!

16.2. Когда контейнеры дают течь – отладка Docker

В этом разделе будут рассмотрены методы, которые помогут вам понять и устранить проблемы с приложениями, работающими в контейнерах Docker. Мы расскажем, как перейти в сеть контейнера, используя инструменты из вашего хоста для устранения проблем, и рассмотрим альтернативу, которая позволяет избежать манипулирования контейнером путем непосредственного мониторинга сетевых интерфейсов.

Наконец, покажем, как может сломаться абстракция Docker, приводя к тому, что контейнеры будут работать на одном хосте, а не на другом, и как отлаживать это в реальных системах.

МЕТОД 111 Отладка сети контейнера с помощью nsenter

В идеальном мире вы могли бы использовать socat (см. метод 4) в контейнере ambassador для диагностики проблем, связанных с обменом данными между контейнерами. Вы запустите дополнительный контейнер и убедитесь, что соединения идут к этому новому контейнеру, который работает как прокси-сервер. Прокси-сервер позволяет диагностировать и контролировать соединения, а затем перенаправляет их в нужное место. К сожалению, настраивать такой контейнер только для отладки не всегда удобно (или возможно).

> **СОВЕТ.** Смотрите метод 74, где приводится описание шаблона ambassador.

Вы уже читали о команде `docker exec` в методах 15 и 19. Здесь же обсуждается *nsenter*, инструмент, который выглядит аналогично, но позволяет использовать инструменты из вашего компьютера внутри контейнера, а не ограничиваться тем, что установил контейнер.

ПРОБЛЕМА

Вы хотите отладить сетевую проблему в контейнере, но инструменты находятся не в контейнере.

РЕШЕНИЕ

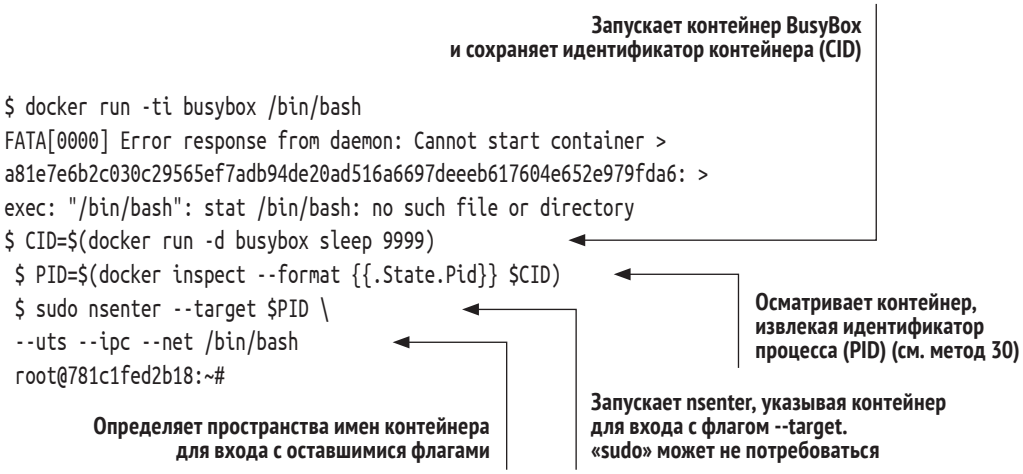
Используйте nsenter, чтобы перейти в сеть контейнера, но сохраните инструментарий своего хоста.

Если у вас еще нет nsenter на хосте Docker, можете создать его с помощью следующей команды:

```
$ docker run -v /usr/local/bin:/target jpetazzo/nsenter
```

nsenter будет установлен в /usr/local/bin, и вы сможете сразу же использовать его. nsenter также может быть доступен в вашем дистрибутиве (в пакете util-linux).

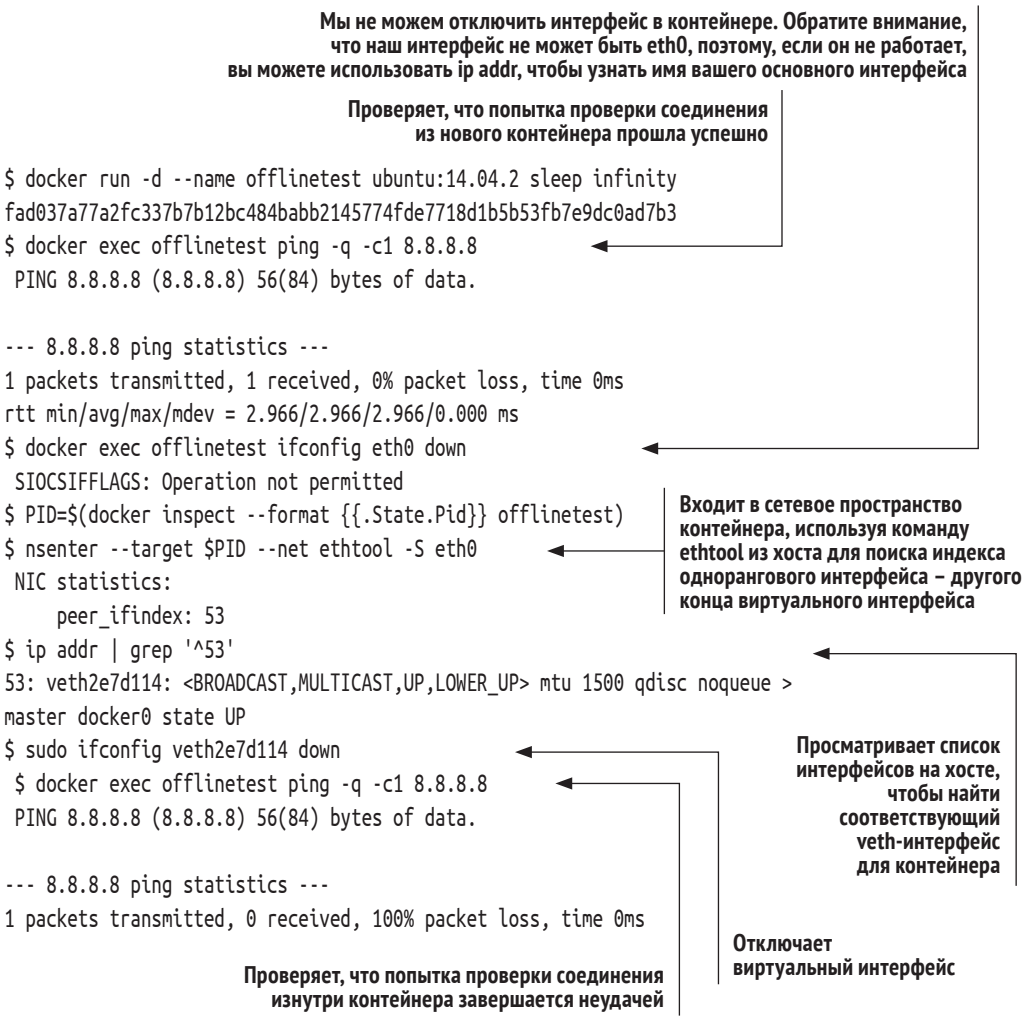
Вы, возможно, уже заметили, что, как правило, полезный образ BusyBox не поставляется с bash по умолчанию. В качестве демонстрации nsenter мы покажем, как можно войти в контейнер BusyBox с помощью программы bash вашего хоста:



Смотрите метод 109 для получения более подробной информации о пространствах имен, которые понимает nsenter. Критическим моментом в выборе пространств имен является то, что вы не применяете флаг `--mount`, который будет использовать файловую систему контейнера, поскольку bash не будет доступен. /bin /bash указан как исполняемый файл для запуска.

Следует отметить, что вы не получаете прямого доступа к файловой системе контейнера, но получаете все инструменты, которые есть у вашего хоста.

То, что нам было нужно раньше, – это способ выяснить, какое интерфейсное устройств veth на хосте какому контейнеру соответствует. Например, иногда желательно быстро выбить контейнер из сети. Непривилегированный контейнер не может отключить сетевой интерфейс, поэтому вам нужно сделать это из хоста, выяснив имя интерфейса veth.



Последний пример программы, которую вы можете использовать из контейнера, – это tcpdump, инструмент, записывающий все TCP-пакеты на сетевом интерфейсе. Чтобы использовать его, вам нужно запустить `nsenter` с командой `--net`, что позволит «видеть» сеть контейнера из хоста, а следовательно, отслеживать пакеты с помощью tcpdump.

Например, команда tcpdump в следующем коде записывает все пакеты в файл /tmp/google.tcpdump (мы предполагаем, что вы все еще находитесь в сеансе nsenter, который начали ранее). Затем при извлечении веб-страницы инициируется какой-то сетевой трафик:

```
root@781c1fed2b18:/# tcpdump -XXs 0 -w /tmp/google.tcpdump &
root@781c1fed2b18:/# wget google.com
--2015-08-0715:12:04-- http://google.com/
Resolving google.com (google.com)... 216.58.208.46, 2a00:1450:4009:80d::200e
```

```
Connecting to google.com (google.com)|216.58.208.46|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://www.google.co.uk/?gfe_rd=cr&ei=tLzEVcCXN7Lj8wepgarQAQ >
[following]
--2015-08-0715:12:04-- >
http://www.google.co.uk/?gfe_rd=cr&ei=tLzEVcCXN7Lj8wepgarQAQ
Resolving www.google.co.uk (www.google.co.uk)... 216.58.208.67, >
2a00:1450:4009:80a::2003
Connecting to www.google.co.uk (www.google.co.uk)|216.58.208.67|:80... >
connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: 'index.html'

index.html              [ <=>            ] 18.28K   --.-KB/s     in 0.008s

2015-08-0715:12:05 (2.18 MB/s) - 'index.html' saved [18720]

root@781c1fed2b18:# 15:12:04.839152 IP 172.17.0.26.52092 > >
google-public-dns-a.google.com.domain: 7950+ A? google.com. (28)
15:12:04.844754 IP 172.17.0.26.52092 > >
google-public-dns-a.google.com.domain: 18121+ AAAA? google.com. (28)
15:12:04.860430 IP google-public-dns-a.google.com.domain > >
172.17.0.26.52092: 79501/0/0 A 216.58.208.46 (44)
15:12:04.869571 IP google-public-dns-a.google.com.domain > >
172.17.0.26.52092: 181211/0/0 AAAA 2a00:1450:4009:80d::200e (56)
15:12:04.870246 IP 172.17.0.26.47834 > lhr08s07-in-f14.1e100.net.http: >
Flags [S], seq 2242275586, win 29200, options [mss 1460,sackOK,TS val >
49337583 ecr 0,nop,wscale 7], length 0
```

В зависимости от настроек вашей сети вам может понадобиться временно изменить файл resolv.conf, чтобы разрешить поиск DNS. Если вы получаете сообщение об ошибке «Temporary failure in name resolution», попробуйте добавить строку `nameserver 8.8.8.8` в начало файла /etc/resolv.conf. Не забудьте вернуть его, когда закончите.

ОБСУЖДЕНИЕ

Этот метод дает вам возможность быстро изменить сетевое поведение контейнеров, не прибегая к каким-либо инструментам из главы 10 (методы 78 и 79) для моделирования обрыва сети.

Вы также видели убедительный пример использования Docker – гораздо проще отладить сетевые проблемы в изолированной сетевой среде, которую предоставляет Docker, чем делать это в неконтролируемой среде. Попытка запомнить правильные аргументы для tcpdump для надлежащей фильтрации нерелевантных пакетов посреди ночи – процесс, подверженный ошибкам.

Используя nsenter, можете забыть об этом и собирать все данные внутри контейнера без необходимости использования tcpdump (или необходимости устанавливать его) на образе.

МЕТОД 112 Использование tcpflow для отладки в полете без перенастройки

tcpdump – стандарт де-факто в исследовании сети и, вероятно, первый инструмент, к которому обращается большинство людей, если их просят заняться отладкой проблемы в сети.

Но tcpdump обычно используется для отображения сводок пакетов и изучения заголовков пакетов и информации о протоколах – он не настолько полнофункционален, чтобы отображать поток данных на уровне приложения между двумя программами. Это может быть очень важно при расследовании проблем, связанных с обменом данными между двумя приложениями.

ПРОБЛЕМА

Вам необходимо отслеживать коммуникационные данные контейнеризированного приложения.

РЕШЕНИЕ

Используйте tcpflow для сбора трафика, пересекающего интерфейс.

tcpflow похож на tcpdump (принимает те же выражения сопоставления с образцом), но предназначен, для того чтобы дать вам лучшее понимание потоков данных приложения. tcpflow может быть доступен в вашем системном менеджере пакетов, но, если такой возможности нет, мы подготовили образ Docker, который вы можете использовать, и который по функциональности должен быть практически идентичен эквивалентной установке менеджера пакетов:

```
$ IMG=dockerinpractice/tcpflow
$ docker pull $IMG
$ alias tcpflow="docker run --rm --net host $IMG"
```

Можно использовать tcpflow с Docker двумя способами: навести его на интерфейс `docker0` и применить выражение фильтрации пакетов, чтобы получить только те пакеты, которые вы хотите, или использовать прием из предыдущего метода, чтобы найти veth-интерфейс контейнера, который вас интересует, и выполнить сбор.

> **СОВЕТ.** Вы можете обратиться к рис. 10.2 в главе 10, чтобы освежить память о том, как сетевой трафик течет внутри Docker, и понять, почему при использовании `docker0` будет собран контейнерный трафик.

Фильтрация выражений – это мощная функция tcpflow, которую можно использовать после подключения к интерфейсу. Она позволяет детализировать интересующий вас трафик. Для начала покажем простой пример:

```
$ docker run -d --name tcpflowtest alpine:3.2 sleep 30d
fa95f9763ab56e24b3a8f0d9f86204704b770ffb0fd55d4fd37c59dc1601ed11
$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' tcpflowtest
172.17.0.1
$ tcpflow -c -J -i docker0 'host 172.17.0.1 and port 80'
tcpflow: listening on docker0
```

В предыдущем примере вы просите tcpflow вывести подсвеченный поток любого трафика, поступающего в ваш контейнер или из него, с портом источника или назначения 80 (обычно используется для HTTP-трафика). Теперь можете опробовать его, получив веб-страницу в контейнере в новом терминале:

```
$ docker exec tcpflowtest wget -O /dev/null http://www.example.com/
Connecting to www.example.com (93.184.216.34:80)
null            100% |*******************************|  127    00:00:00 ETA
```

Вы увидите подсвеченный вывод в терминале tcpflow. Совокупный вывод команды пока будет выглядеть примерно так:

```
$ tcpflow -J -c -i docker0 'host 172.17.0.1 and (src or dst port 80)'
tcpflow: listening on docker0
172.017.000.001.36042-093.184.216.034.00080: >
GET / HTTP/1.1                          <-- Blue coloring starts
Host: www.example.com
User-Agent: Wget
Connection: close

093.184.216.034.00080-172.017.000.001.36042: >
HTTP/1.0 200 OK                         <-- Red coloring starts
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Date: Mon, 17 Aug 2015 12:22:21 GMT
[...]

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
[...]
```

ОБСУЖДЕНИЕ

tcpflow – отличное дополнение к вашему инструментарию, учитывая, насколько он ненавязчив. Можете запустить его на долго работающих контейнерах, чтобы получить представление о том, что они передают прямо сейчас, или использовать его вместе с tcpdump (как в предыдущем методе), чтобы получить более полную картину того, какие запросы делает ваше приложение и какая передается информация.

Как и tcpdump, предыдущий метод также охватывает использование nsenter для мониторинга трафика только на одном контейнере (что и будет делать `docker0`).

МЕТОД 113 Отладка контейнеров, которые не работают на определенных хостах

Два предыдущих метода показали, как можно приступить к изучению проблем, вызванных взаимодействием между вашими контейнерами и другими локациями (будь то в большей степени контейнеры или третьи лица в интернете).

Если вы изолировали проблему для одного хоста и уверены, что причина кроется не во внешнем взаимодействии, следующим шагом должно стать уменьшение количества движущихся частей (удаление томов и портов) и проверка деталей самого хоста (свободное место на диске, количество дескрипторов открытых файлов и т. д.). Вероятно, также стоит проверить, что на каждом хосте установлена последняя версия Docker.

В некоторых случаях ничего из вышеперечисленного не поможет – у вас есть образ, который можно запустить без аргументов (например, `docker run imagename`); он должен быть в идеальном состоянии, но работает по-разному на разных хостах.

ПРОБЛЕМА

Вы хотите определить, почему конкретное действие в контейнере не работает на конкретном хосте.

РЕШЕНИЕ

Сконцентрируйтесь на процессе, чтобы увидеть, какие системные вызовы он совершает, и сравните это с работающей системой.

Хотя заявленная цель Docker состоит в том, чтобы позволить пользователям «запускать любое приложение где угодно», средства, с помощью которых он пытается достичь этого, не всегда можно считать надежными.

Docker рассматривает API ядра Linux как его *хост* (среду, в которой он может работать). Когда впервые узнают, как работает Docker, многие спрашивают, как он обрабатывает изменения в API Linux. Насколько мы знаем, он этого еще не делает. К счастью, API-интерфейс Linux обратно совместим, но нетрудно представить себе сценарий в будущем, когда создается новый API-вызов интерфейса Linux и используется Docker-изированным приложением, а затем это приложение развертывается в ядре, достаточно новом для запуска Docker, но достаточно старом, чтобы не поддерживать этот вызов.

> **ПРИМЕЧАНИЕ.** Вы можете подумать, что изменение API ядра Linux – это какая-то теоретическая проблема, но мы столкнулись с этим сценарием при написании первого издания этой книги. Проект, над которым мы работали, использовал системный вызов Linux memfd_create, существующий только в ядрах версии 3.17 и выше.

Поскольку некоторые хосты имели более старые ядра, наши контейнеры не работали на некоторых системах и работали на других.

Этот сценарий не единственный способ, при котором абстракция Docker может потерпеть неудачу. Контейнеры могут давать сбой в определенных ядрах, потому что приложение строит предположения относительно файлов на хосте. Хотя и редко, но такое случается, и важно быть готовым к этому.

SELinux и контейнеры: взаимное влияние

Примером того, где абстракция Docker может не работать, – когда что-то взаимодействует с SELinux. Как обсуждалось в главе 14, SELinux – это уровень безопасности, реализованный в ядре, который работает вне прав доступа обычного пользователя.

Docker использует этот уровень, чтобы повысить безопасность контейнера, управляя действиями, которые можно выполнять внутри контейнера. Например, если вы пользователь root в контейнере, вы являетесь тем же пользователем, что и root на хосте. Несмотря на то что трудно вырваться из контейнера, чтобы получить root на хосте, это не является невозможным; были обнаружены эксплойты, и могут существовать другие, о которых сообщество не знает.

SELinux обеспечивает еще один уровень защиты, так что даже если пользователь root выходит из контейнера на хост, существуют ограничения на то, какие действия он может выполнять на хосте.

Пока все хорошо, но проблема для Docker заключается в том, что SELinux реализован на хосте, а не в контейнере. Это означает, что программы, работающие в контейнерах, которые запрашивают состояние SELinux и обнаруживают, что он включен, могут строить определенные предположения относительно среды, в которой они работают, и неожиданно завершиться неудачно, если эти ожидания не будут выполнены.

В следующем примере мы запускаем машину с CentOS 7 и Vagrant с установленным Docker, а внутри нее – контейнер Ubuntu 12.04. При выполнении довольно простой команды, чтобы добавить пользователя, код завершения будет равен 12, что указывает на ошибку. И действительно, пользователь не был создан:

```
[root@centos vagrant]# docker run -ti ubuntu:12.04
Unable to find image 'ubuntu:12.04' locally
Pulling repository ubuntu
78cef618c77e: Download complete
b5da78899d3a: Download complete
87183ecb6716: Download complete
```

```
82ed8e312318: Download complete
root@afade8b94d32:/# useradd -m -d /home/dockerinpractice dockerinpractice
root@afade8b94d32:/# echo $?
12
```

Та же команда, выполненная в контейнере ubuntu:14.04, работает просто отлично. Если хотите попробовать воспроизвести этот результат, вам понадобится машина с CentOS 7 (или аналогичная). Но для образовательных целей достаточно будет следовать остальной части метода, используя любую команду и контейнер.

> **ПОДСКАЗКА.** В bash, `$?` дает код завершения команды последнего запуска. Значение кода завершения варьируется от команды к команде, но обычно код 0 означает, что вызов был успешным, а ненулевой код указывает на ошибку или какое-либо исключительное состояние.

Отладка API-вызовов Linux

Поскольку мы знаем, что вероятное различие между контейнерами связано с различиями между API-интерфейсами ядра, работающими на хостах, strace может помочь вам определить различия между вызовами API-интерфейса ядра.

strace – это инструмент, который позволяет отслеживать вызовы API Linux, совершаемые процессом (так называемые системные вызовы). Это чрезвычайно полезный инструмент для отладки и обучения.

Можно увидеть, как он работает, на рис. 16.2.

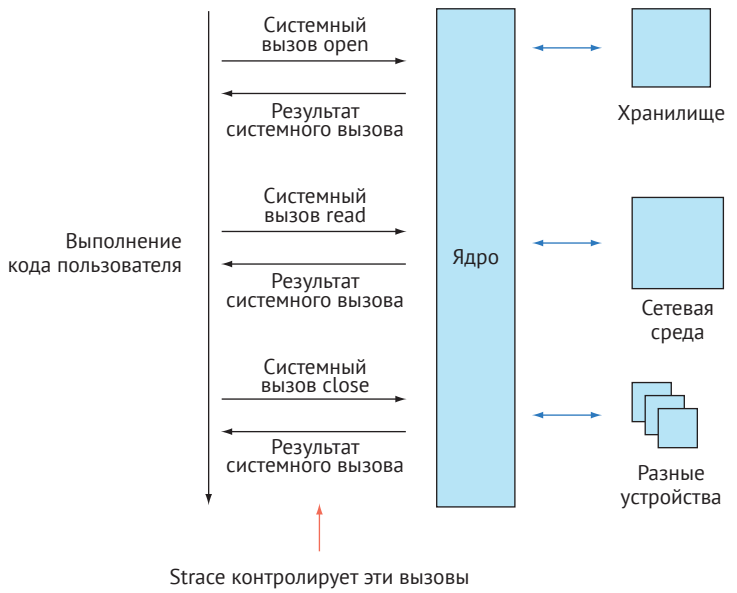
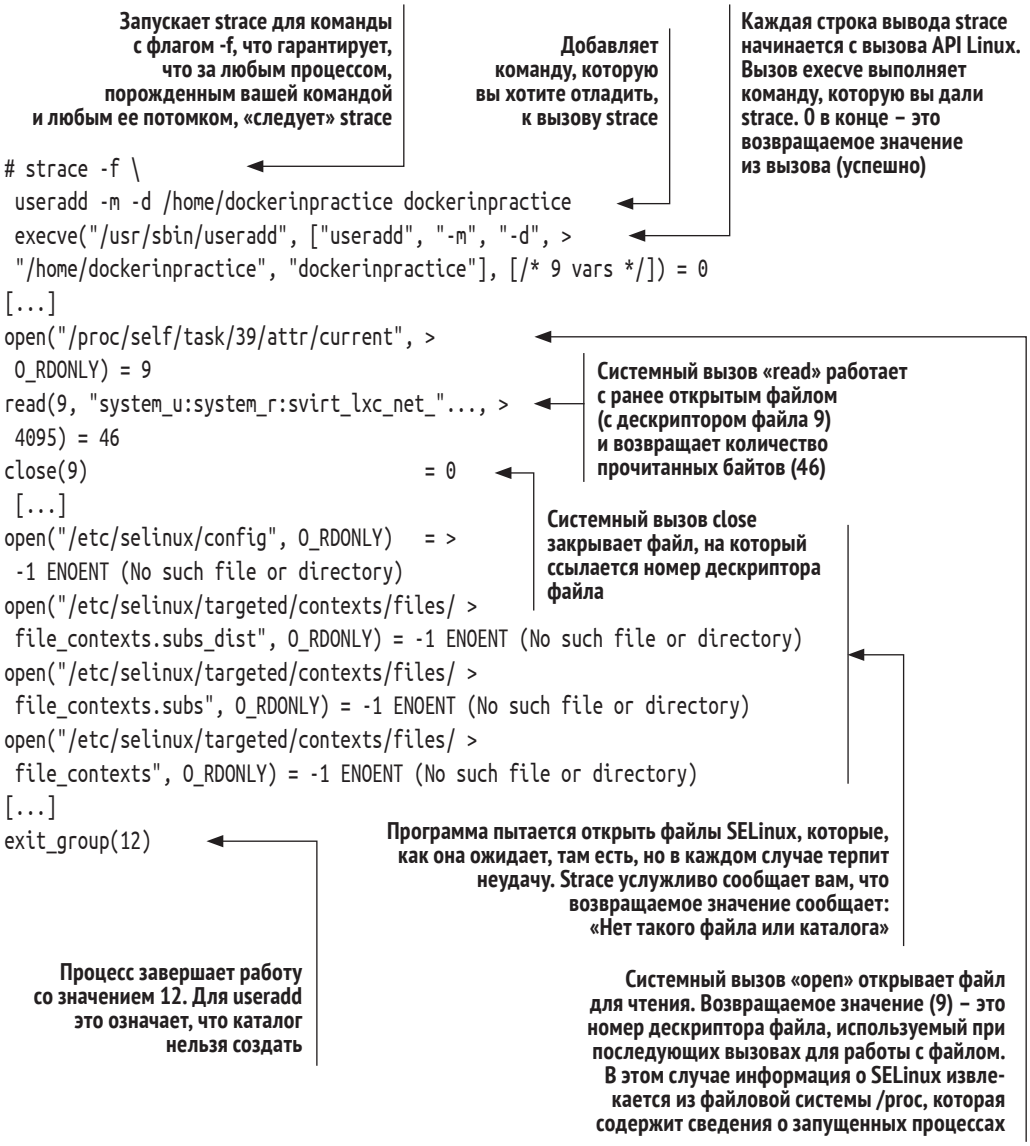


Рис. 16.2 ❖ Как работает strace

Сначала вам нужно установить strace в контейнере с помощью соответствующего менеджера пакетов, а затем выполнить другую команду с добавлением команды `strace`.

Вот пример вывода неудачного вызова `useradd`:



Предыдущий вывод на первый взгляд может показаться запутанным, но, если прочитать его несколько раз, все становится понятным. Каждая строка представляет собой вызов ядра Linux для выполнения действия в так называемом *пространстве ядра* (в отличие от *пространства пользователя*, где действия выполняются программами без передачи ответственности ядру).

> **СОВЕТ.** Если вы хотите узнать больше о конкретном системном вызове, можете выполнить команду `man 2 callname`. Потребуется установить страницы руководства с помощью команды `apt-get install manpages-dev` или аналогичной команды для вашей системы упаковки. Кроме того, поиск в Google «man 2 callname», скорее всего, даст вам то, что нужно.

Это пример того, где абстракции Docker ломаются. В данном случае действие завершается неудачно, потому что программа ожидает присутствия файлов SELinux, поскольку SELinux, по-видимому, включен в контейнере, но сведения о принудительном выполнении хранятся на хосте.

> **СОВЕТ.** Невероятно полезно прочитать страницы `man 2`, касающиеся всех системных вызовов, если вы серьезно настроены стать разработчиком. Поначалу может показаться, что в них полно непонятного жаргона, но, читая различные темы, вы многое узнаете о фундаментальных концепциях Linux. В какой-то момент начнете видеть, что большинство языков происходит от этого корня, и некоторые их причуды и странности станут более понятными. Будьте терпеливы, так как вы не сразу все поймете.

Хотя такие ситуации случаются редко, возможность отладки и понимания того, как ваша программа взаимодействует с помощью strace, – бесценна не только для Docker, но и для более общей разработки.

Если у вас есть очень минимальные образы Docker, возможно, созданные с использованием метода 57, и вы предпочитаете не устанавливать strace на свой контейнер, можно использовать strace с вашего хоста. Вам понадобится использовать команду `docker top <container_id>`, чтобы найти PID процесса в контейнере, и аргумент `-p` для strace, чтобы связать его с конкретным работающим процессом. Не забудьте использовать sudo. Присоединение к процессу потенциально позволяет прочитать его тайны, поэтому для этого требуются дополнительные полномочия.

МЕТОД 114 Извлечение файла из образа

Копирование файла из контейнера легко выполняется с помощью команды `docker cp`. Нередко вам нужно извлечь файл из образа, но нет чистого контейнера для копирования. В этих случаях вы можете искусственно запустить контейнер образа, выполнить команду `docker cp`, а затем удалить контейнер. Это уже три команды, и вы можете столкнуться с проблемами, если, например, образ имеет точку входа по умолчанию, которая требует значимого аргумента.

Этот метод дает вам псевдоним команды, который можно вставить в сценарии запуска оболочки, чтобы сделать все это с помощью одной команды и двух аргументов.

ПРОБЛЕМА

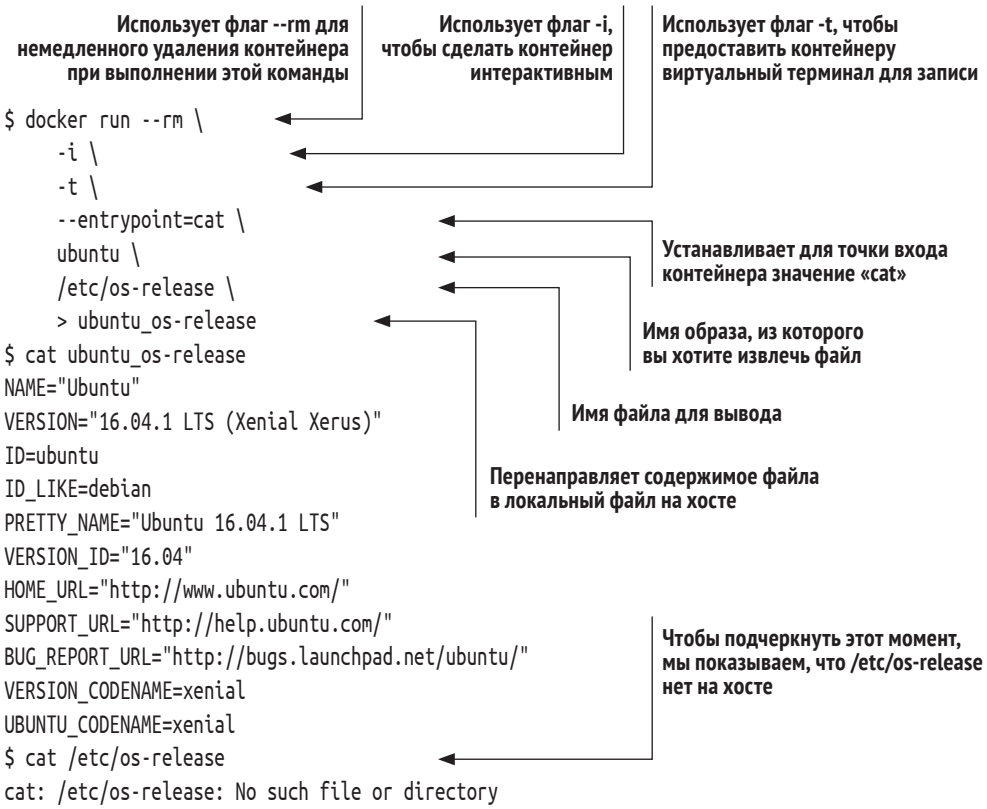
Вы хотите скопировать файл из образа на ваш хост.

РЕШЕНИЕ

Используйте псевдоним, чтобы запустить контейнер из образа с точкой входа, чтобы вывести содержимое файла в файл на хосте.

Сначала мы покажем, как создать команду `docker run` для извлечения файла из образа, а затем вы узнаете, как превратить это в псевдоним для удобства.

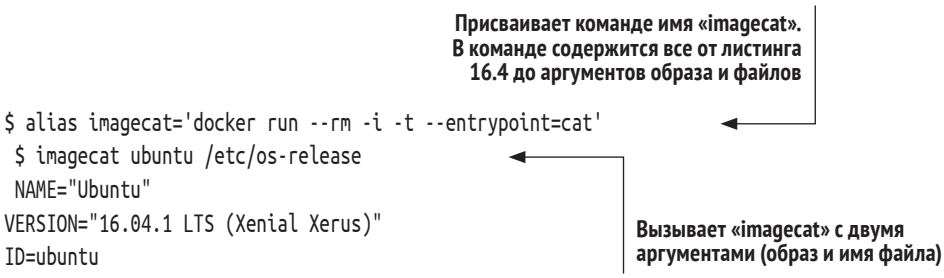
Листинг 16.4. Извлечение файла из образа с помощью команды docker run



Вам может быть интересно, почему мы используем здесь `entrypoint`, а не просто выполняем команду `cat` для вывода содержимого файла. А это потому, что некоторые образы уже установили точку входа. Когда подобное происходит, docker рассматривает `cat` как аргумент команды `entrypoint`, что приводит к нежелательному поведению.

Для удобства вы можете поместить эту команду в псевдоним.

Листинг 16.5. Использование псевдонима для извлечения файла из образа




```
$ alias imagecat='docker run --rm -i -t --entrypoint=cat'
 $ imagecat ubuntu /etc/os-release
 NAME="Ubuntu"
VERSION="16.04.1 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.1 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
```

Данный метод предполагает наличие `cat` в ваших контейнерах. Если вы создавали минимальные контейнеры с помощью метода 58, это может быть не так, поскольку в контейнере присутствует только ваш двоичный файл – стандартных инструментов Linux нет.

Если это так, можете рассмотреть возможность использования команды `docker export` из метода 73, но, вместо того чтобы отправлять их на другой компьютер, вы можете просто извлечь из них нужный файл. Имейте в виду, что для экспорта не нужно выполнять успешный запуск контейнера – вы можете попытаться запустить его с помощью команды, которой нет внутри контейнера, а затем экспортировать остановленный контейнер (или просто используйте команду `docker create`, которая подготавливает контейнер к выполнению без его запуска).

Резюме

- Вы можете передавать аргументы Docker, чтобы отключать различные виды изоляции, либо для большей гибкости контейнеров, либо для повышения производительности.
- Вы можете отключить OOM killer для отдельных контейнеров, чтобы указать, что Linux никогда не должен пытаться восстановить ограниченную память, убивая этот процесс.
- nsenter можно использовать для получения доступа к сетевому контексту контейнера из хоста.
- tcpflow позволяет контролировать весь входящий и исходящий трафик ваших контейнеров без необходимости что-либо перенастраивать или перезапускать.

- strace – это жизненно важное средство, чтобы определить, почему контейнер Docker не работает на конкретном хосте.

На этом мы подошли к концу! Надеемся, что мы открыли вам глаза на то, как использовать Docker, и дали некоторые идеи относительно того, как интегрировать его в вашу компанию или личные проекты.

Если вы хотите связаться с нами или оставить отзыв, создайте тему на форуме *Manning Docker* (https://livebook.manning.com/#!/book/docker-in-practice-second-edition/about-this-book/) или расскажите о проблеме в одном из репозиториев GitHub «docker-in-practice».

Приложение A

Установка и использование Docker

Приемы, описанные в этой книге, иногда требуют создания файлов и клонирования репозиториев GitHub. Чтобы избежать помех, мы предлагаем вам создать новую пустую папку для каждого метода, когда понадобится некое рабочее пространство.

У пользователей Linux это относительно просто, когда дело доходит до установки и использования Docker, хотя мелкие детали могут значительно различаться в разных дистрибутивах Linux.

Вместо того чтобы перечислять здесь различные возможности, мы предлагаем вам обратиться к последней документации к Docker по адресу https://docs.docker.com/installation/. Docker Community Edition подходит для использования с этой книгой.

Хотя мы предполагаем, что вы используете дистрибутив Linux (контейнеры, на которые вы будете смотреть, основаны на Linux, поэтому это упрощает задачу), многие пользователи, интересующиеся Docker, работают на компьютерах под управлением Windows или macOS. Для них стоит отметить, что методы, описанные в этой книге, все еще будут работать, поскольку Docker для Linux официально поддерживается на этих платформах. Те, кто не хочет (или не может) следовать инструкциям, приведенным в предыдущей ссылке, могут использовать один из следующих подходов для настройки демона Docker.

> **ПРИМЕЧАНИЕ.** Microsoft стремится поддерживать парадигму контейнеров и интерфейс управления Docker и сотрудничает с компанией Docker Inc. для создания контейнеров на базе Windows. Несмотря на то что есть много уроков, которые вы можете получить в контейнерах Windows после изучения Linux, существует много вещей, которые отличаются из-за очень разных экосистем и нижележащих уровней. Мы рекомендуем начать с бесплатной электронной книги от Microsoft и Docker, если вам это интересно, но имейте в виду, что это более новая область: https://blogs.msdn.microsoft.com/microsoft_press/2017/08/30/free-ebook-introduction-to-windows-containers/.

Подход с использованием виртуальной машины

Одним из подходов к применению Docker в Windows или macOS является установка полноценной виртуальной машины Linux. После ее установки вы можете использовать виртуальную машину точно так же, как и любую другую нативную машину Linux.

Самый распространенный способ добиться этого – установить VirtualBox. Посетите сайт http://virtualbox.org для получения дополнительной информации и руководства по установке.

Docker-клиент, подключенный к внешнему серверу Docker

Если у вас уже есть демон Docker, настроенный в качестве сервера, можете установить клиента на своем компьютере с Windows или macOS, который общается с ним. Помните, что открытые порты будут доступны на внешнем сервере Docker, а не на вашем локальном компьютере – вам может понадобиться изменить IP-адреса для получения доступа к открытым службам.

Смотрите метод 1 для ознакомления с основами этого более продвинутого подхода и метод 96, чтобы узнать подробности о том, как сделать его безопасным.

Нативный Docker-клиент и виртуальная машина

Распространенный (и официально рекомендуемый) подход заключается в том, чтобы иметь минимальную виртуальную машину с Linux и Docker и клиента Docker, который общается с Docker на этой виртуальной машине.

В настоящее время рекомендуемый и поддерживаемый способ сделать это выглядит так:

- пользователи Mac должны установить Docker для Mac: https://docs.docker.com/docker-for-mac/;
- пользователи Windows должны установить Docker для Windows: https://docs.docker.com/docker-for-windows/.

В отличие от подхода с использованием виртуальной машины, описанного ранее, виртуальная машина, созданная Docker для средств Mac/Windows, очень легкая, так как она запускает только Docker, но стоит помнить, что вам все равно может потребоваться изменить память виртуальной машины в настройках, если вы запускаете ресурсоемкие программы.

Docker для Windows не следует путать с контейнерами Windows (хотя можно использовать контейнеры Windows после установки Docker для Windows). Помните, что для работы Docker на Windows требуется Windows 10 (но *не* Windows 10 Home Edition) по причине зависимости от последних функций Hyper-V.

Если вы используете Windows 10 Home или более старую версию, также можно попробовать установить Docker Toolbox, более старый вариант того же подхода. Docker Inc. описывает этот подход как устаревший, и мы настоятельно

рекомендуем следовать одному из альтернативных методов использования Docker, если это возможно, поскольку вы, скорее всего, столкнетесь с некоторыми странностями, подобными этим:

- вначале тома требуют наличия двойной косой черты (https://github.com/docker/docker/issues/12751);
- поскольку контейнеры работают в виртуальной машине, которая плохо интегрирована в систему, если вы хотите получить доступ к открытому порту из хоста, нужно будет использовать в оболочке команду `docker-machine ip default` для нахождения IP-адреса виртуальной машины, чтобы зайти туда.;
- если вы хотите, чтобы порты находились за пределами хоста, нужно использовать такой инструмент, как socat, для переадресации порта;
- если вы ранее использовали Docker Toolbox и хотите перейти на более новые инструменты, можете найти инструкции по переходу для Mac и Windows на сайте Docker.

Мы не будем рассматривать Docker Toolbox. Как было упомянуто выше, его можно использовать в качестве альтернативного подхода.

Поскольку Windows – это операционная система, которая сильно отличается от Mac и Linux, мы подробнее рассмотрим некоторые распространенные проблемы и их решения. Вы должны были установить Docker для Windows, используя страницу https://docs.docker.com/docker-for-windows/, и удостовериться, что напротив опции **Use Windows Containers Instead of Linux Containers** (Использовать контейнеры Windows вместо контейнеров Linux) *не* установлен флажок. При запуске только что созданного Docker для Windows начнется загрузка Docker, что может занять минуту, – он сообщит, как только запустится, и вы будете готовы к работе!

Можете проверить, работает ли он, открыв PowerShell и выполнив команду `docker run hello-world`. Docker автоматически извлечет образ `hello-world` из Docker Hub и запустит его. Выходные данные этой команды дают краткое описание только что предпринятых шагов относительно обмена данными между клиентом Docker и демоном. Не беспокойтесь, если это не имеет особого смысла, – более подробно о том, что происходит за кулисами, рассказано в главе 2.

Имейте в виду, что в Windows неизбежно будут некоторые странности, поскольку сценарии, используемые в этой книге, предполагают, что вы используете `bash` (или похожую оболочку), и у вас имеется ряд доступных утилит, включая `git` для загрузки примеров кода из всей книги. Мы рекомендуем заглянуть в Cygwin и Windows Subsystem for Linux (WSL), чтобы восполнить этот пробел, – обе они предоставляют Linux-подобную среду с такими командами, как `socat`, `ssh` и `perl`, хотя вы, вероятно, найдете WSL более полным, когда дело доходит до очень специфичных для Linux инструментов, таких как `strace` и `ip` (для `ip addr`).

> **СОВЕТ.** Cygwin, который можно найти по адресу https://www.cygwin.com/, представляет собой набор инструментов от Linux, доступных для работы с Windows. Если вам нужна Linux-подобная среда, с которой можно поэкспериментировать, или вы хотите получить инструмент Linux для нативного использования в Windows (такой как .exe), Cygwin должен быть первым в вашем списке. Он поставляется с менеджером пакетов, поэтому вы можете просматривать доступное программное обеспечение. WSL (описанная на странице https://docs.microsoft.com/en-us/windows/wsl/install-win10), напротив, представляет собой попытку Microsoft предоставить полную эмулированную среду Linux в Windows, насколько это возможно, чтобы вы могли копировать исполняемые файлы с реальных машин Linux и запускать их в WSL. Он еще не идеален (например, нельзя запустить демон Docker), но вы можете эффективно рассматривать его как машину Linux для большинства целей. Полное рассмотрение этих приложений выходит за рамки данного приложения.

Пара замен Windows для некоторых команд и компонентов перечислены ниже, но стоит иметь в виду, что некоторые из них окажутся несовершенны, и это будет заметно – книга посвящена использованию Docker для запуска контейнеров Linux, и имеет смысл, что «полная» установка Linux (будь то «толстая» ВМ, коробка в облаке или установка на вашем локальном компьютере) сможет лучше раскрыть весь потенциал Docker.

- `ip addr` – эта команда обычно используется в книге для поиска IP-адреса вашего компьютера в локальной сети. Эквивалентом в Windows является команда `ipconfig`;
- `strace` – используется в книге для присоединения к процессу, запущенному в контейнере. Взгляните на раздел «Хост-подобный контейнер» в методе 109 для получения подробных сведений о том, как обойти контейнеризацию Docker и получить хост-подобный доступ внутри виртуальной машины, на которой запущен Docker – вам нужно будет запускать оболочку, вместо того чтобы выполнять команду `chroot`, а также использовать дистрибутив Linux с менеджером пакетов, например Ubuntu вместо BusyBox. Оттуда вы можете устанавливать и запускать команды, как будто работаете на хосте. Этот совет относится ко многим командам и едва не позволяет вам рассматривать виртуальную машину Docker как «толстую» виртуальную машину.

Внешнее открытие портов в Windows

Переадресация портов обрабатывается автоматически при использовании Docker для Windows, поэтому вы должны иметь возможность использовать `localhost` для получения доступа к открытым портам, как и ожидается. Брандмауэр Windows может помешать вам, если пытаетесь подключиться с внешних компьютеров.

Находясь в надежной и защищенной сети, вы можете обойти эту проблему, временно отключив брандмауэр Windows, но не забудьте снова включить его!

Один из нас обнаружил, что в конкретной сети это не помогло, в конечном итоге определив, что сеть была настроена в Windows как сеть «Домен», что потребовало использования дополнительных настроек брандмауэра Windows для временного отключения.

Графические приложения в Windows

Запустить графические приложения для Linux в Windows может быть непросто: вам нужно не только заставить весь код работать в Windows, но и решить, как его отобразить. Оконный интерфейс, используемый в Linux (известный как *X Window System*, или X11), не встроен в Windows. К счастью, X позволяет отображать окно приложения через сеть, поэтому можете использовать реализацию X в Windows для отображения приложений, работающих в контейнере Docker.

В Windows существует несколько различных реализаций X, поэтому мы просто рассмотрим установку, которую вы можете получить с помощью Cygwin. Официальная документация находится по адресу: http://x.cygwin.com/docs/ug/setup.html#setup-cygwin-x-install. Следуйте ей. При выборе пакетов для установки вы должны убедиться, что выбраны `xorg-server`, `xinit` и `xhost`.

После завершения установки откройте терминал Cygwin и выполните команду `XWin: 0-listen tcp -multiwindow`. После этого на вашем компьютере, где установлена Windows, будет запущен X-сервер с возможностью прослушивать соединения из сети (`-listen tcp`) и отображать каждое приложение в своем собственном окне (`-multiwindow`), а не в одном окне, которое работает как виртуальный экран для отображения приложений. После запуска вы должны увидеть значок «X» в области уведомлений панели задач.

> **ПРИМЕЧАНИЕ.** Хотя этот X-сервер может прослушивать сеть, в настоящее время он доверяет только локальному компьютеру. Во всех случаях, которые мы видели, он разрешает доступ с вашей виртуальной машины Docker, но если вы заметили проблемы с авторизацией, то можете попробовать выполнить небезопасную команду `xhost +`, чтобы разрешить доступ со всех компьютеров. Если вы это сделаете, убедитесь, что ваш брандмауэр настроен на отклонение любых попыток подключения из сети – ни при каких обстоятельствах не выполняйте ее с отключенным брандмауэром Windows! Если вы все же запустите ее, не забудьте потом выполнить команду `xhost-`, чтобы снова сделать ее безопасной.

Пришло время опробовать ваш X-сервер. Узнайте IP-адрес вашего локального компьютера с помощью команды `ipconfig`. Обычно мы добиваемся успеха при использовании IP-адреса на внешнем адаптере, будь то беспроводное или проводное соединение, так как похоже, что это то место, откуда исходят

соединения из ваших контейнеров. Если у вас есть несколько таких адаптеров, возможно, потребуется опробовать IP-адрес каждого из них по очереди.

Запуск вашего первого графического приложения должен быть таким же простым, как и запуск команды `docker run -e DISPLAY=$MY_IP:0 --rm fr3nd/xeyes` в PowerShell, где `$ MY_IP` – это IP-адрес, который вы нашли.

Если вы не подключены к сети, можете упростить ситуацию с помощью небезопасной команды `xhost +`, чтобы разрешить использование интерфейса `DockerNAT`. Как и раньше, не забудьте выполнить команду `xhost -`, когда закончите.

Если нужна помощь

Если вы используете операционную систему, отличную от Linux, и хотите получить дополнительную помощь или совет, в документации по Docker (https://docs.docker.com/install/) содержатся последние официально рекомендуемые советы для пользователей Windows и macOS.

Приложение B

Настройка Docker

В различных местах этой книги вам рекомендуется изменить конфигурацию Docker, чтобы сделать изменения постоянными при запуске хост-компьютеров Docker. Приложение B расскажет вам о лучших методах для достижения этой цели. Дистрибутив операционной системы, который вы используете, будет значительным в этом контексте.

Настройка Docker

Расположение файлов конфигурации большинства основных дистрибутивов указано в табл. B.1.

Таблица B.1 ❖ Расположение файла конфигурации Docker

Дистрибутив	Конфигурация
Ubuntu, Debian, Gentoo	/etc/default/docker
OpenSuse, CentOS, Red Hat	/etc/sysconfg/docker

Обратите внимание, что в некоторых дистрибутивах конфигурация хранится в одном файле, тогда как в других используется каталог и несколько файлов. Например, в Red Hat Enterprise License есть файл /etc/sysconfig/docker/docker-storage, который по соглашению содержит конфигурацию, относящуюся к опциям хранилища для демона Docker.

Если в вашем дистрибутиве нет файлов, соответствующих названиям в табл. B.1, стоит проверить папку /etc/docker, поскольку там могут быть соответствующие файлы.

Внутри этих файлов осуществляется управление аргументами команды запуска демона Docker. Например, при редактировании строка, подобная той, что показана ниже, позволяет установить начальные аргументы для демона Docker на вашем хосте.

```
DOCKER_OPTS=""
```

Например, если вы хотите изменить расположение корневого каталога Docker по умолчанию (то есть /var/lib/docker), вы можете изменить предыдущую строку следующим образом:

```
DOCKER_OPTS="-g /mnt/bigdisk/docker"
```

Если в вашем дистрибутиве используются файлы конфигурации systemd (в отличие от / etc), вы также можете выполнить поиск строки `ExecStart` в файле docker в папке systemd и при желании изменить ее. Этот файл может быть расположен, например, в /usr/lib/systemd/system/service/docker или /lib/systemd/system/docker.service. Вот пример файла:

```
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.io
After=network.target

[Service]
Type=notify
EnvironmentFile=-/etc/sysconfig/docker
ExecStart=/usr/bin/docker -d --selinux-enabled
Restart=on-failure
LimitNOFILE=1048576
LimitNPROC=1048576

[Install]
WantedBy=multi-user.target
```

Строка `EnvironmentFile` отсылает сценарий запуска в файл с записью `DOCKER_OPTS`, которую мы обсуждали ранее. Если вы измените файл systemctl напрямую, нужно будет выполнить команду `systemctl daemon-reload`, чтобы убедиться, что демон system подхватил изменение.

Перезапуск Docker

Изменение конфигурации для демона Docker недостаточно – чтобы применить изменения, демон должен быть перезапущен. Имейте в виду, что в этом случае будут остановлены все работающие контейнеры и отменены все выполняющиеся загрузки образов.

Перезапуск с помощью systemctl

Большинство современных дистрибутивов Linux используют systemd для управления запуском служб на компьютере. Если вы выполните команду `systemctl` в командной строке и получите страницы вывода, то ваш хост работает под управлением systemd. Получив сообщение «команда не найдена», переходите к следующему разделу.

Если вы хотите внести изменения в вашу конфигурацию, можете остановить и запустить Docker следующим образом:

```
$ systemctl stop docker
$ systemctl start docker
```

Или можно просто перезагрузить:

```
$ systemctl restart docker
```

Проверьте ход выполнения с помощью этих команд:

```
$ journalctl -u docker
$ journalctl -u docker -f
```

Первая строка выводит доступные журналы для процесса демона docker. Вторая строка просматривает журналы на предмет любых новых записей.

Перезапуск с помощью service

Если ваша система использует набор сценариев инициализации на основе System V, попробуйте выполнить команду `service --status-all`. Если она вернет список служб, можете использовать `service` для перезапуска Docker с новой конфигурацией.

```
$ service docker stop
$ service docker start
```

Приложение C

Vagrant

В различных местах этой книги мы используем виртуальные машины, чтобы продемонстрировать метод, который требует полного представления машины или даже оркестровки с несколькими виртуальными машинами. Vagrant предлагает простой способ запуска, подготовки и управления виртуальными машинами из командной строки. Он доступен на нескольких платформах.

Настройка

Перейдите на страницу https://www.vagrantup.com и следуйте приведенным там инструкциям, чтобы выполнить настройку.

Графические интерфейсы

При выполнении команды `vagrant up` для запуска виртуальной машины Vagrant считывает локальный файл Vagrantfile, чтобы определить настройки.

Полезный параметр, который вы можете создать или изменить в разделе для своего провайдера, это:

```
v.gui = true
```

Например, если ваш провайдер – VirtualBox, типичный раздел конфигурации выглядит следующим образом:

```
Vagrant.configure(2) do |config|
  config.vm.box = "hashicorp/precise64"

  config.vm.provider "virtualbox" do |v|
    v.memory = 1024
    v.cpus = 2
    v.gui = false
  end
end
```

Вы можете изменить значение `false` в строке `v.gui` на `true` (или добавить его, если его там еще нет), перед тем как выполнить команду `vagrant up`, чтобы получить графический интерфейс пользователя для работающей виртуальной машины.

ПОДСКАЗКА. *Провайдер* в Vagrant – это название программы, которая предоставляет среду ВМ. Для большинства пользователей это будет `virtualbox`, но также может быть `libvirt`, `openstack` или `vmware_fusion` (среди прочих).

Память

Vagrant использует виртуальные машины для создания своих сред, и они могут потреблять много памяти. Если вы используете кластер из трех узлов, где каждая виртуальная машина занимает 2 Гб памяти, вашей машине потребуется 6 Гб доступной памяти. Если ваша машина не может работать, скорее всего, это вызвано нехваткой памяти. Единственное решение – остановить ненужные виртуальные машины или приобрести больше памяти. Возможность избежать этого – одна из причин, по которой Docker – более мощный, чем виртуальная машина; вам не нужно предварительно выделять ресурсы для контейнеров – они просто потребляют то, что им нужно.

Предметный указатель

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Иан Милл, Эйдан Хобсон Сейерс

Docker на практике

Главный редактор *Мовчан Д.А.*
dmkpress@gmail.com
Перевод *Беликов Д.А.*
Корректор *Чистякова Л.А.*
Верстка *Антонова А.И.*
Дизайн обложки *Мовчан А.Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 41,93. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**