O'REILLY®

Использование Docker

Эдриен Моуэт

ДМК ПРЕСС
издательство

Эдриен Моуэт

Использование Docker

Chris Simmonds

Using Docker

Developing and Deploying Software with Containers

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Эдриен Моуэт

Использование Docker

Разработка и внедрение программного обеспечения при помощи технологии контейнеров

ДМК ПРЕСС
издательство
Москва, 2017

УДК 004.451Docker
ББК 32.972.1
М89

Моуэт Э.
М89 Использование Docker / пер. с англ. А. В. Снастина; науч. ред. А. А. Маркелов. – М.: ДМК Пресс, 2017. – 354 с.: ил.

ISBN 978-5-97060-426-7

Контейнеры Docker предоставляют простые быстрые и надежные методы разработки, распространения и запуска программного обеспечения, особенно в динамических и распределенных средах. Из книги вы узнаете, почему контейнеры так важны, какие преимущества вы получите от применения Docker и как сделать Docker частью процесса разработки. Вы последовательно пройдете по всем этапам, необходимым для создания, тестирования и развертывания любого веб-приложения, использующего Docker. Также вы изучите обширный материал — начиная от основ, необходимых для запуска десятка контейнеров, и заканчивая описанием сопровождения крупной системы со множеством хостов в сетевой среде со сложным режимом планирования.

Издание предназначено разработчикам, инженерам по эксплуатации и системным администраторам.

УДК 004.451Docker
ББК 32.972.1

ISBN 978-1-491-91576-9 (анг.)
ISBN 978-5-97060-426-7 (рус.)

Посвящается всем, кто пытается что-то делать,
независимо от того, чем заканчиваются их попытки:
неудачей или успехом.

Содержание

Предисловие

Контейнеры – это простые и переносимые
между платформами хранилища
для приложения и его зависимостей.

Написанное выше звучит сухо и скучно. Но возможности контейнеров вовсе не ограничиваются усовершенствованием процессов; при правильном использовании контейнеры способны изменить всю картину. Такое обоснование возможности использования контейнеров в архитектурах и рабочих процессах выглядит весьма убедительно, и похоже на то, что все крупные ИТ-компании, которые раньше никогда не слышали о контейнерах и Docker, буквально за год перешли к активным исследованиям и практическому применению этих технологий.

Рост популярности Docker был поразительным. Я не помню, чтобы какая-либо другая среда оказывала столь основательное и глубокое воздействие в области информационных технологий. Эта книга представляет собой попытку помочь читателям понять, почему контейнеры так важны, какую пользу принесет применение контейнеризации и, что самое главное, как перейти к этой технологии.

Для кого предназначена эта книга

В этой книге предпринята попытка дать целостный подход к программной среде Docker с обоснованием ее использования и описанием способов ее практического применения и внедрения в рабочий поток разработки программного обеспечения. Книга полностью охватывает весь жизненный цикл программного продукта – от разработки до промышленного тиражирования и сопровождения.

Я старался не делать каких-либо предположений об уровне знаний читателя об ОС Linux и о процессе разработки программного обеспечения в целом. К предполагаемому кругу читателей прежде всего относятся разработчики программного обеспечения, инженеры по эксплуатации и системные администраторы (особенно те, кто серьезно интересуется развитием методики DevOps). Однако руководители, обладающие достаточными техническими знаниями, и все интересующиеся этой темой также могут кое-что почерпнуть из этой книги.

Почему я написал эту книгу

Мне повезло оказаться среди тех, кто узнал о Docker и стал использовать эту программную среду в самом начале ее стремительного взлета. Когда появилась возможность написать книгу о Docker, я ухватился за нее обеими руками. Если мои труды помогут кому-то из вас понять эту технологию и проделать бóльшую часть

пути к контейнеризации, то я буду считать это более значимым достижением, чем годы разработки программного обеспечения.

Искренне надеюсь, что вы с удовольствием прочтете эту книгу и она поможет вам при переходе к практическому применению Docker в вашей организации.

Структура книги

Книга состоит из трех основных частей:

- часть I начинается с определения контейнеров и краткого описания их свойств, которые представляют интерес для читателей, далее следует глава-руководство по основам Docker. Первая часть завершается большой главой, в которой описаны главные концепции и технология, на которых основана программная среда Docker, включая обзор различных команд среды Docker;
- в части II рассматривается использование Docker в жизненном цикле разработки программного обеспечения. Описан процесс установки среды разработки, затем создание простого веб-приложения, которое используется в дальнейшем как пример на протяжении всей части II. Отдельная глава посвящена разработке, тестированию и объединению с существующей программной средой, также рассматриваются процедура развертывания контейнеров и способы эффективного наблюдения и журналирования в целевой рабочей системе;
- в части III все внимание сосредоточено на мельчайших подробностях, а также на инструментах и методиках, необходимых для безопасного и надежного функционирования многохостовых кластеров, состоящих из Docker-контейнеров. Если вы уже используете Docker и хотите понять, как выполнить масштабирование или решить проблемы безопасности и работы в сетевой среде, то эта часть предназначена для вас.

Условные обозначения и соглашения, принятые в книге

В книге используются следующие типографские соглашения:

Курсив – используется для смыслового выделения важных положений, новых терминов, имен команд и утилит, а также имен и расширений файлов и каталогов.

`Моноширинный шрифт` – используется для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов, ключевых слов и других программных конструкций и элементов исходного кода.

`Моноширинный полужирный шрифт` – используется для обозначения команд или фрагментов текста, которые пользователь должен ввести дословно без изменений.

`Моноширинный курсив` – используется для обозначения в исходном коде или в командах шаблонных меток-заполнителей, которые должны быть заменены соответствующими контексту реальными значениями.

Такая пиктограмма обозначает совет, указание или примечание общего характера.

Эта пиктограмма предупреждает о неочевидных и скрытых «подводных камнях» и «ловушках», которых следует всячески избегать.

Примеры исходного кода

Дополнительные материалы (примеры исходного кода, учебные задания и т. п.) можно получить по ссылке https://github.com/using-docker/.

Эта книга написана для того, чтобы помочь вам в работе. Вообще говоря, вы можете использовать код из данной книги в своих программах и в документации. Если вы копируете для собственных нужд фрагмент исходного кода незначительного размера, то нет необходимости обращаться к автору и издателям для получения разрешения на это. Например, при включении в свою программу нескольких небольших фрагментов кода из книги вам не потребуется какое-либо специальное разрешение. Но для продажи или распространения CD-диска с примерами из книг издательства O'Reilly необходимо будет получить официальное разрешение на подобные действия. При ответах на вопросы можно цитировать текст данной книги и приводить примеры кода из нее без дополнительных условий. При включении крупных фрагментов исходного кода из книги в документацию собственного программного продукта также потребуется официальное разрешение.

При цитировании мы будем особенно благодарны за библиографическую ссылку на источник. Обычно ссылка на источник состоит из названия книги, имени автора, наименования издательства и номера по ISBN-классификации. Например: «Using Docker (Использование Docker) by Adrian Mouat (Эдриэн Моуэт) (O'Reilly). Copyright 2016 Adrian Mouat, 978-1-491-91576-9».

Если у вас возникли сомнения в легальности использования примеров исходного кода без получения специального разрешения при условиях, описанных выше, то без колебаний обращайтесь по адресу электронной почты: permissions@oreilly.com.

Онлайн-сервис Safari Books

Онлайн-сервис Safari Books (http://safaribooksonline.com) – это электронная библиотека с весьма быстрым обслуживанием заявок, которая предоставляет в формате книг и в видеоформате высококачественные материалы, созданные ведущими авторами с мировой известностью в технических дисциплинах и в сфере бизнеса.

Специалисты-профессионалы в различных областях техники, разработчики программного обеспечения, веб-дизайнеры, бизнесмены и творческие работники используют онлайн-сервис Safari Books как основной ресурс для научных исследований, решения профессиональных задач, обучения и подготовки к сертификации.

Онлайн-сервис Safari Books предлагает широкий ассортимент разнообразных подборок материалов с индивидуальным формированием цены каждой такой подборки для организаций, государственных учреждений, учебных заведений и частных лиц.

В рамках общей базы данных с единым механизмом полнотекстового поиска подписчики получают доступ к тысячам книг, учебных видеоматериалов и даже к еще не опубликованным рукописям, предоставленным такими известными издательствами, как O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, и многими другими. Более подробную информацию об онлайн-сервисе Safari Books можно получить на сайте https://www.safaribooksonline.com/.

От издательства

Замечания, предложения и вопросы по этой книге отправляйте по адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США или в Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги создана специальная веб-страница, на которой мы разместили список обнаруженных опечаток и ошибок, исходные коды примеров и другую дополнительную информацию. Сетевой адрес этой страницы: http://bit.ly/using-docker.

Комментарии и технические вопросы по теме данной книги отправляйте по адресу электронной почты: bookquestions@oreilly.com.

Для получения более подробной информации о книгах, учебных курсах, конференциях и новостях издательства O'Reilly посетите наш веб-сайт: http://www.oreilly.com/.

Издательство представлено:

- в соцсети Facebook: http://facebook.com/oreilly;
- в Twitter: http://twitter.com/oreillymedia;
- и в YouTube: http://www.youtube.com/oreillymedia.

Благодарности

Я в высшей степени благодарен за всю помощь, советы и критику, которые получал во время написания этой книги. Если в приведенном ниже списке я пропустил чье-то имя, примите мои извинения; ваш вклад оценен по достоинству вне зависимости от содержания этого списка.

За плодотворное активное сотрудничество благодарю Элли Хьюм (Ally Hume), Тома Сагдена (Tom Sugden), Лукаша Гумински (Lukasz Guminski), Тайлейе Элему (Tilaye Alemu), Себастиена Гусгоэна (Sebastien Goasduen), Максима Белоусова (Maxim Beloussov), Михаэля Белена (Michael Boelen), Ксению Бурлаченко (Ksenia Burlachenko), Карлоса Санчеса (Carlos Sanchez), Дэниэла Брайанта (Daniel Bryant), Кристофера Холмштедта (Christoffer Holmstedt), Майка Рэтбана (Mike Rathbun), Фабрицио Соппелза (Fabrizio Soppelsa), Юн-Чжин Ху (Yung-Jin Hu), Йоуни Миикки (Jouni Miikki) и Дэйла Бьюли (Dale Bewley).

За обсуждение технических вопросов и технологий, описываемых в этой книге, спасибо Эндрю Кеннеди (Andrew Kennedy), Питеру Уайту (Peter White), Алексу Поллитту (Alex Pollitt), Финтэну Райану (Fintan Ryan), Шону Крэмптону (Shaun Crampton), Спайку Кертису (Spike Curtis), Алексису Ричардсону (Alexis Richardson), Илье Дмитриченко (Ilya Dmitrichenko), Кейси Биссон (Casey Bisson), Тийсу Шнитгеру (Tijs Schnitger), Шен Лян (Sheng Liang), Тимо Дерштаппену (Timo Derstappen), Пуйа Аббасси (Puja Abbassi), Александру Ларссону (Alexander Larsson) и Келзи Хайтауэр (Kelsey Hightower). Отдельная благодарность Кевину Годэну (Kevin Gaudin) за разрешение неограниченного использования модуля monsterid.js.

За всю оказанную мне помощь благодарю коллектив издательства O'Reilly, особая благодарность моему редактору Брайану Андерсону (Brian Anderson) и Меган Бланшетт (Meghan Blanchette) за работу на начальном этапе процесса.

Диого Моника (Diogo Mónica) и Марк Коулмэн (Mark Coleman), спасибо вам обоим за то, что откликнулись на мою просьбу о срочной помощи.

Кроме того, следует особо отметить две компании: Container Solutions и CloudSoft. Джейми Добсон (Jamie Dobson) и Container Solutions поощряли ведение моего блога и выступления на соответствующих мероприятиях, а также обеспечивали связь с некоторыми людьми, оказавшими влияние на содержание этой книги. Компания CloudSoft любезно предоставила в мое распоряжение офис на время написания книги и провела семинар Edinburgh Docker – оба этих события были чрезвычайно важными для меня.

За терпеливую поддержку моей одержимости и моих роптаний во время работы над книгой спасибо моим друзьям и моей семье – вам хорошо известно, что вы значите для меня (хотя вряд ли прочтете когда-либо эти строки).

Наконец, спасибо ди-джеям BBC 6 Music, предоставившим саундтрек для этой книги, в том числе Лорен Лаверн (Lauren Laverne), Рэдклифф и Макони (Radcliffe and Maconie), Шону Кэвени (Shaun Keaveny) и Игги Поп (Iggy Pop).

Часть I

ПРЕДПОСЫЛКИ И ОСНОВЫ

В первой части книги вы узнаете, что такое контейнеры и почему они становятся столь широко распространенными. Далее следуют введение в технологию Docker и описание основных концепций, которые необходимо понимать, чтобы наилучшим образом использовать контейнеры.

Глава 1

Что такое контейнеры и для чего они нужны

Контейнеры коренным образом изменяют способ разработки, распространения и функционирования программного обеспечения. Разработчики могут создавать программное обеспечение на локальной системе, точно зная, что оно будет работать одинаково в любой операционной среде – в программном комплексе ИТ-отдела, на ноутбуке пользователя или в облачном кластере. Инженеры по эксплуатации могут сосредоточиться на поддержке работы в сети, на предоставлении ресурсов и на обеспечении бесперебойной работы и тратить меньше времени на конфигурирование окружения и на «борьбу» с системными зависимостями. Масштабы перехода к практическому применению контейнеров стремительно растут во всей промышленности информационных технологий, от небольших стартапов до крупных предприятий. Разработчики и инженеры по эксплуатации должны понимать, что необходимость постоянного использования контейнеров будет возрастать в течение нескольких следующих лет.

Контейнеры (containers) представляют собой средства инкапсуляции приложения вместе с его зависимостями. На первый взгляд контейнеры могут показаться всего лишь упрощенной формой виртуальных машин (virtual machines – VM) – как и виртуальная машина, контейнер содержит изолированный экземпляр операционной системы (ОС), который можно использовать для запуска приложений.

Но контейнеры обладают некоторыми преимуществами, обеспечивающими такие варианты использования, которые трудно или невозможно реализовать в обычных виртуальных машинах:

- контейнеры совместно используют ресурсы основной ОС, что делает их на порядок более эффективными. Контейнеры можно запускать и останавливать за доли секунды. Для приложений, запускаемых в контейнерах, накладные расходы минимальны или вообще отсутствуют, по сравнению с приложениями, запускаемыми непосредственно под управлением основной ОС;
- переносимость контейнеров обеспечивает потенциальную возможность устранения целого класса программных ошибок, вызываемых незначительными изменениями рабочей среды, – лишается обоснования древний довод разработчика: «но это работает на моем компьютере»;

- упрощенная сущность контейнера означает, что разработчики могут одновременно запускать десятки контейнеров, что дает возможность имитации работы промышленной распределенной системы. Инженеры по эксплуатации могут запустить на одном хосте намного больше контейнеров, чем при использовании отдельных виртуальных машин;
- кроме того, контейнеры предоставляют преимущества конечным пользователям и разработчикам без необходимости развертывания приложения в облаке. Пользователи могут загружать и запускать сложные приложения без многочасовой возни с конфигурированием и проблемами при установке и при этом не беспокоиться о каких-либо изменениях в их локальных системах. В свою очередь, разработчики подобных приложений могут избежать проблем, связанных с различиями в конфигурациях пользовательских сред и с доступностью зависимостей для этих приложений.

И что более важно, существуют принципиальные различия в целях использования виртуальных машин и контейнеров – целью применения виртуальной машины является полная эмуляция инородной программной (операционной) среды, тогда как цель применения контейнера – сделать приложения переносимыми и самодостаточными.

Сравнение контейнеров с виртуальными машинами

Несмотря на то что контейнеры и виртуальные машины на первый взгляд кажутся похожими, между ними существуют важные различия, которые проще всего продемонстрировать на графических схемах.

На рис. 1.1 показаны три приложения, работающих в отдельных виртуальных машинах на одном хосте. Здесь требуется гипервизор[1] для создания и запуска виртуальных машин, управляющий доступом к нижележащей ОС и к аппаратуре, а также при необходимости интерпретирующий системные вызовы. Для каждой виртуальной машины необходимы полная копия ОС, запускаемое приложение и все библиотеки поддержки.

В противоположность описанной схеме на рис. 1.2 показано, как те же самые три приложения могут работать в системе с контейнерами. В отличие от виртуальных машин, ядро[2] хоста совместно используется (разделяется) работающими кон-

---

[1] На диаграмме изображен гипервизор типа 2, такой как Virtualbox или VMWare Workstation, который работает поверх основной ОС. Существуют также гипервизоры типа 1, такие как Xen, работающие непосредственно на «голом железе».

[2] Ядро (kernel) является главным компонентом любой ОС и отвечает за предоставление приложениям важнейших системных функций (ресурсов), связанных с оперативной памятью, процессором и доступом к различным устройствам. Полноценная ОС состоит из ядра и разнообразных системных программ, таких как программа инициализации системы, компиляторы и оконные менеджеры.

тейнерами. Это означает, что контейнеры всегда ограничиваются использованием того же ядра, которое функционирует на хосте. Приложения Y и Z пользуются одними и теми же библиотеками и могут совместно работать с этими данными, не создавая избыточных копий. Внутренний механизм контейнера отвечает за пуск и остановку контейнеров так же, как гипервизор в виртуальной машине. Тем не менее процессы внутри контейнеров равнозначны собственным процессам ОС хоста и не влекут за собой дополнительных накладных расходов, связанных с выполнением гипервизора.

Как виртуальные машины, так и контейнеры можно использовать для изоляции приложений друг от друга при работе на одном хосте. Дополнительная степень изоляции в виртуальных машинах обеспечивается гипервизором, и виртуальные машины являются многократно проверенной в реальных условиях технологией. Контейнеры представляют собой сравнительно новую технологию, и многие организации не вполне доверяют методике изоляции функциональных компонентов в контейнерах, пока не увидят воочию весомых доказательств преимуществ контейнеров. По этой причине часто встречаются гибридные системы, в которых контейнеры работают внутри виртуальных машин для совмещения преимуществ обеих технологий.

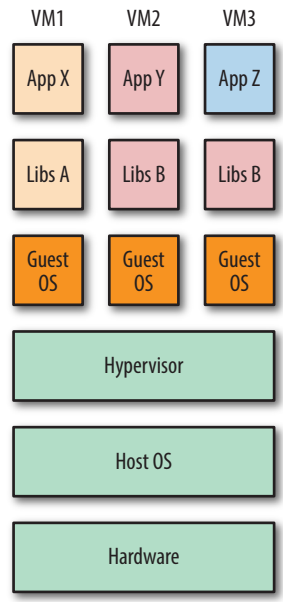

Рис. 1.1. Три виртуальные машины, работающие на одном хосте

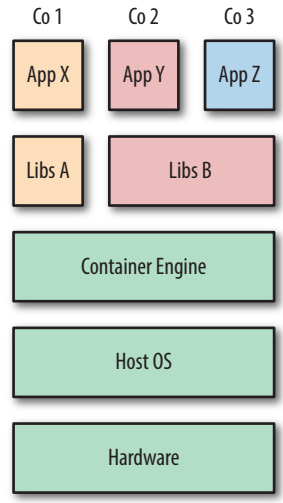

Рис. 1.2. Три контейнера, работающих на одном хосте

Docker и контейнеры

Контейнеры являются старой теоретической концепцией. Еще несколько десятков лет назад в Unix-системах существовала команда chroot, обеспечивающая простейшую форму изоляции части файловой системы. С 1998 года в ОС FreeBSD появилась утилита jail, распространяющая изоляционные возможности chroot на процессы. В 2001 году реализация Solaris Zones обеспечивала относительно полную технологию контейнеризации, но ее применение было ограничено только ОС Solaris. В том же 2001 году компания Parrallels Inc (в дальнейшем SWsoft) выпустила коммерческую версию технологии контейнеров Virtuozzo для ОС Linux, а в 2005 году открыла исходные коды ядра этой технологии под именем OpenVZ[1]. Затем компания Google начала разаработку CGroups для ядра ОС Linux и приступила к перемещению своей инфраструктуры в контейнеры. Проект Linux Containers (LXC) был создан в 2008 году и вскоре объединил CGroups, пространства имен ядра, технологию chroot и некоторые другие технологии, чтобы предоставить полностью завершенное решение по обеспечению контейнеризации. Наконец, в 2013 году Docker стал последним штрихом в общей картине состояния контейнеризации, и эта технология по праву заняла свое место как одно из главных направлений развития ИТ-индустрии.

В основу Docker была заложена существующая технология Linux-контейнеров с разнообразными обертками и расширениями – в основном использующими пе-

[1] Технология OpenVZ никогда не применялась в массовом порядке, возможно, из-за того, что для ее реализации требуется модифицированное ядро.

реносимые образы и удобный для пользователя интерфейс – для создания полностью готового к применению решения, обеспечивающего создание и распространение контейнеров. Платформа Docker состоит из двух отдельных компонентов: Docker Engine, механизма, отвечающего за создание и функционирование контейнеров, и Docker Hub, облачного сервиса для распространения контейнеров.

Механизм Docker Engine предоставляет эффективный и удобный интерфейс для запуска контейнеров. До этого для запуска контейнеров, использующих такую технологию, как, например, LXC, требовались изрядный запас специальных знаний в этой области и большой объем ручной работы. Docker Hub предоставляет огромное количество образов контейнеров с открытым доступом для загрузки, позволяя пользователям быстро начать работу с ними и избежать рутинной работы, ранее уже проделанной другими людьми. Несколько позже были разработаны инструментальные средства для Docker: Swarm – менеджер кластеров, Kitematic – графический пользовательский интерфейс для работы с контейнерами и Machine – утилита командной строки для поддержки работы Docker-хостов.

Ввиду открытости исходных кодов Docker Engine стали возможными создание большого сообщества приверженцев технологии Docker и его постоянный рост, а также вытекающие из этого преимущества массовой помощи в исправлении ошибок и внесении усовершенствований. Быстрый рост популярности этой технологии свидетельствует о том, что Docker действительно становится стандартом де-факто, и это уже привело к разработке независимых стандартов для функциональности времени выполнения и формата контейнеров. В 2015 году наивысшей точкой развития стало создание Open Container Initiative, «управляющей структуры», поддерживаемой Docker, Microsoft, CoreOS и многими другими известными организациями, целью которой является разработка промышленного стандарта. Основой для разработки служат формат и функциональные формы времени выполнения Docker-контейнера.

Темпы роста применения контейнеров в наибольшей степени обеспечили разработчики, которым с самого начала были предоставлены инструментальные средства для эффективного использования контейнеров. Минимальное время от начала разработки до ввода в эксплуатацию Docker-контейнеров чрезвычайно важно для разработчиков, которым необходимы быстрые итеративные циклы разработки с возможностью немедленного просмотра результатов изменений, внесенных в исходный код. Переносимость и изолированность контейнеров способствуют упрощению сотрудничества с другими разработчиками и инженерами по эксплуатации: разработчики могут быть вполне уверены в том, что их код будет работать в любых программных средах, а инженеры по эксплуатации могут сосредоточиться на организации и настройке работы контейнеров на хостах, не беспокоясь о том, какой код выполняется внутри контейнеров.

Новшества, внесенные с применением технологии Docker, существенно изменили способ разработки программного обеспечения. Без Docker контейнеры, вероятнее всего, еще долго оставались бы малозаметным направлением в развитии информационных технологий.

Метафорическая аналогия с доставкой грузов

Философию Docker часто описывают с помощью метафоры «доставки грузов в контейнерах», которая вполне очевидно объясняет происхождение имени Docker. Ниже приводится наиболее типичное изложение этой философии.

При транспортировке грузов используются разнообразные средства, включая трейлеры, автопогрузчики, краны, поезда и корабли. Эти средства должны обладать возможностями работы с широким спектром грузов различных размеров и соблюдать многочисленные специальные требования (например, при транспортировке упаковок кофе, бочек с опасными химикатами, коробок с электронными товарами, парков дорогих автомобилей и стеллажей с замороженными мясными продуктами). На протяжении многих лет это был тяжелый и дорогостоящий процесс, требующий больших трудозатрат многих людей, в том числе и портовых докеров, для погрузки и выгрузки вручную различных предметов в каждом транзитном пункте (рис. 1.3).

Рис. 1.3. Работа докеров в порту Бристоля (Англия) в 1940 году (снимок предоставлен отделом фотографии Министерства информации Великобритании)

Коренной переворот в транспортной промышленности произвело появление универсальных грузовых контейнеров. Эти контейнеры имели стандартные размеры и были специально спроектированы таким образом, чтобы для их перемещения между различными видами транспортных средств требовался минимум ручного труда. Все виды грузового транспорта создавались с учетом характеристик этих

контейнеров – от автопогрузчиков и кранов до грузовиков, поездов и кораблей. Контейнеры-рефрижераторы и изолированные контейнеры предназначены для перевозки грузов с жестко заданным температурным режимом, например некоторых продуктов питания и фармацевтических товаров. Кроме того, преимущества стандартизации распространились и на другие вспомогательные системы, такие как система маркировки грузов и система герметизации и опечатывания контейнеров. Это означает, что ответственность за содержимое контейнеров полностью перекладывается на производителей транспортируемых товаров, а работники транспорта могут полностью сосредоточиться на перевозке и хранении самих контейнеров.

Основная цель программной среды Docker – перенести преимущества стандартизации контейнеров в область информационных технологий. В последние годы программные системы отличаются впечатляющим разнообразием. Прошли времена технологии LAMP[1], реализованной на одном компьютере. Типичная современная система может состоять из фреймворков JavaScript, баз данных NoSQL, очередей сообщений, прикладных программных интерфейсов REST и внутренних компонентов, написанных на различных языках программирования. Такой стек должен частично или полностью работать на разнообразной аппаратуре – от личного компьютера или ноутбука разработчика и небольшого тестового кластера до крупного промышленного узла провайдера облачных сервисов. Все эти среды отличаются разнообразием, используют различные операционные системы с разными версиями библиотек и работают на различной аппаратуре. Короче говоря, проблема точно такая же, как и в транспортной промышленности, – для перемещения кода приложений между разными средами требуется значительный объем ручного труда. Подобно тому, как универсальные контейнеры упрощают перевозку грузов, контейнеры Docker упрощают перемещение (перенос) программных приложений. Разработчики могут полностью сосредоточиться на создании приложения, на проведении цикла тестирования и на вводе приложения в эксплуатацию, не беспокоясь о различиях в программных средах и обеспечении необходимых зависимостей. Инженеры по эксплуатации могут уделить все внимание специфическим вопросам обеспечения работы контейнеров, таким как распределение ресурсов, запуск и остановка контейнеров, перемещение контейнеров между серверами.

Краткая история Docker

В 2008 году Соломон Хайкс (Solomon Hykes) основал компанию dotCloud для реализации облачной технологии «платформа как услуга» (Platform-as-a-Service – PaaS), полностью независимой от какого-либо языка программирования. Решение вопроса независимости от языка являлось наиболее значимой отличительной характеристикой компании dotCloud, поскольку существующие «платформы как услуги» были привязаны к конкретным наборам языков (например, платформа Heroku поддерживала Ruby, платформа Google App Engine поддерживала Java и Python). В 2010 году компания dotCloud приняла участие в разработке программы Y Combinator accelerator, в результате чего установила связи с новыми

[1] LAMP – Linux, Apache, MySQL, PHP – основные компоненты любого веб-приложения.

партнерами и начала привлекать серьезных инвесторов. Самое важное событие произошло в марте 2013 года, когда dotCloud открыла исходные коды Docker, ключевого компонента, ядра программного сервиса dotCloud. В то время как некоторые компании опасались разглашения своих «чудо-секретов», компания dotCloud поняла, что Docker принесет гораздо больше пользы, если станет открытым проектом, управляемым свободным сообществом.

Ранние версии Docker представляли собой немного усовершенствованную обертку механизма LXC в сочетании с файловой системой с каскадно-объединенным монтированием, но при этом ввод в эксплуатацию и скорость разработки были потрясающе быстрыми. В течение шести месяцев проект заработал более 6700 звезд на сайте GitHub и привлек 175 добровольных участников. Это привело к изменению названия компании с dotCloud на Docker Inc. и к смене ориентации бизнес-модели. Спустя 15 месяцев после релиза 0.1, в июне 2014 года была представлена версия Docker 1.0. В этой версии было продемонстрировано значительное улучшение стабильности и надежности, и она была официально объявлена «готовой к промышленной эксплуатации», хотя до этого уже использовалась в полной мере некоторыми компаниями, в том числе Spotify и Baidu. В то же время началась работа по превращению из простого механизма контейнеров Docker в полноценную платформу, сопровождаемая запуском открытого репозитория для контейнеров Docker Hub.

Другие компании быстро оценили потенциальные возможности Docker. В сентябре 2013 года основным партнером становится компания Red Hat, которая начала использовать Docker в качестве движка своего облачного продукта OpenShift. Вскоре после этого Google, Amazon и DigitalOcean обеспечили поддержку Docker в своих облачных сервисах, а некоторые стартапы стали специализироваться на создании Docker-хостов, например компания StackDock. В октябре 2014 года корпорация Microsoft объявила о полной поддержке Docker в будущих версиях Windows Server, что свидетельствовало о существенном изменении позиционирования компании, ранее знаменитой своим массивным, «неповоротливым» корпоративным программным обеспечением.

На конференции DockerConEU в декабре 2014 года был представлен Docker Swarm, менеджер кластеров для Docker и Docker Machine, инструмент командной строки для поддержки работы Docker-хостов. Это стало очевидным подтверждением намерения компании Docker создать полное комплексное решение для обеспечения работы контейнеров, не ограничиваясь при этом одним лишь механизмом Docker.

В декабре того же года проект CoreOS объявил о разработке rkt, собственного механизма поддержки контейнеров, а также о разработке спецификации контейнеров appc. В июне 2015 года на конференции DockerCon в Сан-Франциско Соломон Хайкс (Docker) и Алекс Полви (Alex Polvi) (CoreOS) объявили о создании организации Open Container Initiative (OCI) (позже переименованной в Open Container Project) для разработки общего стандарта формата контейнеров и механизмов запуска.

Также в июне 2015 года проект FreeBSD заявил о поддержке Docker в ОС FreeBSD с использованием файловой системы ZFS и механизма обеспечения совместимости с ОС Linux. В августе 2015 года Docker и Microsoft совместно выпустили «предварительный технический обзор» реализации Docker Engine для Windows Server.

При выпуске версии Docker 1.8 компания Docker представила новую функцию управления надежностью содержимого контейнера, которая проверяет целостность Docker-образа и подлинность авторства стороны, опубликовавшей этот образ. Управление надежностью содержимого является важнейшим компонентом для создания проверенных рабочих процессов на основе образов, загружаемых из реестров Docker.

Дополнительные модули и надстройки

Docker Inc., как и любая другая компания, всегда тонко чувствовала, что своим успехом она обязана всей экосистеме в целом. Компания Docker Inc. сконцентрировала внимание на создании стабильной, готовой к промышленной эксплуатации версии механизма контейнеров, в то время как другие компании, например CoreOS, WeaveWorks и ClusterHQ, работали в смежных областях, таких как оркестрация и сетевая связанность контейнеров. Но сразу стало понятно, что Docker Inc. планирует представить полностью завершенную платформу «из коробки», включающую все возможности поддержки сетевой среды, хранения данных и организации работы в целом. Для поощрения непрерывного развития экосистемы и обеспечения доступа пользователей к решениям из широкого набора вариантов использования компания Docker Inc. заявила, что намерена создать модульную расширяемую программную среду для Docker, в которой все компоненты могут быть заменены на равноценные компоненты от независимых производителей («третьих сторон») или расширены с помощью функциональных компонентов независимых производителей. Компания Docker Inc. охарактеризовала эту философию фразой «батарейки включены в комплект, но их можно полностью заменить», означающей, что должно быть представлено полностью завершенное решение, но отдельные его части могут быть заменены[1].

На момент написания данной книги инфраструктура дополнительных подключаемых модулей уже доступна, хотя и находится на самой ранней стадии своего развития. Существует несколько дополнительных модулей для обеспечения сетевой связанности контейнеров и управления данными.

Кроме того, компания Docker следует так называемому «Манифесту наращивания функциональности инфраструктуры» («Infrastructure Plumbing Manifesto»),

---

[1] Автору никогда не нравилась эта фраза, поскольку все батарейки предоставляют в основном одну и ту же функциональность, но могут быть заменены только батарейками того же типоразмера и с одинаковой величиной напряжения. Автор полагает, что приведенная фраза происходит от философской концепции языка Python «все батарейки включены в комплект» как констатации того факта, что все стандартные библиотеки расширений всегда включены в дистрибутивный комплект Python.

в котором придается особое значение обязательному повторному использованию компонентов и улучшению их существующей инфраструктуры, когда есть возможность для этого, а также предоставление сообществу усовершенствованных повторно используемых компонентов при возникновении потребности в новых инструментах. Следствием этого является выделение исходного кода низкого уровня для поддержки работы контейнеров в проект runC, управляемый OCI, и этот код можно многократно использовать как основу для других контейнерных платформ.

64-битовая версия ОС Linux

Во время написания данной книги единственной стабильной и готовой к промышленному использованию платформой для Docker можно назвать только 64-битовую версию ОС Linux. Это означает, что для работы Docker необходимо установить на компьютер дистрибутив 64-битовой версии Linux, и все создаваемые контейнеры также будут образами 64-битовой версии этой ОС. Если вы работаете с ОС Windows или Mac OS, то можете запустить Docker в виртуальной машине. Поддержка «родных» контейнеров для других платформ, включая BSD, Solaris и Windows Server, находится на различных стадиях разработки. Поскольку Docker сам по себе не обеспечивает реализацию любого типа виртуализации, контейнеры всегда должны соответствовать ядру хоста – контейнер на Windows Server может работать только на хосте под управлением ОС Windows Server, 64-битовый Linux-контейнер работает только на хосте с установленной 64-битовой версией ОС Linux.

Микросервисы и «монолиты»

Одним из наиболее часто встречающихся вариантов использования контейнеров, в наибольшей степени способствующим их широкому распространению, являются микросервисы (microservices).

Микросервисы представляют собой такой способ разработки и компоновки программных систем, при котором они формируются из небольших независимых компонентов, взаимодействующих друг с другом через сеть. Такая методика полностью противоположна традиционному «монолитному» способу разработки программного обеспечения, где создается одна большая программа, обычно написанная на C++ или на Java.

При необходимости масштабирования «монолитной» программы выбор, как правило, ограничен только вариантом вертикального масштабирования (scale up), и растущие потребности удовлетворяются использованием более мощного компьютера с большим объемом оперативной памяти и более производительным процессором. В противоположность такому подходу микросервисы предназначены для горизонтального масштабирования (scale out), когда рост потребностей удовлетворяется добавлением нескольких серверов с распределением нагрузки между ними. В архитектуре микросервисов возможно регулирование только тех ресурсов, которые требуются для конкретного сервиса, то есть можно сосредо-

точиться лишь на узких проблемных местах в системе. В «монолитной» системе масштабируется либо все, либо ничего, в результате ресурсы используются крайне нерационально.

С точки зрения сложности микросервисы подобны обоюдоострому клинку. Каждый отдельный микросервис должен быть простым для понимания и модификации. Но в системе, сформированной из десятков или даже сотен таких микросервисов, общая сложность возрастает из-за многочисленных взаимодействий между отдельными компонентами.

Простота и высокая скорость работы контейнеров позволяют считать их наиболее подходящими компонентами для реализации архитектуры микросервисов. По сравнению с виртуальными машинами, контейнеры намного меньше по размерам и гораздо быстрее развертываются, что позволяет использовать в архитектурах микросервисов минимум ресурсов и быстро реагировать на требуемые изменения.

Чтобы узнать больше о микросервисах, прочтите книги Building Microservices (англ.) Сэма Ньюмена (Sam Newman) и Microservice Resource Guide (англ.) Мартина Фаулера (Martin Fowler). Русский перевод вышел в издательстве «Питер» в 2016 году.

Глава 2

Установка

В этой главе кратко описываются действия, необходимые для установки Docker. Здесь рассматриваются некоторые тонкости, зависящие от используемой операционной системы, но при некоторой доле везения установка должна проходить без затруднений. Если у вас уже установлена одна из последних версий Docker (1.8 или более новая), можете сразу перейти к чтению следующей главы.

Установка Docker в ОС Linux

Наилучшим способом установки Docker в ОС Linux является использование установочного скрипта, предоставляемого компанией Docker. Несмотря на то что большинство основных дистрибутивов Linux формируют собственные пакеты, зачастую они отстают от выпусков новых версий Docker, что становится существенной проблемой, учитывая темп развития Docker.

Требования при установке Docker
Docker не выдвигает много требований, но для его работы необходима достаточно новая версия ядра (3.10 или более поздняя во время написания книги). Проверить версию ядра можно командой `uname -r`. Если вы используете дистрибутив RHEL или CentOS, то необходима версия дистрибутива 7 или более новая. Кроме того, следует помнить, что для работы нужна поддержка 64-битовой архитектуры. Это можно проверить командой `uname -m` – результат должен быть таким: `x86_64`.

Для автоматической установки Docker нужна возможность использования скрипта, предоставленного на сайте https://get.docker.com. Официальные инструкции советуют просто выполнить команду `curl -sSL | sh` или команду `wget -q0- | sh`, и вы можете поступить именно так, но я рекомендую изучить этот скрипт перед запуском, чтобы знать, какие изменения будут внесены в вашу систему:

```
$ curl https://get.docker.com > /tmp/install.sh
$ cat /tmp/install.sh
...
$ chmod +x /tmp/install.sh
$ /tmp/install.sh
...
```

Скрипт выполнит несколько проверок, затем установит Docker, используя пакет, подходящий для вашей системы. Кроме того, при необходимости будут установлены некоторые дополнительные пакеты, от которых зависит Docker, для обеспечения безопасности и некоторых возможностей файловой системы.

Если вы просто не хотите пользоваться скриптом установки или вам нужна версия Docker, отличающаяся от предлагаемой в скрипте, то можно скачать бинарный пакет с веб-сайта Docker. Недостатком этого способа является отсутствие проверок зависимостей, и вам придется устанавливать или обновлять их вручную. Для получения более подробной информации и ссылок на бинарные пакеты обратитесь к странице https://docs.docker.com/installation/binaries/.

Проверено на версии Docker 1.8
Во время написания этой книги текущей стабильной версией была Docker 1.8. Все команды были протестированы именно на этой версии.

Запуск SELinux в разрешающем режиме

Если вы пользуетесь дистрибутивом на основе Red Hat, например RHEL, CentOS или Fedora, то, вероятнее всего, в системе установлен модуль обеспечения безопасности SELinux.

Перед началом работы Docker рекомендуется перевести SELinux в разрешающий режим (permissive mode), в котором ошибки регистрируются в системном журнале, но никакие ограничения не применяются. Если для SELinux установлен усиленный режим (enforcing mode), то при попытках выполнения примеров из этой книги вы, вероятнее всего, увидите загадочные сообщения об ошибках «Доступ запрещен» («Permission Denied»).

Для проверки режима работы SELinux в вашей системе запустите команду sestatus и изучите выведенную информацию. Например, вывод может быть следующим:

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   enforcing ❶
Mode from config file:          error (Success)
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
```

❶ Если в этой строке вы видите «enforcing», то SELinux активен, и правила обеспечения безопасности применяются принудительно.

Для перевода SELinux в разрешающий режим выполните команду `sudo setenforce 0`.

Чтобы узнать больше о SELinux и понять, почему необходимо установить доверительный режим для использования Docker, см. раздел «SELinux» в главе 13.

Запуск Docker без sudo

Поскольку по умолчанию для запуска бинарного файла Docker требуются привилегии суперпользователя, необходимо вводить все команды с префиксом `sudo`. Это быстро надоедает. Ситуацию можно исправить, добавив нужного пользователя в группу `docker`. В Ubuntu это делается так:

```
$ sudo usermod -aG docker
```

Здесь будет создана группа `docker`, если она не существовала ранее, и в эту группу добавляется текущий пользователь. После этого нет необходимости многократно входить в режим суперпользователя и выходить из него. В других дистрибутивах Linux эта операция выполняется похожим образом.

Кроме того, потребуется перезапустить сервис Docker, но в разных дистрибутивах для этого служат различные команды. Например, в Ubuntu это будет выглядеть следующим образом:

```
$ sudo service docker restart
```

В дальнейшем для краткости префикс `sudo` не будет указываться для всех команд Docker.

Добавление какого-либо пользователя в группу `docker` равносильно предоставлению ему прав суперпользователя `root`. Вы должны помнить о влиянии этого действия на безопасность, особенно если пользуетесь компьютером совместно с другими людьми. Более подробную информацию можно получить на странице https://docs.docker.com/articles/security/.

Установка Docker в Mac OS или в ОС Windows

Если вы работаете в ОС Windows или в Mac OS, то для запуска Docker потребуется одна из форм виртуализации[1]. Можно скачать и установить какую-либо виртуальную машину в полной комплектации и следовать инструкциям для ОС Linux, чтобы установить Docker, или установить пакет Docker Toolbox (https://www.docker.com/toolbox), в который включена минимальная виртуальная машина boot2docker, а также другие инструменты, использование которых описывается далее в этой книге, например Compose и Swarm. Если в Mac OS для установки приложений вы пользуетесь Homebrew, то можно найти инструкцию для установки boot2docker, тем не менее я рекомендую выполнить стандартную установку Docker Toolbox, чтобы избежать проблем.

---

[1] Windows и Docker недавно объявили о совместной инициативе по поддержке Docker в ОС Windows Server. Это позволит пользователям Windows Server запускать образы программ Windows без виртуализации.

После установки Docker Toolbox вы можете сразу начать работу с Docker, открыв терминал быстрого доступа[1]. В качестве другого способа можно предложить конфигурирование существующего терминала с помощью следующих команд:

```
$ docker-machine start default
Starting VM...
Started machines may have new IP addresses. You may need to rerun the
'docker-machine env' command.
(Запуск ВМ...
Виртуальные машины могут получить новые IP-адреса. Может потребоваться
повторное выполнение команды 'docker-machine env'.)
$ eval $(docker-machine env default)
```

Эти команды настраивают вашу операционную среду, добавляя в нее параметры, необходимые для доступа к механизму Docker, работающему в виртуальной машине.

При работе с Docker Toolbox всегда необходимо помнить о следующих фактах:

- в примерах данной книги предполагается, что Docker работает на хост-компьютере. При использовании Docker Toolbox это не обязательно. В некоторых случаях может потребоваться изменение ссылки localhost на реальный IP-адрес виртуальной машины. Например:

  ```
  $ curl localhost:5000
  ```

  нужно будет изменить следующим образом:

  ```
  $ curl 192.168.59.103:5000
  ```

  IP-адрес работающей виртуальной машины можно без затруднений определить командой `docker-machine ip default` и даже немного автоматизировать эту процедуру:

  ```
  $ curl $(docker-machine ip default):5000
  ```

- тома (разделы диска), для которых установлено соответствие между локальной ОС и контейнером Docker, обязательно должны быть кросс-смонтированы (cross-mounted) внутри виртуальной машины. Docker Toolbox автоматизирует эту операцию до определенной степени, но о необходимости кросс-монтирования следует помнить, если возникают проблемы при использовании томов (разделов) в Docker;

---

[1] В пакет Docker Toolbox также включена программа Kitematic, обеспечивающая графический пользовательский интерфейс для запуска контейнеров Docker. В этой книге мы не рассматриваем Kitematic, но эта программа несомненно заслуживает внимательного изучения, особенно теми, кто только начинает работу с Docker.

- при наличии особых требований можно изменять настройки виртуальной машины. Для виртуальной машины boot2docker существует файл */var/lib/boot2docker/profile* с разнообразными параметрами, включающий и конфигурацию механизма Docker. Также можно автоматически запускать собственные скрипты после инициализации виртуальной машины, отредактировав файл */var/lib/boot2docker/bootlocal.sh*. Более подробные инструкции можно найти в документации https://github.com/boot2docker/boot2docker.

Если при выполнении примеров из книги возникают проблемы, то можно попытаться напрямую подключиться на виртуальную машину с помощью команды `docker-machine ssh default` и выполнять команды из ее среды.

Экспериментальный канал Docker

Кроме стабильной рабочей версии, Docker поддерживает экспериментальную версию с целью тестирования новейших функциональных возможностей. Поскольку эти функциональные возможности продолжают находиться в стадии обсуждения и разработки, они с большой вероятностью могут существенно изменяться до момента их включения в стабильную версию. Экспериментальную версию можно использовать только для изучения новых возможностей, не вошедших в основной релиз, но не для промышленного применения.

В ОС Linux экспериментальную версию можно установить с помощью специального скрипта:

```
$ curl -sSL https://experimental.docker.com/ | sh
```

или загрузить бинарную версию с сайта http://bit.ly/1Q8g39C. Следует отметить, что экспериментальная версия обновляется раз в сутки, а для проверки корректности загруженной версии имеются хэш-коды.

Оперативная проверка

Чтобы убедиться в правильности установки и работоспособности программной среды Docker, следует выполнить команду `docker version`. Эта команда должна выдать информацию, похожую на приведенную ниже:

```
$ docker version
Client:
 Version:      1.8.1
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   d12ea79
 Built:        Thu Aug 13 02:35:49 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.8.1
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   d12ea79
 Built:        Thu Aug 13 02:35:49 UTC 2015
 OS/Arch:      linux/amd64
```

Если выведенный результат похож на приведенный выше, то установка завершена успешно, и вы готовы к чтению следующей главы. Если выведено следующее (или подобное) предупреждение:

```
$ docker version
Client:
 Version:      1.8.1
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   d12ea79
 Built:        Thu Aug 13 02:35:49 UTC 2015
 OS/Arch:      linux/amd64
Get http:///var/run/docker.sock/v1.20/version: dial unix /var/run/docker.sock: no such file
or directory.
(Попытка http:///var/run/docker.sock/v1.20/version: dial unix /var/run/docker.sock: нет такого файла или каталога.)
* Are you trying to connect to a TLS-enabled daemon without TLS?
(* Вы пытаетесь установить соединение с демоном, основанным на TLS, без работающего TLS?[1])
* Is your docker daemon up and running?
(* Демон docker запущен и работает?)
```

это означает, что демон Docker не работает (или клиент не получил доступа к нему). Для изучения возникшей проблемы попробуйте запустить демон Docker вручную командой `sudo docker daemon` – это должно дать вам дополнительную информацию о том, что пошло не так, и помочь в поиске решения. (Следует отметить, что этот способ работает только на Linux-хосте. Если вы используете Docker Toolbox или подобный инструментальный пакет, то за помощью обращайтесь к соответствующей документации.)

[1] TLS – протокол Transport Layer Security, обеспечивающий безопасность при аутентификации.

Глава 3

Первые шаги

Эта глава позволит вам сделать первые шаги в практическом применении Docker. Мы начнем с процедуры запуска, затем используем несколько простых контейнеров, чтобы понять, как работает Docker. Потом перейдем к рассмотрению *Dockerfiles* – основных строительных блоков для контейнеров Docker – и реестров *Docker Registries*, обеспечивающих поддержку распределения контейнеров. Глава завершается обзором способов использования контейнеров для запуска хранилища ключ-значение с сохранением данных.

Запуск первого образа

Для проверки правильности установки программной системы Docker выполните следующую команду:

```
$ docker run debian echo "Hello World"
```

Выполнение может занять некоторое время, в зависимости от скорости вашего интернет-соединения, но в итоге вы должны увидеть результат, похожий на приведенный ниже:

```
Unable to find image 'debian' locally
(Невозможно найти образ 'debian' на локальной системе)
debian:latest: The image you are pulling has been verified
(debian:latest: Загружаемый образ проверен)
511136ea3c5a: Pull complete
(511136ea3c5a: Загрузка завершена)
638fd9704285: Pull complete
61f7f4f722fb: Pull complete
Status: Downloaded newer image for debian:latest
(Состояние: Загружен обновленный образ для debian:latest)
Hello World
```

Возникает естественный вопрос: что все это означает? Мы выполнили команду `docker run`, инициализирующую запуск контейнеров. Аргумент `debian` – это имя образа[1], который мы намерены использовать, в данном случае – упрощенная вер-

[1] Определение образа (image) будет дано позже во всех подробностях; пока можно считать, что это «шаблоны» для контейнеров.

сия дистрибутива Debian Linux. Первая строка вывода сообщает об отсутствии локальной копии образа Debian. Затем Docker в онлайн-режиме проверяет Docker Hub и загружает самую новую версию образа Debian. После загрузки и проверки образа Docker помещает его в работающий контейнер и выполняет заданную команду `echo "Hello World"` внутри контейнера. Результат выполнения этой команды показан в последней выведенной строке.

Если выполнить эту же команду еще раз, то контейнер запустится немедленно, без предварительной загрузки образа. Выполнение команды должно занять около секунды, и это удивительно, если учесть объем необходимой работы: Docker подготовил и запустил контейнер, выполнил команду `echo`, затем остановил контейнер. Если попытаться сделать нечто подобное в обычной виртуальной машине, то придется ждать несколько секунд, а может быть, даже несколько минут.

С помощью Docker можно запустить командную оболочку shell внутри контейнера, выполнив следующую команду:

```
$ docker run -i -t debian /bin/bash
root@622ac5689680:/# echo "Hello from Container-land!"
Hello from Container-land!
root@622ac5689680:/# exit
exit
```

Здесь мы получаем новый промпт командной строки внутри контейнера, что очень похоже на вход с помощью сервиса ssh в командную оболочку на удаленном компьютере. Флаги `-i` и `-t` сообщают Docker, что необходимо создать сеанс интерактивной работы на подключаемом терминальном устройстве *tty*. Команда `/bin/bash` инициализирует командную оболочку bash (Bourne again shell). При выходе из командной оболочки контейнер прекратит работу – контейнеры работают, пока существует их основной процесс.

Основные команды

Теперь займемся более подробным исследованием Docker, запуская контейнер и наблюдая в нем результаты выполнения различных команд и действий. Сначала инициализируем новый контейнер, но в этот раз зададим для него имя хоста с помощью флага `-h`:

```
$ docker run -h CONTAINER -i -t debian /bin/bash
root@CONTAINER:/#
```

А что будет, если «сломать» контейнер?

```
root@CONTAINER:/# mv /bin /basket
root@CONTAINER:/# ls
bash: ls: command not found (команда не найдена)
```

Мы переместили (фактически переименовали) каталог */bin* и сделали контейнер почти бесполезным, по крайней мере временно[1]. Прежде чем покинуть «сломанный» контейнер, посмотрим, что нам могут сказать о нем команды `ps`, `inspect` и `diff`. Откройте новый терминал (не завершая работу первого контейнера) и выполните команду `docker ps` прямо с хоста. Результат должен быть похож на приведенный ниже:

```
CONTAINER ID   IMAGE    COMMAND       ...   NAMES
00723499fdbf   debian   "/bin/bash"   ...   stupefied_turing
```

Это позволяет нам узнать некоторые подробности обо всех контейнерах, работающих в текущий момент. Большинство столбцов вывода понятно само по себе, но следует отметить, что Docker присвоил контейнеру удобное для чтения имя, которое может служить идентификатором контейнера на данном хосте, в нашем случае это имя `"stupefied_turing"`[2]. Больше информации о конкретном контейнере можно получить с помощью команды `docker inspect` с указанием имени или идентификатора нужного контейнера:

```
$ docker inspect stupefied_turing
[
{
    "Id": "00723499fdbfe55c14565dc53d61452519deac72e18a8a6fd7b371ccb75f1d91",
    "Created": "2015-09-14T09:47:20.2064793Z",
    "Path": "/bin/bash",
    "Args": [],
    "State": {
        "Running": true,
...
```

Здесь выводится огромное количество важной информации, но изучать ее достаточно трудно. Можно воспользоваться утилитой `grep` или аргументом `--format` (который принимает шаблон языка Go[3]) для выделения той информации, которая нас интересует. Например:

```
$ docker inspect stupefied_turing | grep IPAddress
        "IPAddress": "172.17.0.4",
        "SecondaryIPAddress": null,
```

---

[1] На презентациях я обычно использую команду `rm`, но мысль о том, что кто-то может выполнить команду `rm` на своем рабочем хосте, заставляет меня воспользоваться командой `mv` в данной книге.

[2] Docker генерирует имена в виде случайного прилагательного, за которым следует имя известного ученого, инженера или хакера. Вы можете задать любое имя с помощью аргумента `--name` (например, `docker run --name boris debian echo "Boo"`).

[3] По аналогии с механизмом шаблонов для языка программирования Go. Это полнофункциональный механизм обработки шаблонов, обеспечивающий гибкость и мощь при фильтрации и выборке данных. Более подробную информацию о практическом применении этого механизма можно найти по адресу https://docs.docker.com/reference/commandline/inspect/.

```
$ docker inspect --format {{.NetworkSettings.IPAddress}} stupefied_turing
172.17.0.4
```

Обе команды выдают IP-адрес работающего контейнера. А тепепрь рассмотрим еще одну команду – `docker diff`:

```
$ docker diff stupefied_turing
C /.wh..wh.plnk
A /.wh..wh.plnk/101.715484
D /bin
A /basket
A /basket/bash
A /basket/cat
A /basket/chacl
A /basket/chgrp
A /basket/chmod
...
```

Здесь мы видим список файлов, измененных в работающем контейнере; в данном случае это удаление каталога `/bin` и добавление его содержимого в каталог `/basket`, а также создание каталога и файла, связанных с драйвером подсистемы хранения данных. Для контейнеров Docker использует файловую систему UnionFS (Union File System), которая позволяет монтировать несколько файловых систем в общую иерархию, которая выглядит как единая файловая система. Файловая система конкретного образа смонтирована как уровень только для чтения, а любые изменения в работающем контейнере происходят на уровне с разрешенной записью, монтируемого поверх основной файловой системы образа. Поэтому Docker при поиске изменений в работающей системе должен рассматривать только самый верхний уровень, на котором возможна запись.

Последняя команда, заслуживающая внимания при первом знакомстве с контейнерами, – `docker logs`. Если выполнить эту команду с указанием имени контейнера, то будет выведен список всех событий, произошедших внутри заданного контейнера:

```
$ docker logs stupefied_turing
root@CONTAINER:/# mv /bin /basket
root@CONTAINER:/# ls
bash: ls: command not found
```

К настоящему моменту все действия с контейнером завершены, поэтому можно закончить его работу. Сначала выйдем из командной оболочки:

```
root@CONTAINER:/# exit
exit
$
```

После этой команды завершится работа и самого контейнера, так как командная оболочка была его единственным активным процессом. Если выполнить команду `docker ps`, то мы не увидим ни одного работающего контейнера.

Но это еще не все. После выполнения команды `docker ps -a` выводится список всех контейнеров, включая остановленные (stopped) (формально их называют «контейнерами, из которых был совершен выход» (exited containers)). Такие контейнеры могут быть перезапущены командой `docker start` (хотя в нашем случае контейнер перезапустить не удастся, так как все стандартные пути в нем некорректны). Чтобы окончательно избавиться от контейнера, следует воспользоваться командой `docker rm`:

```
$ docker rm stupefied_turing
stupefied_turing
```

Удаление остановленных контейнеров

Если необходимо удалить все остановленные контейнеры, можно использовать результат выполнения команды `docker ps -aq -f status=exited`, которая выводит идентификаторы всех остановленных контейнеров. Например:

```
$ docker rm -v $(docker ps -aq -f status=exited)
```

Поскольку эта операция повторяется достаточно часто, можно поместить ее в скрипт командной оболочки или присвоить ее псевдоним (alias). Обратите внимание на то, что аргумент `-v` позволяет удалить все тома (разделы), управляемые Docker, на которые не ссылаются какие-либо другие контейнеры.

Избежать перезапуска ранее остановленных контейнеров можно, включив в команду `docker run` флаг `--rm`, который позволяет удалить остановленный контейнер и созданную на время его существования соответствующую файловую систему.

Теперь попробуем разобраться, как создать новый, действительно полезный контейнер, заслуживающий сохранения для дальнейшего использования[1]. Мы создадим приложение cowsay, работающее внутри Docker. Тем, кто не знает, что такое cowsay, рекомендую предварительно познакомиться с этим приложением. Начнем с запуска контейнера и установки нескольких пакетов:

```
$ docker run -it --name cowsay --hostname cowsay debian bash
root@cowsay:/# apt-get update
...
Reading package lists... Done
(Чтение списков пакетов... Выполнено)
root@cowsay:/# apt-get install -y cowsay fortune
...
root@cowsay:/#
```

Приступаем к работе:

```
root@cowsay:/# /usr/games/fortune | /usr/games/cowsay
 ______________________________________
/ Writing is easy; all you do is sit  \
| staring at the blank sheet of paper |
```

[1] Хотя этот контейнер и назван «полезным», это не вполне соответствует действительности.

```
| until drops of blood form on your   |
| forehead.                           |
| (Писать легко, нужно всего лишь     |
| сидеть и смотреть на чистый лист    |
| бумаги, пока на лбу не выступят     |
| капли крови.)                       |
|                                     |
| -- Gene Fowler                      |
\ (Джин Фаулер)                       /
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Превосходно. Сохраним этот контейнер[1]. Для превращения контейнера в образ нужно всего лишь выполнить команду `docker commit`. При этом не имеет значения, работает контейнер или он остановлен. Необходимо передать в команду имя контейнера («cowsay»), имя для создаваемого образа («cowsayimage»), а также имя репозитория, в котором образ будет сохранен («test»):

```
root@cowsay:/# exit
exit
$ docker commit cowsay test/cowsayimage
d1795abbc71e14db39d24628ab335c58b0b45458060d1973af7acf113a0ce61d
```

Возвращаемое командой значение представляет собой уникальный идентификатор созданного образа. Теперь у нас есть образ с предустановленным приложением cowsay, который мы можем запускать в любое время:

```
$ docker run test/cowsayimage /usr/games/cowsay "Moo"
 _____
< Moo >
 ------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Великолепно. Но имеются и некоторые затруднения. Если нужно что-то изменить, то нам придется вручную повторить все ранее проделанные операции. Например, если потребуется использовать другой основной образ ОС, то мы будем вынуждены снова выполнить процедуру с самого начала. И, что более важно, такая процедура не является легко воспроизводимой – повторение набора действий для создания образа связано с трудностями и является потенциальным источни-

[1] Только для начальных ознакомительных примеров. Так будет проще.

ком ошибок. Решение этой проблемы заключается в использовании специального файла *Dockerfile* для создания образа и автоматизированного повторного воспроизведения этой процедуры.

Создание образов из файлов Dockerfile

Dockerfile – это обычный текстовый файл, содержащий набор операций, которые могут быть использованы для создания Docker-образа. Сначала создадим новый каталог и собственно Dockerfile для нашего примера:

```
$ mkdir cowsay
$ cd cowsay
$ touch Dockerfile
```

Затем в созданный *Dockerfile* добавим следующее содержимое:

```
FROM debian:wheezy

RUN apt-get update && apt-get install -y cowsay fortune
```

Инструкция `FROM` определяет базовый образ ОС (это, как и ранее, `debian`, но сейчас мы уточнили, что необходимо воспользоваться конкретной версией «wheezy»). Инструкция `FROM` является строго обязательной для всех файлов Dockerfile как самая первая незакомментированная инструкция. Инструкции `RUN` определяют команды, выполняемые в командной оболочке внутри данного образа. В нашем случае это команда установки пакетов cowsay и fortune, которая ранее была выполнена вручную.

Теперь мы можем создать образ, выполнив команду `docker build` в том же каталоге, где расположен наш Dockerfile:

```
$ ls
Dockerfile
$ docker build -t test/cowsay-dockerfile .
Sending build context to Docker daemon  2.048 kB
(Передача контекста создания в демон Docker)
Step 0 : FROM debian:wheezy
(Шаг 0 :)
 ---> f6fab3b798be
Step 1 : RUN apt-get update && apt-get install -y cowsay fortune
(Шаг 1 :)
 ---> Running in 29c7bd4b0adc
( ---> Запуск в 29c7bd4b0adc)
...
Setting up cowsay (3.03+dfsg1-4) ...
(Настройка cowsay (3.03+dfsg1-4) ...)
 ---> dd66dc5a99bd
Removing intermediate container 29c7bd4b0adc
(Удаление вспомогательного контейнера 29c7bd4b0adc)
Successfully built dd66dc5a99bd
(Успешное создание dd66dc5a99bd)
```

После этого мы можем запускать образ точно таким же способом, как раньше:

```
$ docker run test/cowsay-dockerfile /usr/games/cowsay "Moo"
```

Образы, контейнеры и файловая система Union File System

Чтобы понять взаимосвязь между образами и контейнерами, необходимо более подробно рассмотреть ключевой элемент технологии, лежащей в основе Docker, – UFS (иногда используется термин «каскадно-объединенное монтирование» (union mount)[1]). Файловые системы с каскадно-объединенным монтированием позволяют подключать несколько файловых систем с перекрытием (или наложением друг на друга), причем для пользователя они будут выглядеть как одна файловая система. Каталоги могут содержать файлы из нескольких файловых систем, но если двум файлам в точности соответствует один и тот же путь, то файл, смонтированный самым последним, скроет все ранее монтированные файлы. Docker поддерживает несколько различных реализаций UnionFS, включая AUFS, Overlay, devicemapper, BTRFS и ZFS. Реализацию, используемую в конкретной системе, можно определить командой `docker info` – смотреть содержимое заголовка «Storage Driver». Файловую систему можно заменить, но это рекомендуется только в тех случаях, когда вы точно знаете, что делаете, и хорошо знакомы со всеми преимуществами и недостатками используемых файловых систем.

Образы Docker состоят из нескольких уровней (layers). Каждый уровень представляет собой защищенную от записи файловую систему. Для каждой инструкции в Dockerfile создается свой уровень, который размещается поверх предыдущих уровней. Во время преобразования образа в контейнер (командой `docker run` или `docker create`) механизм Docker выбирает нужный образ и добавляет на самом верхнем уровне файловую систему с возможностью записи (одновременно с этим инициализируются разнообразные параметры настройки, такие как IP-адрес, имя, идентификатор и ограничения ресурсов).

Поскольку ненужные уровни значительно увеличивают размеры образов (а для файловой системы AUFS установлен строгий лимит, равный 127 уровням), во многих файлах Dockerfile можно обнаружить попытку свести к минимуму количество уровней посредством записи нескольких команд Unix в одной инструкции `RUN`.

Контейнер может находиться в одном из следующих состояний: «создан» (created), «перезапуск» (restarting), «активен» или «работает» (running), «приостановлен» (paused) или «остановлен» (exited). «Созданным» считается контейнер, который был инициализирован командой `docker create`, но его работа пока еще не началась. Состояние exited в общем случае соответствует состоянию «остановлен» (stopped), когда в данном контейнере нет активно выполняющихся процессов (их нет и в «созданном» контейнере, но остановленный контейнер уже запускался, по крайней мере один раз). Контейнер существует, пока существует его основной процесс. Остановленный контейнер можно перезапустить командой `docker start`. Остановленный контейнер – это не то же самое, что исходный образ. Остановленный контейнер сохраняет все изменения в его параметрах настройки, метаданных

[1] Вообще говоря, аббревиатура UFS во многих случаях используется для обозначения Unix File System. Файловую систему с каскадно-объединенным монтированием чаще обозначают как UnionFS.

и файловой системе, в том числе и параметры конфигурации времени выполнения, например IP-адрес, которые не хранятся в образах. Состояние перезапуска на практике встречается редко и возникает в тех случаях, когда механизм Docker пытается повторно запустить контейнер после неудачной первой попытки.

Выполнение этой задачи можно немного упростить для обычного пользователя, воспользовавшись преимуществами инструкции для файлов Dockerfile `ENTRYPOINT`. Эта инструкция позволяет определить выполняемый файл, который будет вызываться для обработки любых аргументов, переданных в команду `docker run`.

Добавим в конец нашего Dockerfile следующую строку:

```
ENTRYPOINT ["/usr/games/cowsay"]
```

Теперь нужно заново создать образ, после чего его можно запускать без указания команды cowsay:

```
$ docker build -t test/cowsay-dockerfile .
...
$ docker run test/cowsay-dockerfile "Moo"
...
```

Запуск стал проще. Но при этом мы лишились возможности использования внутри контейнера команды fortune в качестве генератора потока ввода для программы cowsay. Это можно исправить, если написать специальный скрипт для инструкции `ENTRYPOINT` – обычное дело при создании Dockerfile. В том же каталоге, где расположен наш Dockerfile, создадим файл с именем *entrypoint.sh* и введем в него следующее содержимое[1]:

```
#!/bin/bash
if [ $# -eq 0 ]; then
    /usr/games/fortune | /usr/games/cowsay
  else
    /usr/games/cowsay "$@"
fi
```

После сохранения необходимо сделать этот файл выполняемым при помощи команды `chmod +x entrypoint.sh`.

При вызове без аргументов скрипт создает программный канал для передачи потока вывода программы fortune в поток ввода программы cowsay, в противном случае вызывается программа cowsay с передачей ей заданных аргументов. Теперь необходимо изменить Dockerfile: добавить только что созданный файл в образ и вызвать скрипт в инструкции `ENTRYPOINT`. Отредактированный Dockerfile должен выглядеть так:

[1] При написании скриптов для `ENTRYPOINT` будьте внимательны, чтобы не запутать пользователей, – помните, что этот скрипт будет заглатывать целиком все команды, передаваемые в `docker run`, а это может привести к непредсказуемым результатам.

```
FROM debian

RUN apt-get update && apt-get install -y cowsay fortune
COPY entrypoint.sh /  ❶

ENTRYPOINT ["/entrtypoint.sh"]
```

❶ Инструкция COPY просто копирует файл из файловой системы хоста в файловую систему образа, где первый аргумент определяет файл хоста, а второй – целевой путь, таким образом, эта инструкция очень похожа на обычную команду cp.

Попробуем создать новый образ и запустить контейнеры с аргументами и без них:

```
$ docker build -t test/cowsay-dockerfile .
...(вывод пропущен для краткости)...
$ docker run test/cowsay-dockerfile
 _____________________________________
/ The last thing one knows in         \
| constructing a work is what to put  |
| first.                              |
| (Лишь в конце работы мы обычно узнаем,  |
| с чего ее нужно было начать.)       |
\ -- Blaise Pascal (Блез Паскаль)     /
 -------------------------------------
        \
         \   ^__^
          \  (oo)\_______
             (__)\       )\/\
                 ||----w |
                 ||     ||
$ docker run test/cowsay-dockerfile Hello Moo
 _____________
< Hello Moo >
 -------------
        \
         \   ^__^
          \  (oo)\_______
             (__)\       )\/\
                 ||----w |
                 ||     ||
```

Работа с реестрами

После того как мы создали нечто выдающееся, хочется поделиться своим творением со всеми, но как? При первом запуске образа Debian в начале главы сам образ загружался из официального реестра Docker Hub. Аналогичным образом можно выгружать свои образы в реестр Docker Hub, чтобы другие пользователи скачивали и использовали их.

Доступ к реестру Docker Hub можно получить как из командной строки, так и через сайт. Поиск существующих образов выполняется с помощью специальной команды поиска Docker или непосредственно на сайте http://registry.hub.docker.com.

Реестры, репозитории, образы и теги

Для хранения образов применяется иерархическая система. При этом используются следующие термины:

- реестр (registry) – сервис, отвечающий за хранение и распространение образов. По умолчанию используется реестр Docker Hub;
- репозиторий (repository) – набор взаимосвязанных образов (обычно представляющих различные версии одного приложения или сервиса);
- тег (tag) – алфавитно-цифровой идентификатор, присваиваемый образам внутри репозитория (например, 14.04 или stable).

Таким образом, команда `docker pull amouat/revealjs:latest` загрузит образ с тегом `latest` в репозиторий `amouat/revealjs` из реестра Docker Hub.

Для выгрузки в реестр нашего образа cowsay необходимо зарегистрироваться (создать учетную запись) в реестре Docker Hub (в онлайн-режиме на сайте или с помощью команды `docker login`). После этого достаточно связать образ с соответствующим репозиторием и выполнить команду `docker push` для выгрузки помеченного образа в Docker Hub. Но сначала добавим в Dockerfile инструкцию `MAINTAINER`, которая просто определяет информацию, позволяющую связаться с автором данного образа:

```
FROM debian

MAINTAINER John Smith <john@smith.com>
RUN apt-get update && apt-get install -y cowsay fortune
COPY entrypoint.sh /

ENTRYPOINT ["/entrypoint.sh"]
```

Теперь заново сгенерируем образ и выгрузим его в Docker Hub. В этот раз нужно будет использовать имя репозитория, начинающееся с вашего пользовательского имени в Docker Hub (в моем случае это `amouat`), за которым следуют слеш (/) и имя выгружаемого образа. Например:

```
$ docker build -t amouat/cowsay .
...
$ docker push amouat/cowsay
The push refers to a repository [docker.io/amouat/cowsay] (len: 1)
(Команда push выполняется для репозитория [docker.io/amouat/cowsay] (длина: 1))
e8728c722290: Image successfully pushed (Образ выгружен успешно)
5427ac510fe6: Image successfully pushed
4a63ead8b301: Image successfully pushed
73805e6e9ac7: Image successfully pushed
c90d655b99b2: Image successfully pushed
```

```
30d39e59ffe2: Image successfully pushed
511136ea3c5a: Image successfully pushed
latest: digest: sha256:bfd17b7c5977520211cecb202ad73c3ca14acde6878d9ffc81d95...
```

Так как я не указал тег после имени репозитория, образу был автоматически присвоен тег `latest`. Чтобы определить свой тег, необходимо добавить его после имени репозитория и символа двоеточия (например, `docker build -t amouat/cowsay:stable`).

После успешного завершения выгрузки любой пользователь может загрузить этот образ командой `docker pull amouat/cowsay`.

Закрытые частные репозитории

Возможно, вы не согласны предоставить доступ к своему образу любому желающему. В этом случае можно выбрать один из двух вариантов. Воспользуйтесь платным защищенным репозиторием (на сайте Docker Hub или аналогичным сервисом, например quay.io) или создайте собственный реестр. В главе 7 более подробно рассматриваются закрытые частные репозитории и реестры.

Пространства имен для образов

Выгружаемые Docker-образы могут принадлежать одному из трех пространств имен, определяемых по имени самого образа:

- имена, начинающиеся с текстовой строки и слеша (/), такие как `amouat/revealjs`, принадлежат пространству имен «user». В репозитории Docker Hub это образы, выгруженные конкретным пользователем. Например, упомянутый выше `amouat/revealjs` – это образ `revealjs`, выгруженный пользователем `amouat`. Можно свободно и бесплатно выгружать общедоступные образы в репозиторий Docker Hub, который уже содержит тысячи образов, от эксцентричного `supertest2014/nyan` до весьма полезного `gliderlabs/logspout`;
- имена, подобные `debian` и `ubuntu`, без префиксов и слешей, принадлежат пространству имен «root», управляемому компанией Docker Inc. и зарезервированному для официальных образов широко распространенных программ и дистрибутивов, доступных для загрузки с Docker Hub. Несмотря на контроль со стороны компании Docker, эти образы сопровождаются в основном третьими сторонами, обычно авторами и поставщиками соответствующего ПО (например, образ `nginx` сопровождается компанией nginx). Для большинства широко распространенных пакетов ПО здесь представлены официальные образы, на которые следует обратить внимание в первую очередь при поиске контейнеров для производственных целей;
- имена с префиксами в виде имени хоста или IP-адреса представляют образы, хранящиеся в сторонних реестрах (не в Docker Hub). Это могут быть собственные реестры различных организаций, а также реестры конкурентов Hub, таких как quay.io. Например, имя `localhost:5000/wordpress` определяет образ WordPress, расположенный в локальном реестре.

Такое распределение по пространствам имен помогает пользователю не запутаться при определении происхождения различных образов: используя образ `debian`, вы уверены, что это официальный образ из реестра Docker Hub, а не какая-то случайная версия из другого реестра.

Использование официального образа Redis

Вынужден признать, что от созданного ранее образа cowsay польза невелика. Рассмотрим, как можно использовать образ из какого-либо официального репозитория Docker, для конкретного примера возьмем официальный образ для Redis, широко распространенной системы хранения данных в виде пар «ключ-значение».

Официальные репозитории

Если вы ищете в реестре Docker Hub общеизвестное приложение или сервис, например язык программирования Java или СУБД PostgreSQL, то обнаружите сотни результатов[1]. Официальные репозитории Docker предназначены для предоставления управляемых образов с гарантированным качеством и известным происхождением, поэтому по возможности их нужно рассматривать в первую очередь. Они должны возвращаться в верхней части списка результатов поиска и помечаться как официальные.

При извлечении из официального репозитория имя образа не содержит части, указывающей на пользователя, или эта часть обозначена как `library` (например, репозиторий СУБД MongoDB доступен и как `mongo`, и как `library/mongo`). Кроме того, будет выведено сообщение «The image you are pulling has been verified» («Загружаемый вами образ проверен»), подтверждающее, что демон Docker проверил контрольные суммы для данного образа и его происхождение не вызывает сомнений.

Начнем с загрузки образа:

```
$ docker pull redis
Using default tag: latest
(Используется тег по умолчанию: latest)
latest: Pulling from library/redis

d990a769a35e: Pull complete (Загрузка завершена)
8656a511ce9c: Pull complete
f7022ac152fb: Pull complete
8e84d9ce7554: Pull complete
c9e5dd2a9302: Pull complete
27b967cdd519: Pull complete
3024bf5093a1: Pull complete
e6a9eb403efb: Pull complete
c3532a4c89bc: Pull complete
35fc08946add: Pull complete
d586de7d17cd: Pull complete
1f677d77a8fa: Pull complete
ed09b32b8ab1: Pull complete
54647d88bc19: Pull complete
2f2578ff984f: Pull complete
ba249489d0b6: Already exists (Уже существует)
19de96c112fc: Already exists
library/redis:latest: The image you are pulling has been verified.
(library/redis:latest: Загружаемый вами образ проверен)
```

[1] Во время написания книги существовало 1350 образов PostgreSQL.

```
Important: image verification is a tech preview feature and should not be re...
(Важно: проверка образа является предварительной технической функцией и не должна...)
Digest: sha256:3c3e4a25690f9f82a2a1ec6d4f577dc2c81563c1ccd52efdf4903ccdd26cada3
Status: Downloaded newer image for redis:latest
(Состояние: загружен более новый образ для redis:latest)
```

Запустим контейнер Redis, но при этом используем аргумент `-d`:

```
$ docker run --name myredis -d redis
585b3d36e7cec8d06f768f6eb199a29feb8b2e562288445263377216969 5b94a
```

Аргумент `-d` сообщает Docker, что контейнер нужно запустить в фоновом режиме. Docker начинает работу контейнера как обычно, но вместо вывода результатов из контейнера возвращает только его идентификатор и выполняет выход. Контейнер продолжает работу в фоновом режиме, а чтобы увидеть вывод результатов из контейнера, можно воспользоваться командой `docker logs`.

Но как мы будем работать с этим контейнером? Очевидно, что необходимо каким-то образом установить связь с базой данных. Подходящего приложения у нас нет, поэтому используем инструмент командной строки `redis-cli`. Его можно установить прямо на хост, но проще, а главное – полезнее в учебных целях создать новый контейнер для запуска в нем `redis-cli` и установить соединение между двумя контейнерами:

```
$ docker run --rm -it --link myredis:redis redis /bin/bash
root@ca38735c5747:/data# redis-cli -h redis -p 6379
redis:6379> ping
PONG
redis:6379> set "abc" 123
OK
redis:6379> get "abc"
"123"
redis:6379> exit
root@ca38735c5747:/data# exit
exit
```

Итак, мы только что установили соединение между двумя контейнерами и добавили данные в СУБД Redis буквально за несколько секунд. Но каким образом была проделана эта работа?

Будущие изменения в сетевой среде Docker

В этой главе и во всех последующих главах книги для установления соединения между контейнерами в сети используется команда `--link`. В будущем планируются изменения способов организации сетевой работы Docker, вероятно, вместо установления соединения (link) между контейнерами будет использоваться более характерный для современной сетевой среды метод «публикации сервисов» («publish services»). Тем не менее в ближайшем будущем будет поддерживаться метод установления соединений, и для корректного выполнения примеров из данной книги не потребуются какие-либо изменения.

Для получения более подробной информации о грядущих изменениях в сетевой среде см. раздел «Новая сетевая среда Docker» (глава 11).

Установление соединения определяется аргументом `--link myredis:redis` в команде `docker run`. Docker получает информацию о том, что нам нужно установить соединение между новым контейнером и существующим контейнером myredis, и в новом контейнере ссылка на существующий должна быть обозначена именем redis. Для этого Docker создает в файле нового контейнера */etc/hosts* запись redis, указывающую на IP-адрес контейнера myredis. Это позволяет нам пользоваться именем хоста redis непосредственно в командной строке `redis-cli`, без дополнительного определения пути к нему или поиска IP-адреса контейнера Redis.

После этого выполняется команда СУБД Redis `ping` для проверки правильности установленного соединения с Redis-сервером перед добавлением и извлечением каких-либо данных с помощью команд `set` и `put`.

Все выглядит замечательно, но остался один вопрос: как сохранить наши данные и создать их резервную копию? Для этого не следует использовать стандартную файловую систему контейнера, необходима возможность простого совместного использования хранилища данных контейнером и хостом или другими контейнерами. Docker предоставляет такую возможность посредством реализации концепции томов. *Тома* (*volumes*) – это файлы или каталоги, которые смонтированы непосредственно на хосте и не являются частью каскадно-объединенной файловой системы. Это означает, что другие контейнеры могут совместно использовать их, и все изменения будут сразу же фиксироваться в файловой системе хоста. Существуют два способа объявления каталога как тома: использование инструкции `VOLUME` в Dockerfile или включение флага `-v` в команду `docker run`. Ниже приведены примеры реализации обоих способов с одинаковым результатом – определение каталога */data* как тома внутри контейнера:

```
VOLUME /data
```

или

```
$ docker run -v /data test/webserver
```

По умолчанию заданный каталог или файл будет смонтирован на хосте внутри каталога, в котором был установлен Docker (обычно это каталог */var/lib/docker/*). В качестве точки монтирования можно определить любой другой каталог хоста в команде `docker run` (например, `docker run -d -v /host/dir:/container/dir test/webserver`). В файле Dockerfile определить каталог хоста как точку монтирования невозможно по причинам, связанным с обеспечением переносимости и безопасности (заданный файл или каталог может отсутствовать на других системах, а контейнерам нельзя предоставлять возможность монтирования критически важных файлов, подобных */etc/passwd*, без явно определенных прав доступа к ним).

Но как применить все описанное выше для создания резервных копий содержимого контейнера Redis? Ниже показан один способ, предполагается, что контейнер `myredis` продолжает работу:

```
$ docker run --rm -it --link myredis:redis redis /bin/bash
root@a1c4abf81f:/data# redis-cli -h redis -p 6379
```

```
redis:6379> set "persistence" "test"
OK
redis:6379> save
OK
redis:6379> exit
root@a1c4abf81f:/data# exit
exit
$ docker run --rm --volumes-from myredis -v $(pwd)/backup:/backup \
        debian cp /data/dump.rdb /backup/
$ ls backup
dump.rdb
```

Обратите внимание на то, что аргумент `-v` использован для монтирования известного нам существующего каталога хоста, а аргумент `--volumes-from` – для установления соединения между новым контейнером и каталогом базы данных Redis.

После завершения работы с контейнером `myredis` можно остановить и удалить его:

```
$ docker stop myredis
myredis
$ docker rm -v myredis
myredis
```

Все оставшиеся вспомогательные контейнеры можно удалить с помощью команды:

```
$ docker rm $(docker ps -aq)
45e404caa093
e4b31d0550cd
7a24491027fc
...
```

Резюме

На этом описание основ, позволяющих начать работу с Docker, завершено. Описание было кратким и не слишком подробным, но после чтения этой главы вы должны чувствовать себя более уверенно при создании собственных контейнеров и в процессе работы с ними. В следующей главе мы более подробно рассмотрим архитектуру Docker и некоторые важные теоретические концепции, лежащие в ее основе.

Глава 4

Основы Docker

В этой главе мы подробно рассмотрим основополагающие концепции Docker. Начнем с описания общей архитектуры Docker, включая технологии, на которых основано его функционирование. Далее последуют разделы с подробным разбором процесса создания Docker-образов, сетевых контейнеров и обработки данных, хранящихся на томах. Завершается глава обзором команд Docker, с которыми мы еще не встречались.

Поскольку в этой главе содержится большой объем справочного материала, вы можете уделить внимание только основным важным пунктам и сразу перейти к чтению главы 5, при необходимости возвращаясь к данной главе как к справочнику.

Архитектура Docker

Чтобы понять, как наиболее эффективно использовать Docker и некоторые не вполне очевидные его свойства, необходимо хотя бы в целом представлять себе, каким образом организована совместная работа компонентов платформы, скрытых от пользователя.

На рис. 4.1 показаны главные компоненты установленной и готовой к использованию платформы Docker:

- в центре расположен *демон Docker* (*Docker daemon*), ответственный за создание, запуск и контроль работы контейнеров, а также за создание и хранение образов. Контейнеры и образы представлены в правой части диаграммы. Демон Docker запускается командой `docker daemon`, обычно об этом заботится операционная система хоста;
- клиент Docker, размещенный в левой части диаграммы, используется для диалога с демоном Docker по протоколу HTTP. По умолчанию это соединение устанавливается через сокет домена Unix, но также может использоваться TCP-сокет для поддержки соединений с удаленными клиентами или дескриптор файла для сокетов, управляемых systemd. Так как все операции обмена данными выполняются по протоколу HTTP, можно без затруднений организовать соединение с удаленными демонами Docker и разработать привязки (bindings) к нужному языку программирования, но при этом

следует учитывать особенности реализации этих возможностей, например обязательное наличие *контекста создания* (*building context*), описанного в соответствующем разделе данной книги. Интерфейсы прикладного программирования, используемые для организации обмена данными с демоном, четко определены и подробно документированы, что позволяет разработчикам писать программы, взаимодействующие напрямую с демоном, без использования клиента Docker. Клиент и демон Docker распространяются как отдельные независимые бинарные файлы;

- реестры Docker используются для хранения и распространения образов. Реестром, выбираемым по умолчанию, является Docker Hub, на котором хранятся тысячи общедоступных образов, а также управляемые «официальные» образы. Многие организации создают собственные реестры, которые используются для хранения коммерческих и приватных образов и для устранения накладных расходов, связанных с загрузкой образов через Интернет. В разделе «Создание собственного реестра» содержится более подробная информация об организации и сопровождении нового частного реестра. Демон Docker загружает образы из реестров по запросу `docker pull`. Кроме того, он выполняет автоматическую загрузку образов, указанных в запросе `docker run` и в инструкции `FROM` файла Dockerfile, если эти образы недоступны на локальной системе.

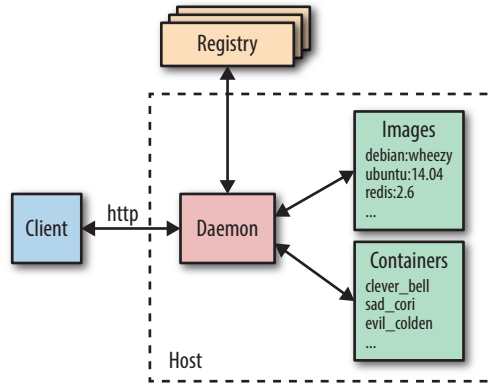

Рис. 4.1. Общая схема взаимодействия главных компонентов Docker

Базовые технологии

Демон Docker использует *«драйвер выполнения»* (*execution driver*) для создания контейнеров. По умолчанию выбирается собственный драйвер Docker *runc*, но, кроме того, обеспечивается поддержка более старого драйвера для механизма LXC. Драйвер runc очень тесно связан со следующими механизмами ядра:

- *cgroups* – механизм, отвечающий за управление ресурсами, используемыми контейнером (процессор, оперативная память и т. д.). Механизм cgroups так-

же обеспечивает выполнение операций *«замораживания»* (*freezing*) и *«размораживания»* (*unfreezing*) контейнеров как поддержку функциональности команды `docker pause`;

- *пространства имен* (*namespaces*) отвечают за изоляцию контейнеров, гарантируют, что файловая система, имя хоста, пользователи, сетевая среда и процессы любого контейнера полностью отделены от остальной части системы.

Libcontainer также поддерживает *SElinux* и *AppArmor*, которые можно использовать для создания более строгой системы безопасности. Более подробную информацию об этом можно получить в главе 13.

Еще одной основополагающей технологией для Docker является *файловая система с каскадно-объединенным монтированием* (*Union File System – UnionFS*), обеспечивающая хранение уровней для контейнеров. Функциональность UnionFS обеспечивается одним из нескольких драйверов файловой системы: AUFS, devicemapper, BTRFS или Overlay. С кратким описанием UnionFS можно ознакомиться в примечании «Образы, контейнеры и файловая система Union File System» (глава 3).

Сопровождающие технологии

Сами по себе механизм Docker и реестр Docker Hub не предоставляют завершенного полноценного решения для работы с контейнерами. Для большинства пользователей потребуются сервисы поддержки и вспомогательное ПО, например система управления кластерами, инструменты обнаружения сервисов, расширенные сетевые функциональные возможности и проч. В разделе «Дополнительные модули и надстройки» (глава 1) было отмечено, что Docker Inc. планирует создание полноценного решения «из коробки», включающего все требуемые возможности, но позволяющего пользователям без труда заменять компоненты, установленные по умолчанию, на компоненты сторонних производителей. Стратегия «заменяемых батареек» в первую очередь реализуется на уровне интерфейсов прикладного программирования, позволяя подключать компоненты непосредственно к движку Docker, но ее также можно наблюдать на примере формирования дистрибутивных пакетов Docker как независимых автономных бинарных файлов, которые с легкостью заменяются аналогами от третьих сторон.

На текущий момент можно привести следующий список технологий поддержки, предоставляемых Docker:

- *Swarm* – решение задачи кластеризации от Docker. Swarm позволяет сгруппировать несколько Docker-хостов, после чего пользователь может работать с этой группой как с единым ресурсом. В главе 12 эта технология описана более подробно;
- *Docker Compose* – инструмент для создания и выполнения приложений, скомпонованных из нескольких Docker-контейнеров. Такие компоновки используются главным образом при разработке и тестировании, но гораздо реже в производственной среде. В разделе «Автоматизация с помощью Compose» можно найти более подробную информацию;

- *Docker Machine* устанавливает и конфигурирует Docker-хосты на локальных и удаленных ресурсах. Кроме того, Machine конфигурирует клиента Docker, упрощая процедуру переключения между средами. Пример использования Machine приведен в главе 9;
- *Kitematic* представляет собой графический пользовательский интерфейс для операционных систем Mac OS и Windows, обеспечивающий запуск и управление контейнеров Docker;
- *Docker Trusted Registry* – локально устанавливаемое программное решение для хранения и управления образами Docker. В действительности это локальная версия реестра Docker Hub, которую можно объединить с существующей инфраструктурой обеспечения безопасности и согласовать с правилами хранения и обеспечения защиты данных, принятых в конкретной организации. Все функциональные возможности локального реестра, в том числе различные метрики, *управление доступом на основе ролей* (*Role-based access control – RBAC*) и регистрационные журналы, контролируются через административную консоль. В настоящий момент это единственный программный продукт Docker Inc., исходный код которого закрыт.

Список сервисов и приложений третьих сторон, которые основаны на Docker или используют эту платформу, уже сейчас достаточно велик. Некоторые полноценные решения доступны в следующих областях:

- *сетевая среда* – создание сетей контейнеров, распределенных между разными хостами, представляет собой непростую задачу, которую можно решить разнообразными способами. Недавно в этой области появилось несколько решений, например Weave (http://weave.works/net/) и Project Calico (http://www.projectcalico.org/). Кроме того, Docker намерен в ближайшем будущем представить комплексное сетевое решение под названием Overlay. Пользователи смогут заменить драйвер Overlay на любые другие решения, используя подключаемую программную рабочую среду для работы в сети;
- *обнаружение сервисов* – сразу после появления контейнеров Docker потребовался способ поиска других сервисов для взаимодействия с ними. Обычно такие сервисы также работают в контейнерах. Поскольку IP-адреса присваиваются контейнерам динамически, задача обнаружения сервисов в больших системах достаточно сложна. К решениям в этой области относятся Consul (https://consul.io/), Registrator (https://github.com/gliderlabs/registrator), SkyDNS (https://github.com/skynetservices/skydns/) и etcd (https://github.com/coreos/etcd);
- оркестровка и управление кластером – при развертывании большого количества контейнеров весьма важно наличие инструментов для контроля и управления всей системой в целом. Каждый новый контейнер должен быть размещен на некотором хосте, его нужно контролировать и обновлять. Система должна правильно реагировать на сбои или изменения нагрузки, перемещая, запуская или останавливая контейнеры соответствующим образом. Здесь можно отметить несколько конкурирующих решений:

Kubernetes (http://kubernetes.io) от Google, Marathon (https://github.com/mesosphere/marathon); фреймворк для Mesos (https://mesos.apache.org), Fleet (https://github.com/coreos/fleet) от CoreOS, а также собственный инструмент Docker Swarm.

Более подробно все эти темы будут рассматриваться в части III книги. Следует отметить, что наряду с Docker Trusted Registry также существуют альтернативные решения, такие как Enterprise Registry (https://coreos.com/products/enterprise-registry/) от CoreOS и Artifactory (http://www.jfrog.com/open-source/#os-arti) от JFrog.

В дополнение к упомянутым выше подключаемым сетевым драйверам Docker поддерживает еще и *подключаемые тома* (*volume plugins*) для интеграции с другими системами хранения данных. Здесь особого внимания заслуживают Flocker (https://github.com/ClusterHQ/flocker), инструмент для управления данными и их перемещения в многохостовой системе, и GlusterFS (https://github.com/calavera/docker-volume-glusterfs) для распределенного хранения данных. Более подробную информацию о фреймворке подключаемых драйверов можно найти на сайте https://docs.docker.com/extend/plugins/.

Широкое распространение контейнеров привело к любопытному побочному эффекту, заключающемуся в появлении нового поколения операционных систем, специализированных для поддержки контейнеров. Несмотря на то что Docker успешно работает на большинстве современных дистрибутивов Linux, таких как Ubuntu, Red Hat и других, появились проекты, направленные на создание облегченных и простых в сопровождении дистрибутивов, главной задачей которых является только обеспечение работы контейнеров (или контейнеров и виртуальных машин), что особенно важно для расширения функциональных возможностей центра обработки данных или кластера. В качестве примеров можно привести Project Atomic (http://www.projectatomic.io/), CoreOS (https://coreos.com/) и RancherOS (http://rancher.com/rancher-os/).

Хостинг для Docker

Более подробно мы рассмотрим хостинг для Docker в главе 9, но сейчас среди многочисленных вариантов выбора следует отметить некоторые наиболее значимые. Многие известные облачные провайдеры, в том числе Amazon, Google и Digital Ocean, уже предлагают определенный уровень поддержки Docker. Google Container Engine, возможно, является самым интересным вариантом, так как создан непосредственно на основе Kubernetes. Разумеется, даже если облачный провайдер не предлагает прямую поддержку Docker, обычно имеется возможность предоставления виртуальных машин, в которых можно запускать Docker-контейнеры.

В этой области также работает компания Joyent, предлагающая собственный механизм контейнеров под названием Triton на основе SmartOS. С помощью реализации интерфейсов прикладного программирования Docker в своем механизме контейнеров и технологии эмуляции Linux компания Joyent смогла создать общедоступный облачный сервис, взаимодействующий со стандартным Docker-клиентом. Более того, компания Joyent уверена, что ее реализация контейнера об-

ладает высоким уровнем безопасности, позволяющим работать непосредственно с аппаратным обеспечением, без необходимости размещения в виртуальной машине, а это может означать более высокую эффективность и существенное сокращение накладных расходов, особенно с точки зрения операций ввода/вывода.

Есть еще несколько проектов, организовавших PaaS-платформу на основе Docker, – Deis (http://deis.io/), Flynn (https://flynn.io/) и Paz (http://paz.sh).

Как создаются образы

В разделе «Создание образов из файлов Dockerfile» (глава 3) мы рассматривали основной способ создания новых образов с помощью файлов Dockerfile и команды `docker build`. В этом разделе будет более подробно описано все происходящее при создании образа, а в конце раздела приведено краткое руководство по различным инструкциям, используемым в Dockerfile. Всегда полезно понимать внутреннее функционирование команды создания, так как ее поведение иногда может становиться неожиданным.

Контекст создания образа

Для команды `docker build` необходим Dockerfile и *контекст создания образа* (*build context*) (который может быть пустым). Контекст создания – это набор локальных файлов и каталогов, к которым можно обращаться из инструкций `ADD` и/или `COPY` в Dockerfile и которые обычно определяются как путь к нужному каталогу. Например, в разделе «Создание образов из файлов Dockerfile» (глава 3) мы использовали команду создания образа `docker build -t test/cowsay-dockerfile .`, которая определяла контекст создания как `'.'`, то есть текущий рабочий каталог. Все файлы и каталоги, расположенные по указанному пути, формируют контекст создания образа и передаются в демон Docker как часть процесса создания.

В тех случаях, когда контекст не определен, – если задан только URL для Dockerfile или содержимое Dockerfile передается по программному каналу из стандартного потока ввода (`STDIN`), – контекст создания данного образа считается пустым.

Не следует использовать / в качестве контекста создания образа
Поскольку контекст создания образа полностью включается в tar-архив и передается в демон Docker, не используйте для этой цели каталога, в котором содержится большое количество файлов. Например, если для контекста создания вы возьмете /home/user, Загрузки (Downloads) или /, то в результате получите длительную задержку, пока Docker-клиент будет упаковывать все файлы заданного каталога и передавать их в демон.

Если задан URL, начинающийся с *http* или *https*, то предполагается, что это прямая ссылка на Dockerfile. Маловероятно, что это окажется полезным, так как с файлом Dockerfile не связан какой-либо контекст (а ссылки на архивы неприемлемы).

В качестве контекста создания образа разрешается указывать git-репозиторий. В этом случае клиент Docker создает клон такого репозитория и всех подчиненных модулей во временном каталоге, который затем передается в демон Docker

как контекст создания образа. Docker воспринимает контекст как git-репозиторий, если переданный путь начинается с префиксов *github.com/*, *git@* или *git://*. Вообще говоря, я рекомендую избегать применения такого метода, вместо этого лучше перечислять репозитории вручную – это более гибкий способ, снижающий вероятность возникновения беспорядка.

Кроме того, клиент Docker способен принимать входные данные из стандартного потока ввода `STDIN`, если в команде указать аргумент `"-"` вместо контекста создания образа. Входными данными может быть либо Dockerfile без контекста (например, `docker build - < Dockerfile`), либо архивный файл, содержащий контекст, в том числе и Dockerfile (например, `docker build - < context.tar.gz`). Архивные файлы могут передаваться в формате *tar.gz*, *xz* или *bzip2*.

Размещение файла Dockerfile внутри контекста может быть указано с помощью аргумента `-f` (например, `docker build -f dockerfiles/Dockerfile.debug .`). Если нет прямого указания, то Docker попытается найти файл с именем *Dockerfile* в корневом каталоге контекста.

Использование файла .dockerignore

Для удаления ненужных файлов из контекста создания образа можно воспользоваться файлом *.dockerignore*. Этот файл должен содержать имена исключаемых файлов, разделенных символами перехода на новую строку. Допускаются символы шаблонов `*` и `?`. Например, можно сформировать `.dockerignore` со следующим содержимым:

```
.git ❶
*/.git ❷
*/*/.git ❸
*.sw? ❹
```

❶ Из контекста исключается файл или каталог *.git* в корневом каталоге контекста создания образа, но при этом в контекст включается такой файл в любом подкаталоге (таким образом *.git* исключается, но *dir1/.git1* включается).

❷ Из контекста исключается файл или каталог *.git*, расположенный в подкаталоге одним уровнем ниже корневого каталога (таким образом *dir1/.git* исключается, но *.git* и *dir1/dir2/.git* включаются).

❸ Из контекста исключается файл или каталог *.git*, расположенный в подкаталоге двумя уровнями ниже корневого каталога (таким образом *dir1/dir2/.git* исключается, но *.git* и *dir1/.git* включаются).

❹ Из контекста исключаются файлы *test.swp*, *test.swo* и *bla.swp*, но в контексте остается файл *dir1/test.swp*.

Не поддерживаются более сложные регулярные выражения, такие как `[A-Z]*`. На момент выпуска данной книги не существовал какой-либо способ определения шаблона файлов, охватывающего все подкаталоги (то есть в одном выражении невозможно исключить сразу два файла – */test.tmp* и */dir1/test.tmp*).

Уровни образа

Способ создания образов Docker часто ставит в тупик неопытных пользователей. Каждая инструкция в Dockerfile приводит к появлению нового *уровня* (*layer*) об-

раза, который также может участвовать в запуске контейнера. Новый уровень создается во время запуска контейнера с использованием образа предыдущего уровня при выполнении соответствующей инструкции Dockerfile и с сохранением нового образа. После успешного завершения выполнения инструкции Dockerfile вспомогательный контейнер удаляется, если в команде не был задан аргумент `--rm=false`[1]. Так как результатом выполнения каждой инструкции является создание статического образа – в сущности, это файловая система и некоторые метаданные, – все активные процессы в данной инструкции будут завершены. Поэтому, несмотря на возможность инициализации в инструкции `RUN` долговременных процессов, подобных демонам СУБД и SSH, любые активные процессы прекратят свою работу при обработке следующей инструкции или при запуске контейнера. Если необходим сервис или процесс, запускаемый вместе с контейнером, его следует инициализировать с помощью инструкции `ENTRYPOINT` или `CMD`.

Весь набор уровней, формирующих образ, можно увидеть, выполнив команду `docker history`. Например:

```
$ docker history mongo:latest
IMAGE          CREATED      CREATED BY                                      ...
278372cb22b2   4 days ago   /bin/sh -c #(nop) CMD ["mongod"]
341d04fd3d27   4 days ago   /bin/sh -c #(nop) EXPOSE 27017/tcp
ebd34b5e9c37   4 days ago   /bin/sh -c #(nop) ENTRYPOINT &{["/entrypoint.
f3b2b8cf226c   4 days ago   /bin/sh -c #(nop) COPY file:ef2883b33ed7ba0cc
ba53e9f50f18   4 days ago   /bin/sh -c #(nop) VOLUME [/data/db]
c537910de5cc   4 days ago   /bin/sh -c mkdir -p /data/db && chown -R mong
f48ad436057a   4 days ago   /bin/sh -c set -x
df59596772ab   4 days ago   /bin/sh -c echo "deb http://repo.mongodb.org/
96de83c82d4b   4 days ago   /bin/sh -c #(nop) ENV MONGO_VERSION=3.0.6
0dab801053d9   4 days ago   /bin/sh -c #(nop) ENV MONGO_MAJOR=3.0
5e7b428dddf7   4 days ago   /bin/sh -c apt-key adv --keyserver ha.pool.sk
e81ad85ddfce   4 days ago   /bin/sh -c curl -o /usr/local/bin/gosu -SL "h
7328803ca452   4 days ago   /bin/sh -c gpg --keyserver ha.pool.sks-keyser
ec5be38a3c65   4 days ago   /bin/sh -c apt-get update
430e6598f55b   4 days ago   /bin/sh -c groupadd -r mongodb && useradd -r
19de96c112fc   6 days ago   /bin/sh -c #(nop) CMD ["/bin/bash"]
ba249489d0b6   6 days ago   /bin/sh -c #(nop) ADD file:b908886c97e2b96665
```

Если создание образа завершилось неудачно, то может оказаться полезным запуск уровня, предшествующего ошибке. Например, имеется следующий Dockerfile:

```
FROM busybox:latest

RUN echo "This should work"
RUN /bin/bash -c echo "This won't"
```

[1] Смысл этого аргумента будет подробно объяснен немного позже, при подробном разборе вывода команды `docker build` в примере отладки.

Попробуем создать образ:

```
$ docker build -t echotest .
Sending build context to Docker daemon 2.048 kB
(Передача контекста создания в демон Docker 2048 Кб)
Step 0 : FROM busybox:latest
 ---> 4986bf8c1563
Step 1 : RUN echo "This should work"
 ---> Running in f63045cc086b ❶
(---> Запуск в f63045cc086b)
This should work
(Это должно работать)
 ---> 85b49a851fcc ❷
Removing intermediate container f63045cc086b ❸
(Удаление вспомогательного контейнера f63045cc086b)
Step 2 : RUN /bin/bash -c echo "This won't" («Это не будет работать»)
 ---> Running in e4b31d0550cd
(---> Запуск в e4b31d0550cd)
/bin/sh: /bin/bash: not found
(/bin/sh: файл /bin/bash не найден)
The command '/bin/sh -c /bin/bash -c echo "This won't"' returned a non-zero code: 127
(Команда '/bin/sh -c /bin/bash -c echo "This won't"' вернула ненулевой код: 127)
```

❶ Идентификатор временного *контейнера* Docker, запускаемого для выполнения инструкций `RUN`.
❷ Идентификатор *образа*, созданного из временного контейнера.
❸ Теперь временный контейнер удаляется.

Несмотря на то что в нашем простом примере причина возникновения критической ошибки вполне очевидна, можно запустить образ, созданный из последнего успешно сформированного уровня, для отладки ошибочной инструкции. Обратите особое внимание на использование для этой цели идентификатора последнего *образа* (`85b49a851fcc`), а не идентификатора последнего *контейнера* (`e4b31d0550cd`):

```
$ docker run -it 85b49a851fcc
/ # /bin/bash -c "echo hmm"
/bin/sh: /bin/bash: not found
/ # /bin/sh -c "echo ahh!"
ahh!
/ #
```

Проблема становится гораздо более понятной: в образ `busybox` не включена командная оболочка `bash`.

Кэширование

Для ускорения создания образов Docker выполняет кэширование каждого уровня. Кэширование очень важно для повышения эффективности рабочих операций, но не всегда его применение имеет смысл. Кэширование используется для инструкций при следующих условиях:

- в кэше была обнаружена предыдущая инструкция;
- в кэше имеется уровень, который имеет в точности ту же инструкцию и предшествующий родительский уровень (даже случайные пропуски могут сделать кэш некорректным).

Кроме того, для инструкций `COPY` и `ADD` кэш считается некорректным, если изменилась контрольная сумма или метаданные для любого файла.

Это означает, что для сохраняемых в кэше инструкций `RUN` не гарантируется получение одинакового результата при многократных повторных вызовах этих инструкций. Будьте особенно внимательны, используя вызовы из кэша для загрузки файлов, выполнения команд `apt-get update` и/или клонирования репозиториев исходного кода.

Чтобы запретить кэширование, можно выполнить команду `docker build` с аргументом `--no-cache`. Также можно добавить или изменить инструкцию перед строкой, в которой необходимо запретить кэширование, поэтому иногда в файле Dockerfile встречаются строки, подобные следующей:

```
ENV UPDATED_ON "14:12 17 February 2015"
RUN git clone...
```

Я не рекомендую применять такой способ, так как он может привести в полное замешательство всех, кто будет в дальнейшем использовать данный образ, особенно если дата создания образа отличается от указанной в вышеприведенной строке Dockerfile.

Базовые образы

При создании собственных образов необходимо выбрать один из базовых образов в качестве отправного пункта. Выбор велик, поэтому стоит уделить время на изучение разнообразных достоинств и недостатков каждого базового образа.

В идеальном случае создание нового образа вообще не потребуется – можно просто использовать существующий образ, объединив с ним свои конфигурационные файлы и/или данные. Во многих случаях такой подход применим для широко распространенного прикладного ПО, например для СУБД и веб-серверов, для которых доступны готовые официальные образы. Вообще говоря, гораздо лучше воспользоваться официальным образом, чем пытаться сформировать собственный – вам предлагается успешный результат работы людей, которые обладают солидным опытом организации работы ПО внутри контейнеров. Если официальный образ не подходит для вашей работы по некоторой конкретной причине, то попробуйте сформулировать эту причину как тему для обсуждения в исходном проекте, и наверняка найдутся пользователи, встречавшиеся с подобными проблемами или знающие, как их решить.

Если нужен образ для управления приложением, написанным вами, сначала попытайтесь поискать официальный базовый образ для используемого языка или программной среды (например, Go или Ruby on Rails). Достаточно часто можно воспользоваться различными образами для сборки и для распространения собственного ПО (например, для компиляции и сборки Java-приложения мож-

но использовать образ `java:jdk`, а распространять полученный JAR-файл лучше с помощью более компактного образа `java:jre`, из которого исключены ненужные инструменты компиляции и сборки). Некоторые другие официальные образы (такие как `node`) также представлены специализированными компактными вариантами без инструментальных средств разработки и заголовочных файлов.

Иногда возникает потребность в использовании небольшого, но полноценного дистрибутива Linux. Если действительно необходим предельный минимализм, рекомендуется обратить внимание на образ `alpine` размером всего 5 Мб, при этом содержащий мощный менеджер пакетов для простой установки приложений и инструментальных средств. Если нужен образ с более широкими возможностями, я обычно пользуюсь одним из образов `debian`, которые по размеру намного меньше образов самого популярного дистрибутива `ubuntu`, но пакетная база у них одна и та же. Кроме того, для организаций, постоянно использующих конкретный дистрибутив Linux, с большой вероятностью найдется подходящий Docker-образ. Лучше потратить некоторое время на поиск, чем переходить на новый дистрибутив, с которым сотрудники не знакомы, а системные администраторы не имеют опыта его сопровождения.

В большинстве случаев нет необходимости тратить время на поиски образов наименьшего размера. Помните, что базовые уровни совместно используются различными образами, поэтому если вы уже работаете с образом `ubuntu:14.04` и загружаете из репозитория Hub образ, основанный на `ubuntu`, то фактически скачиваются только изменения, а не весь образ целиком. Тем не менее образы минимальных размеров, несомненно, обладают большим преимуществом, когда требуются быстрое развертывание и простое сопровождение распространения.

Размер образа можно уменьшить до предельной величины и формировать образы только из бинарных файлов. Для этого необходим Dockerfile из специального образа `scratch` (абсолютно пустая файловая система). Бинарные файлы просто копируются в файловую систему создаваемого образа, а соответствующие инструкции `CMD` записываются в Dockerfile. Для бинарных файлов нужно включить в образ все требуемые библиотеки (без динамического связывания) и исключить возможность вызова внешних команд. Также следует помнить, что бинарные файлы должны быть скомпилированы для архитектуры контейнера, которая может отличаться от архитектуры компьютера, на которой будет работать Docker-клиент[1].

---

[1] В действительности можно развить эту концепцию минимализма и далее, вплоть до отказа от Docker и стандартного ядра Linux, если перейти к технологии unikernel. В архитектуре unikernel приложения объединены с ядром, содержащим только те функциональные возможности, которые действительно используются данным приложением, запускаемым непосредственно в гипервизоре. Это позволяет устранить ненужные уровни бинарного кода и неиспользуемые драйверы, в результате получается приложение гораздо меньшего размера, работающее намного быстрее (время загрузки unikernel обычно не превышает секунды, таким образом, приложения могут запускаться как немедленная реакция на запросы пользователей). Чтобы узнать больше об этой технологии, обратитесь к статье Анила Мадхавапедди (Anil Madhavapeddy) и Дэвида Дж.Скотта (David J.Scott) «Unikernels: Rise of the Virtual Library Operating System» (https://queue.acm.org/detail.cfm?id=2566628), а также к материалам сайта MirageOS http://www.openmirage.org/.

Несмотря на большую привлекательность минималистского подхода, следует отметить, что он может привести к созданию сложной ситуации при отладке и в процессе сопровождения – в образе `busybox` не так уж много рабочих инструментов, а при использовании `scratch` вам недоступна даже командная оболочка.

«Реакция синтеза»

Еще одним интересным вариантом основного образа является `phusion/baseimage-docker`. Разработчики Phusion создали этот образ как «ответную реакцию» на официальный образ Ubuntu, в котором по их заявлению отсутствуют некоторые весьма важные сервисы. Среди основных разработчиков Docker нет единого мнения относительно точки зрения авторов Phusion, поэтому продолжаются дискуссии в блогах, IRC-каналах и твиттере. Основные пункты разногласий перечислены ниже:

- необходимость сервиса `init`. С точки зрения Docker каждый контейнер должен запускать только одно приложение и в идеальном случае должен представлять собой единственный процесс. Если имеется единственный процесс, то сервис `init` не нужен. Основной аргумент авторов Phusion: отсутствие сервиса `init` может привести к заполнению контейнера процессами-зомби, то есть процессами, не завершенными корректно их родительскими процессами, или принудительно прерванными в срочном порядке процессом-супервизором. Это правильный довод, но в контейнере процессы-зомби могут возникать только из-за ошибок в коде приложения, и подавляющее большинство пользователей вообще не должно сталкиваться с такой проблемой, но если она возникла, то наилучшим решением будет исправление ошибок в коде;
- работа демона `cron`. Основные образы `ubuntu` и `debian` по умолчанию не запускают демон `cron`, а в образе `phusion` демон работает. Авторы Phusion обосновывают это тем, что многие приложения зависимы от `cron`, поэтому его функционирование крайне необходимо. Точка зрения команды Docker (и я готов с ней согласиться): демон `cron` должен запускаться только в том случае, если приложение действительно зависит от него;
- демон SSH. В большинстве основных образов сервис SSH по умолчанию не устанавливается и не запускается. Обычный способ вызова любой командной оболочки – использование команды `docker exec` (см. раздел «Управление контейнерами» ниже), что позволяет избежать запуска лишнего процесса в каждом создаваемом контейнере. Казалось бы, что авторы Phusion согласны с таким подходом, поэтому запретили запуск демона SSH по умолчанию, но из-за наличия кода демона и соответствующих библиотек его поддержки размер их образа существенно увеличен.

Подводя итог, могу порекомендовать использование основного образа Phusion, если существует особая необходимость запуска в контейнере многочисленных процессов, демонов `cron` и `ssh`. В противном случае лучше воспользоваться основными образами из официальных репозиториев Docker, такими как `ubuntu:14.04` и `debian:wheezy`.

Пересборка образов

Следует отметить, что при запуске команды `docker build` Docker считывает инструкцию `FROM` и пытается скачать заданный образ, если его нет в локальной системе. Если такой образ существует локально, Docker использует его без проверки

доступности новой версии. Это означает, что команда `docker build` не может гарантировать, что вызываемый образ соответствует самой новой версии, поэтому необходимо явно выполнить команду `docker pull` для всех родительских образов или удалить их, чтобы команда `docker build` загрузила самые новые их версии. Это становится особенно важным при обновлениях, касающихся безопасности, широко используемых основных образов, таких как `debian`.

Инструкции Dockerfile

В этом разделе кратко описаны различные инструкции, предназначенные для использования в файлах Dockerfile. Мы не будем углубляться в подробности, отчасти из-за того, что эти инструкции продолжают изменяться и корректироваться, и информация о них может не вполне соответствовать действительности, отчасти потому, что абсолютно полная и точная документация по инструкциям доступна на сайте Docker http://docs.docker.com/reference/builder/. Комментарии в файлах Dockerfile записываются с помощью символа # в самом начале строки.

Формат exec и формат командной оболочки

В некоторых инструкциях (`RUN`, `CMD` и `ENTRYPOINT`) допускается использование как формата командной оболочки, так и формата exec. Формат exec принимает JSON-массив (например, `["executable", "param1", "param2"]`), предполагая, что первый элемент массива является именем выполняемого файла, а остальные элементы представляют параметры, передаваемые при запуске. Формат командной оболочки – строка произвольной формы, передаваемая для интерпретации в `/bin/sh -c`. Используйте формат exec, чтобы избежать случайного искажения строк командной оболочки, или в тех случаях, когда образ не содержит `/bin/sh`.

Для использования в файлах Dockerfile доступны описанные ниже инструкции.

`ADD`

Копирует файлы из контекста создания или из удаленных URL-ссылок в создаваемый образ. Если архивный файл добавляется из локального пути, то он будет автоматически распакован. Так как диапазон функциональности инструкции `ADD` достаточно велик, в общем случае лучше воспользоваться более простой командой `COPY` для копирования файлов и каталогов в локальном контексте создания или инструкциями `RUN` с запуском `curl` или `wget` для загрузки удаленных ресурсов (с сохранением возможности обработки и удаления результатов загрузки в той же самой инструкции).

`CMD`

Запускает заданную инструкцию во время инициализации контейнера. Если была определена инструкция `ENTRYPOINT`, то заданная здесь инструкция будет интерпретироваться как аргумент для `ENTRY POINT` (в этом случае необходимо использовать формат exec). Инструкция `CMD` замещается любыми аргументами, указанными в команде `docker run` после имени образа. В действительности выполняется только самая последняя инструкция `CMD`, а все предыдущие инструкции `CMD` будут отменены (в том числе и содержащиеся в основных образах).

`COPY`

Используется для копирования файлов из контекста создания в образ. Имеет два формата: `COPY источник цель` и `COPY ["источник", "цель"]` – оба копируют файл или каталог из *«источник»* в контексте создания в *«цель»* внутри контейнера. Формат JSON-массива обязателен, если путь содержит пробелы. Можно использовать шаблонные символы для определения нескольких файлов или каталогов. Следует обратить особое внимание на невозможность указания путей *«источника»*, расположенных вне пределов контекста создания (например, нельзя указать для копирования файл *../another_dir/myfile*).

`ENTRYPOINT`

Определяет выполняемый файл (программу) (и аргументы по умолчанию), запускаемый при инициализации контейнера. В эту выполняемую программу передаются как аргументы любые инструкции `CMD` или аргументы команды `docker run`, записанные после имени образа. Инструкции `ENTRYPOINT` часто используются для организации скриптов запуска, которые инициализируют переменные и сервисы перед обработкой всех передаваемых в образ аргументов.

`ENV`

Определяет переменные среды внутри образа. На эти переменные можно ссылаться в последующих инструкциях. Например:

```
...
ENV MY_VERSION 1.3
RUN apt-get install -y mypackage=$MY_VERSION
...
```

Определенные в этой инструкции переменные будут доступными также и внутри образа.

`EXPOSE`

Сообщает механизму Docker о том, что в данном контейнере будет существовать процесс, прослушивающий заданный порт или несколько портов. Механизм Docker использует эту информацию при установлении соединения между контейнерами (см. раздел «Соединение между контейнерами» ниже) или при открытии портов для общего доступа при помощи аргумента `-P` в команде `docker run`. Но сама по себе инструкция `EXPOSE` не оказывает никакого воздействия на сетевую среду.

`FROM`

Определяет основной образ для файла Dockerfile. Все последующие инструкции выполняют операции создания поверх заданного образа. Основной образ определяется в форме `IMAGE:TAG` (например, `debian:wheezy`). При отсутствии тега по умолчанию полагается `latest`, но я настоятельно рекомендую всегда явно указывать тег конкретной версии, чтобы избежать неприятных неожиданностей. Эта инструкция обязательно должна быть самой первой в Dockerfile.

`MAINTAINER`
: Определяет метаданные об авторе «Author» для создаваемого образа в заданной строке. Извлечь эти метаданные можно с помощью команды `docker inspect -f {{.Author}} IMAGE`. Обычно используется для записи имени автора образа и его контактных данных.

`ONBUILD`
: Определяет инструкцию, которая должна выполняться позже, когда данный образ будет использоваться как основной уровень для другого образа. Это может оказаться полезным при обработке данных, добавляемых в образ-потомок (например, это может быть инструкция копирования дополнительного кода из заданного каталога и запуска скрипта сборки, обрабатывающего скопированные данные).

`RUN`
: Запускает заданную инструкцию внутри контейнера и сохраняет результат.

`USER`
: Задает пользователя (по имени или по идентификатору UID) для использования во всех последующих инструкциях `RUN`, `CMD`, `ENTRYPOINT`. Отметим, что идентификаторы UID одинаковы на хосте и в контейнере, но имена пользователей могут присваиваться различным идентификаторам UID, что может приводить к затруднениям при установке прав доступа.

`VOLUME`
: Объявляет заданный файл или каталог как том. Если такой файл или каталог уже существует в образе, то он копируется в том при запуске контейнера. Если задано несколько аргументов, то они интерпретируются как определение нескольких томов. Из соображений обеспечения безопасности и сохранения переносимости нельзя определить каталог хоста как том внутри файла Dockerfile. Более подробно об этом см. раздел «Управление данными с помощью томов и контейнеров данных» ниже.

`WORKDIR`
: Определяет рабочий каталог для всех последующих инструкций `RUN`, `CMD`, `ENTRYPOINT`, `ADD`, `COPY`. Инструкцию можно использовать несколько раз. Допускается указание относительных путей, при этом итоговый путь определяется относительно ранее указанного рабочего каталога `WORKDIR`.

Установление связи контейнеров с внешним миром

Допустим, вы запустили веб-сервер внутри контейнера. Но как обеспечить связь сервера с внешним миром? Ответ прост – открыть нужные порты для общего доступа с помощью аргументов `-p` или `-P` в команде запуска. Такая команда перенаправляет порты хоста в контейнер. Например:

```
$ docker run -d -p 8000:80 nginx
af9038e18360002ef3f3658f16094dadd4928c4b3e88e347c9a746b131db5444
$ curl localhost:8000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Аргумент `-p 8000:80` сообщил механизму Docker о необходимости перенаправления порта 8000 хоста на порт 80 в контейнере. В качестве альтернативы при использовании аргумента `-P` механизм Docker должен автоматически выбрать свободный порт для перенаправления с хоста в контейнер. Например:

```
$ ID=$(docker run -d -P nginx)
$ docker port $ID 80
0.0.0.0:32771
$ curl localhost:32771
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Главное преимущество использования аргумента `-P` заключается в устранении дополнительного уровня ответственности за корректное назначение портов, что особенно важно при наличии нескольких контейнеров с портами, открытыми для общего доступа. Чтобы определить номера портов, назначенные механизмом Docker, можно выполнить команду `docker port`.

Соединение между контейнерами

Соединения (*links*) механизма Docker – простейший способ обеспечения обмена информацией между контейнерами на одном хосте. При использовании принятой по умолчанию сетевой модели Docker обмен данными между контейнерами будет происходить во внутренней сети Docker, то есть все коммуникационные операции останутся невидимыми из сети хоста.

Будущие изменения в сетевой среде Docker

В будущих версиях Docker (вероятно, в версии 1.9 и последующих) основной внутренней сетевой технологией для контейнеров станет «публикация сервисов» вместо установления соединений между контейнерами. Но поддержка установления соединений сохранится в ближайшем будущем, так что все примеры из данной книги должны работать без изменений.

Для получения более подробной информации о грядущих изменениях в сетевой среде см. раздел «Новая сетевая среда Docker» (глава 11).

Соединения инициализируются с помощью аргумента `--link CONTAINER:ALIAS` в команде `docker run`, где `CONTAINER` – имя контейнера-адресата (link container)[1], а `ALIAS` – локальное имя, используемое внутри управляющего контейнера для обращения к контейнеру-адресату.

Кроме того, при использовании соединений Docker внутреннее имя и идентификатор контейнера-адресата будут добавлены в файл */etc/hosts* в управляющем контейнере, что позволит обращаться по этому имени к контейнеру-адресату из управляющего контейнера.

Далее Docker создает в управляющем контейнере набор переменных среды, предназначенных для упрощения диалога с контейнером-адресатом. Например, при создании контейнера Redis и установления соединения с ним:

```
$ docker run -d --name myredis redis
c9148dee046a6fefac48806cd8ec0ce85492b71f25e97aae9a1a75027b1c8423
$ docker run --link myredis:redis debian env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=f015d58d53b5
REDIS_PORT=tcp://172.17.0.22:6379
REDIS_PORT_6379_TCP=tcp://172.17.0.22:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.22
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_NAME=/distracted_rosalind/redis
REDIS_ENV_REDIS_VERSION=3.0.3
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-3.0.3.tar.gz
REDIS_ENV_REDIS_DOWNLOAD_SHA1=0e2d7707327986ae652df717059354b358b83358
HOME=/root
```

Мы можем видеть, что Docker создает переменные среды с префиксом `REDIS_PORT`, содержащие информацию, необходимую для установления соединения с контейнером Redis. Некоторые значения кажутся избыточными, поскольку нужная информация уже содержится в имени переменной. Тем не менее все переменные и их значения в любом случае полезны, хотя бы как своеобразная форма документации.

Кроме того, Docker импортировал несколько переменных среды из контейнера-адресата, их можно отличить по префиксу `REDIS_ENV`. Такое функциональное свойство может быть весьма удобным, но о нем следует помнить, когда вы используете переменные среды для хранения секретной информации, например маркеров прикладных программных интерфейсов или паролей к базам данных.

По умолчанию контейнеры могут обмениваться информацией друг с другом вне зависимости от того, было ли установлено соединение в явной форме. Если нужно запретить эту возможность, воспользуйтесь аргументами `--icc=false` и `--iptables` при запуске демона Docker. В этом случае при установлении соединения Docker

---

[1] Здесь и далее в книге я буду называть контейнер, с которым устанавливается соединение, контейнером-адресатом (link container), а запускаемый с аргументом `--link` контейнер – управляющим контейнером (master container), так как он инициализирует процедуру установления соединения.

будет применять правила Iptables, чтобы разрешить контейнерам обмен информацией через любые порты, которые были объявлены открытыми.

К сожалению, соединения Docker в их текущем состоянии имеют ряд недостатков. Возможно, самым существенным недостатком является их статичность – несмотря на то что при перезапуске контейнеров соединения должны сохраняться, они не обновляются, если контейнер-адресат заменен. Кроме того, контейнер-адресат обязательно должен быть инициализирован раньше управляющего контейнера, то есть двунаправленное соединение установить невозможно.

В главе 11 можно найти более подробное описание организации сетевого взаимодействия контейнеров.

Управление данными с помощью томов и контейнеров данных

Резюмируя сказанное ранее, отметим, что *тома* (*volumes*) Docker – это каталоги[1], которые не являются частью файловой системы UnionFS конкретного контейнера (см. раздел «Образы, контейнеры и файловая система Union File System» главы 3), а представляют собой обычные каталоги в файловой системе хоста, но могут быть *смонтированы как отдельные файловые системы* (*bind mounting*) (см. примечание «Монтирование каталогов как файловых систем» ниже) внутри контейнера.

Существуют три[2] различных способа инициализации томов. Важно хорошо понимать различия между этими способами. Во-первых, можно объявить том при запуске контейнера с помощью флага `-v`:

```
$ docker run -it --name container-test -h CONTAINER -v /data debian /bin/bash
root@CONTAINER:/# ls /data
root@CONTAINER:/#
```

Здесь каталог */data* внутри контейнера станет томом. Любые файлы, которые данный образ сохранил в каталоге */data*, копируются на этот том. Мы можем проверить место расположения данного тома в файловой системе хоста, выполнив команду `docker inspect` на хосте из новой командной оболочки:

```
$ docker inspect -f {{.Mounts}} container-test
[{5cad... /mnt/sda1/var/lib/docker/volumes/5cad.../_data /data local true}]
```

Здесь том `/data/` в контейнере представляет собой просто ссылку на каталог */var/lib/docker/volumes/5cad.../_data* в файловой системе хоста. Чтобы проверить это на практике, можно добавить файл в указанный каталог хоста[3]:

---

[1] Точнее, каталоги и файлы, так как том может быть единственным файлом.

[2] Честно говоря, два с половиной – все зависит от того, как считать.

[3] Если у вас установлено соединение с удаленным демоном Docker, то выполнять следующую команду на удаленном хосте необходимо с помощью SSH. Если вы используете Docker Machine (в том случае, когда Docker был установлен через Docker Toolbox), то можно сделать это командой `docker-machine ssh default`.

```
$ sudo touch /var/lib/docker/volumes/5cad.../_data/test-file
```

Этот файл сразу же можно увидеть внутри контейнера:

```
$ root@CONTAINER:/# ls /data
test-file
```

Второй способ – объявление тома с помощью инструкции `VOLUME` в файле Dockerfile:

```
FROM debian:wheezy
VOLUME /data
```

Результат будет в точности тот же самый, что при использовании ключа `-v` в команде `docker run`.

Установка прав доступа к тому в файле Dockerfile

Достаточно часто для тома требуются определение его владельца и установка прав доступа или инициализация тома с помощью некоторых данных по умолчанию или файлов конфигурации. Чрезвычайно важно помнить о том, что любая инструкция, расположенная после инструкции `VOLUME` в Dockerfile, не произведет никаких изменений в этом томе. Например, следующий Dockerfile будет работать совершенно не так, как ожидалось:

```
FROM debian:wheezy
RUN useradd foo
VOLUME /data
RUN touch /data/x
RUN chown -R foo:foo /data
```

Подразумевалось, что команды `touch` и `chown` будут выполнены в файловой системе образа, но в действительности они запускаются в томе временного контейнера, используемого для создания соответствующего уровня (более подробно см. раздел «Как создаются образы» в начале текущей главы). Этот том будет удален после выполнения перечисленных команд, что делает две последние инструкции бессмысленными.
Следующий Dockerfile будет работать правильно:

```
FROM debian:wheezy
RUN useradd foo
RUN mkdir /data && touch /data/x
RUN chown -R foo:foo /data
VOLUME /data
```

При запуске контейнера из этого образа Docker скопирует все файлы из каталога тома в образе в соответствующий том контейнера. Но этого не произойдет, если в качестве тома задан существующий каталог хоста (чтобы случайно не уничтожить файлов, хранящихся в файловой системе хоста).
Если по какой-либо причине невозможно определить владельца и установить права доступа в инструкции `RUN`, то воспользуйтесь инструкцией `CMD` или `ENTRYPOINT`, выполняющей соответствующий скрипт после создания контейнера.

Третий способ[1] состоит в расширении аргумента `-v` команды `docker run` с явным указанием связываемого каталога хоста в формате `-v HOST_DIR:CONTAINER_DIR`. Этот способ нельзя использовать в Dockerfile (так как он нарушает принцип переносимости и создает угрозу безопасности). Например:

```
$ docker run -v /home/adrian/data:/data debian ls /data
```

Здесь каталог файловой системы хоста */home/adrian/data* монтируется как */data* в контейнере. Все файлы, уже существующие в каталоге */home/adrian/data*, становятся доступными внутри контейнера. Если каталог */data* ранее существовал в контейнере, то его содержимое будет скрыто созданным томом. В отличие от других вариантов вызова, никакие файлы из образа не копируются в том, и этот том не будет удален механизмом Docker (то есть команда `docker rm -v` не удаляет том, который монтируется на каталог, указанный пользователем).

Монтирование каталогов как файловых систем
Использование явно заданного каталога хоста в томе (синтаксис с ключом `-v HOST_DIR:CONTAINER_DIR`) часто называют монтированием каталогов как файловых систем (bind mounting). Это не совсем правильно, поскольку с формальной точки зрения все тома являются каталогами, смонтированными как файловые системы. Разница лишь в том, что в одном случае точка монтирования указывается явно, а в другом она скрыта от пользователя в каталоге, принадлежащем механизму Docker.

Совместное использование данных

Синтаксис с использованием ключа `-v HOST_DIR:CONTAINER_DIR` очень удобен для совместного использования файлов хостом и одним или несколькими контейнерами. Например, файлы конфигурации могут храниться на хосте и монтироваться внутри контейнеров, создаваемых из однотипных образов.

Также можно совместно пользоваться одними и теми же данными в нескольких контейнерах, если указать ключ `--volumes-from CONTAINER` в команде `docker run`. Например, новый контейнер, который имеет доступ к томам контейнера, созданного в предыдущем примере, можно получить следующим образом:

```
$ docker run -it -h NEWCONTAINER --volumes-from container-test debian /bin/bash
root@NEWCONTAINER:/# ls /data
test-file
root@NEWCONTAINER:/#
```

Важно отметить, что этот способ работает вне зависимости от того, активен ли в текущий момент контейнер, содержащий тома (в нашем примере это `container-test`). Том невозможно удалить, пока существует хотя бы один контейнер, установивший связь с этим томом.

Контейнеры данных

На практике широко распространено создание *контейнеров данных* (*data containers*), единственной целью которых является обеспечение совместного использова-

[1] В действительности почти не отличающийся от первых двух.

ния данных несколькими контейнерами. Главное преимущество такого подхода состоит в предоставлении удобного пространства имен для томов, загружаемых просто с помощью ключа `--volumes-from` в команде `docker run`.

Например, можно создать контейнер данных для СУБД PostgreSQL, выполнив следующую команду:

```
$ docker run --name dbdata postgres echo "Data-only container for postgres"
```

Здесь создается контейнер из образа `postgres`, а все тома, определенные в этом образе, инициализируются до выполнения команды `echo` и выхода (завершения команды `docker run` в целом)[1]. Нет необходимости оставлять контейнеры данных в активном рабочем состоянии, так как это влечет за собой лишь ненужный расход ресурсов.

В дальнейшем мы можем пользоваться томами созданного контейнера данных, применяя аргумент `--volumes-from`. Например:

```
$ docker run -d --volumes-from dbdata --name db1 postgres
```

Образы для контейнеров данных

Обычно нет необходимости в использовании для контейнеров данных «минималистичных образов» типа `busybox` или `scratch`. Просто используйте тот же образ, что и для контейнера, работающего с данными. Например, при использовании образа `postgres` для создания контейнера данных следует взять тот же образ при создании контейнера для СУБД Postgres.

При использовании одного и того же образа не требуется дополнительное пространство – можете быть уверены, что образ для «потребителя» уже загружен или создан. Кроме того, возникает возможность ввести в контейнер некоторые начальные данные, и обеспечивается корректность установки прав доступа.

Удаление томов

Тома удаляются, только если:

- контейнер, содержащий тома, был удален командой `docker rm -v`

или

- в команду `docker run` был включен флаг `--rm`,

а также при соблюдении следующих условий:

- отсутствие контейнеров, установивших связь с удаляемыми томами;
- удаляемому тому не соответствует какой-либо каталог файловой системы хоста (то есть при создании тома не использовался синтаксис с ключом `-v HOST_DIR:CONTAINER_DIR`).

В данное время это означает, что следует всегда быть особенно внимательным при запуске контейнеров со связанными томами, в противном случае рабочий ка-

---

[1] Можно было бы использовать любую другую команду, после выполнения которой сразу же завершается команда `docker run`, но `echo`-сообщение напоминает нам о предназначении данного контейнера при выполнении команды `docker ps -a`. Другой вариант – вообще не запускать этот контейнер, а воспользоваться командой `docker create` вместо `docker run`.

талог Docker с большой вероятностью будет содержать «потерянные» файлы и каталоги (так называемые «сироты» – orphans), а что именно представляет каждый из них, определить сложно. В компании Docker уже разрабатывают набор команд самого верхнего уровня для томов, которые позволят выводить список томов, создавать, просматривать содержимое и удалять тома независимо от контейнеров. Эти изменения ожидаются в версии 1.9, которая должна выйти после публикации данной книги.

Часто используемые команды Docker

В этом разделе приведено краткое (по сравнению с официальной документацией) и далеко не полное описание различных команд Docker. Главное внимание сосредоточено на командах, которые выполняются наиболее часто. Поскольку программная среда Docker быстро изменяется и развивается, самую точную и подробную информацию об этих командах можно получить на сайте Docker в разделе официальной документации (http://docs.docker.com). Синтаксис и аргументы команд здесь не рассматриваются в подробностях (за исключением команды `docker run`). Для каждой команды имеется встроенная справка, которую можно вывести при помощи аргумента `--help` в конкретной команде или общей командой `docker help`.

Флаги с логическими значениями в командах Docker

Для большинства утилит командной строки Unix вы наверняка обнаружите флаги, не требующие какого-либо значения, например флаг `-l` в команде `ls -l`. Поскольку эти флаги либо присутствуют, либо отсутствуют, Docker определяет их как логические (boolean) флаги – что отличает Docker от большинства других утилит – и обеспечивает поддержку записи логического значения для флага в явной форме (таким образом, приемлемыми являются оба формата: `-f=true` и `-f`). Кроме того, могут существовать флаги, значение которых по умолчанию равно как true, так и false (и это изрядно сбивает с толку). В отличие от флагов с значением false по умолчанию, флаги со значением true по умолчанию считаются установленными, если они не указаны явно. Указание флага без аргумента дает тот же эффект, что и присваивание значения true, – флаг со значением true по умолчанию не отменяется аргументом с произвольным значением, единственный способ отмены флага со значением true – явная установка для него значения false (например, `-f=false`).

Чтобы определить значение флага по умолчанию, воспользуйтесь ключом `--help` для конкретной команды. Например:

```
$ docker logs --help
...
  -f, --follow=false       Follow log output (Вывод журнала операций)
  --help=false             Print usage (Вывод справки)
  -t, --timestamps=false   Show timestamps (Показывать метки времени)
...
```

Отсюда видно, что аргументам `-f`, `--help` и `-t` по умолчанию присвоено значение `false`.

Чтобы лучше понять все сказанное выше, рассмотрим несколько конкретных примеров с аргументом `--sig-proxy` (по умолчанию значение `true`) для команды `docker run`. Единственный способ отмены действия этого аргумента – явное присваивание ему значения false. Например:

```
$ docker run --sig-proxy=false ...
```

Ниже приведены команды, абсолютно равнозначные друг другу:

```
$ docker run --sig-proxy=true ...
$ docker run --sig-proxy ...
$ docker run ...
```

В случае с аргументом, значением которого по умолчанию является false, например `--read-only`, следующие команды изменят его значение на true:

```
$ docker run --read-only=true ...
$ docker run --read-only ...
```

Для данного аргумента его отсутствие в команде или явное присваивание значения false равнозначно.
Кроме всего прочего, такой подход иногда становится источником в некотором роде «хитрого» поведения при использовании флагов, которые обычно подчиняются логике вычисления по короткой схеме (например, `docker ps --help=false` будет работать как обычно, без вывода текста справки).

Команда run

Ранее мы уже наблюдали выполнение команды `docker run` – это команда запуска новых контейнеров. Поэтому на текущий момент она является самой сложной командой и поддерживает обширный список возможных аргументов. Аргументы позволяют пользователям конфигурировать процесс приведения образа в рабочее состояние, заменять параметры настройки из Dockerfile, определять параметры конфигурации сети, устанавливать права доступа и назначать ресурсы для создаваемого контейнера.

Следующие ключи управляют жизненным циклом контейнера и основным режимом его работы:

`-a, --attach`
: Подключает заданный поток (`STDOUT` и прочие) к терминалу. Если ключ не задан, то подключаются поток вывода `stdout` и поток ошибок `stderr`. Если ключ не задан и при этом контейнер запускается в интерактивном режиме (`-i`), то подключается еще и поток ввода `stdin`.
Несовместим с ключом `-d`.

`-d, --detach`
: Запускает контейнер в режиме «отключения от всех потоков». Команда запускает контейнер в фоновом режиме (background mode) и возвращает идентификатор (ID) контейнера.

`-i --interactive`

Поддерживает доступность открытого потока `stdin` (даже если он не был подключен). Как правило, используется вместе с ключом `-t` для запуска контейнера в интерактивном режиме. Например:

```
$ docker run -it debian /bin/bash
root@bd0f26f928bb:/# ls
...(вывод пропущен для экономии места)...
```

`--restart`

Позволяет настроить образ действий при попытке Docker перезапустить остановленный контейнер. Аргумент `no` запрещает любые попытки перезапуска контейнера. При аргументе `always` попытки перезапуска выполняются в любом случае вне зависимости от состояния контейнера после выхода. Если задан аргумент `on-failure`, то попытки перезапуска выполняются для контейнера, завершившего работу с ненулевым статусом. В последнем случае может быть задан дополнительный необязательный аргумент, определяющий максимальное количество попыток перезапуска (если этот аргумент не задан, то попытки будут выполняться бесконечно). Например, команда `docker run --restart on-failure:10 postgres` запускает контейнер `postgres`, и если контейнер завершает работу с ненулевым кодом, то выполняются 10 попыток его перезапуска.

`--rm`

Автоматически удаляет контейнер после завершения сеанса его работы (выхода). Несовместим с ключом `-d`.

`-t, --tty`

Создает псевдоустройство TTY (терминал). Как правило, используется вместе с ключом `-i` для запуска контейнера в интерактивном режиме.

Ниже описываются ключи для определения имен контейнеров и внутренних переменных их среды.

`-e, --env`

Определяет переменные среды внутри контейнера. Например:

```
$ docker run -e var1=val -e var2="val 2" debian env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=b15f833d65d8
var1=val
var2=val 2
HOME=/root
```

Также отметим, что имеется ключ `--env-file` для передачи переменных среды через заданный файл.

`-h --hostname`

Устанавливает для запускаемого контейнера заданное имя Unix-хоста. Например:

```
$ docker run -h "myhost" debian hostname
myhost
```

`--name NAME`
Присваивает контейнеру имя `NAME`. В дальнейшем это имя может использоваться для обращения к данному контейнеру в других командах Docker.

Ключи, описанные ниже, позволяют пользователю создавать и настраивать тома (более подробно об этом см. раздел «Управление данными с помощью томов и контейнеров данных» выше).

`-v, --volume`
Существуют две формы записи этого аргумента для создания и настройки тома (файл или каталог внутри контейнера, являющийся частью файловой системы хоста, а не файловой системы UnionFS контейнера). Первая форма определяет только каталог внутри контейнера, а связываемый с ним каталог хоста выбирает механизм Docker. Вторая форма определяет как внутренний каталог контейнера, так и связываемый с ним каталог хоста.

`--volumes-from`
Монтирует тома из заданного контейнера. Часто используется при работе с контейнерами данных (см. раздел «Контейнеры данных» выше).

Ниже приведены описания наиболее часто используемых ключей для настройки сетевой среды.

`--expose`
Аналог инструкции `EXPOSE` из файла Dockerfile. Определяет номер порта или диапазон номеров портов, предназначенных для использования в контейнере, но в действительности не открывает каких-либо портов. Применение этого ключа имеет смысл только в сочетании с ключом `-P`, а также при установлении соединений между контейнерами.

`--link`
Настраивает интерфейс частной закрытой сети для заданного контейнера. Более подробно об этом см. раздел «Соединение между контейнерами» выше.

`-p --publish`
«Публикует» порт данного контейнера, то есть делает его доступным с хоста. Если соответствующий порт хоста не определен, то произвольным образом выбирается свободный порт с большим номером (за пределами диапазона системных портов), который в дальнейшем можно узнать с помощью команды `docker port`. Также можно определить интерфейс хоста, для которого объявляется данный порт.

`-P, --publish-all`
Объявляет все порты, открываемые в контейнере, доступными на хосте. Для каждого объявляемого порта произвольным образом выбирается свободный

порт с большим номером. Чтобы увидеть установленные соответствия между портами, воспользуйтесь командой `docker port`.

Могут оказаться полезными некоторые ключи, предназначенные для более тонкой настройки сетевой среды. Но следует помнить о том, что применение этих ключей потребует более глубокого понимания работы в сети и реализации ее поддержки в Docker. Более подробно об этом можно прочесть в главе 11.

Кроме того, команда `docker run` имеет ряд ключей для управления привилегиями и функциональными возможностями контейнеров. Подробное описание см. в главе 13.

Следующие ключи предназначены для безусловной замены параметров настройки, определенных в файле Dockerfile.

`--entrypoint`
: Определяет точку входа для запускаемого контейнера в соответствии с заданным аргументом, заменяя содержимое любой инструкции `ENTRYPOINT` из Dockerfile.

`-u, --user`
: Определяет пользователя, от имени которого выполняются команды. Может быть задано как символьное имя пользователя или как числовой идентификатор UID. Заменяет содержимое инструкции `USER` из Dockerfile.

`-w, --workdir`
: Устанавливает рабочий каталог в контейнере в соответствии с заданным путевым именем. Заменяет любые значения, определенные в файле Dockerfile.

Управление контейнерами

Кроме команды `docker run`, существуют и другие команды `docker`, предназначенные для управления контейнерами на протяжении их жизненного цикла.

`docker attach [OPTIONS] CONTAINER`
: Команда `attach` позволяет пользователю наблюдать или взаимодействовать с основным процессом внутри контейнера. Например:

```
$ ID=$(docker run -d debian sh -c "while true; do echo 'tick'; sleep 1; done;")
$ docker attach $ID
tick
tick
tick
tick
...
```

Следует отметить, что использование комбинации клавиш **Ctrl+C** для выхода завершит наблюдаемый процесс и приведет к завершению работы контейнера.

`docker create`
: Создает контейнер из заданного образа, но не запускает его. Аргументы этой команды в основном те же, что для команды `docker run`. Чтобы запустить созданный контейнер, нужно выполнить команду `docker start`.

`docker cp`

Позволяет копировать файлы между файловыми системами контейнера и хоста.

`docker exec`

Запускает заданную команду внутри контейнера. Может использоваться для выполнения задач сопровождения или в качестве замены `ssh` при входе (регистрации) в контейнер. Например:

```
$ ID=$(docker run -d debian sh -c "while true; do sleep 1; done;")
$ docker exec $ID echo "Hello"
Hello
$ docker exec -it $ID /bin/bash
root@5c6c32041d68:/# ls
bin   dev  home  lib64  mnt  proc  run   selinux  sys  usr
boot  etc  lib   media  opt  root  sbin  srv      tmp  var
root@5c6c32041d68:/# exit
exit
```

`docker kill`

Посылает сигнал основному процессу (PID=1) в контейнере. По умолчанию посылает сигнал `SIGKILL`, по которому выполняется немедленное завершение работы контейнера. Можно передать другой сигнал с помощью дополнительного аргумента `-s`. Возвращает идентификатор контейнера. Например:

```
$ ID=$(docker run -d debian bash -c \
    "trap 'echo caught' SIGTRAP; while true; do sleep 1; done;")
$ docker kill -s SIGTRAP $ID
e33da73c275b56e734a4bbbefc0b41f6ba84967d09ba08314edd860ebd2da86c
$ docker logs $ID
caught
$ docker kill $ID
e33da73c275b56e734a4bbbefc0b41f6ba84967d09ba08314edd860ebd2da86c
```

`docker pause`

Временно приостанавливает все процессы внутри заданного контейнера. Процессы не получают никаких сигналов приостановки, следовательно, не могут быть полностью остановлены и завершены или удалены. Продолжить выполнение этих процессов можно с помощью команды `docker unpause`. Команда `docker pause` использует внутреннюю функциональную возможность приостановки (freezing) механизма cgroups в ядре Linux. Эта команда значительно отличается от команды `docker stop`, которая полностью останавливает выполнение всех процессов и посылает процессам сигналы в явной форме.

`docker restart`

Перезапускает один или несколько контейнеров. Можно считать приблизительным аналогом выполнения для заданных контейнеров команды `docker stop`, за которой сразу следует команда `docker start`. Имеется дополнительный аргу-

мент -t, определяющий интервал времени ожидания, необходимого для завершения работы контейнера, перед его остановом по сигналу SIGTERM.

`docker rm`

Удаляет один или несколько контейнеров. Возвращает имена или идентификаторов успешно удаленных контейнеров. По умолчанию `docker rm` не удаляет существующие тома. Аргумент -f позволяет удалять работающие контейнеры. С помощью аргумента -v можно удалить тома, созданные удаляемым контейнером (если эти тома не смонтированы на каталоги и не используются другими контейнерами). Например, для удаления всех остановленных контейнеров нужно выполнить команду:

```
$ docker rm $(docker ps -aq)
b7a4e94253b3
e33da73c275b
f47074b60757
```

`docker start`

Запускает остановленный контейнер (или несколько контейнеров). Может использоваться для повторного запуска остановленного контейнера или для запуска контейнера, ранее созданного командой `docker create`, но никогда не выполнявшегося.

`docker stop`

Останавливает (но не удаляет) один или несколько контейнеров. После выполнения этой команды заданный контейнер переходит в состояние «остановлен» («завершен»). Дополнительный аргумент -t определяет интервал времени ожидания, необходимого для завершения работы контейнера, перед его остановом по сигналу SIGTERM.

`docker unpause`

Перезапускает контейнер, выполнение которого было приостановлено командой `docker pause`.

Отключение от контейнеров

После подключения к контейнеру Docker при запуске его в интерактивном режиме или с помощью команды `docker attach` этот контейнер будет полностью остановлен, если попытаться отключиться от него с помощью комбинации клавиш **Ctrl**+**C**. Отключиться от контейнера без его остановки можно с помощью комбинации клавиш **Ctrl**+**P**, **Ctrl**+**Q**.

Этот способ работает только при подключении в интерактивном режиме с использованием терминального устройства TTY (то есть при использовании флагов -i и -t).

Информация о механизме Docker

Следующие команды используются для получения информации об установленной системе Docker и ее использовании:

`docker info`
Выводит различную информацию о системе Docker и хосте, на котором она работает.

`docker help`
Выводит информацию об использовании и справку по заданной команде. Аналогично выполнению команды с флагом `--help`.

`docker version`
Выводит информацию о версии клиента и сервера Docker, а также о версии языка программирования Go, используемого при компиляции.

Информация о контейнере

Следующие команды предоставляют информацию о работающих и остановленных контейнерах.

`docker diff`
Показывает изменения в файловой системе контейнера по сравнению с файловой системой образа, который был использован для запуска этого контейнера. Например:

```
$ ID=$(docker run -d debian touch /NEW-FILE)
$ docker diff $ID
A /NEW-FILE
```

`docker events`
Выводит в реальном времени события от демона демону. Для выхода из этого режима используйте комбинацию клавиш Ctrl-C. Более подробное описание этой команды см. в главе 10.

`docker inspect`
Предоставляет подробную информацию о заданных контейнерах или образах. В основном это информация о конфигурации, включающая параметры настройки сети и параметры привязки томов. Аргумент `-f` используется для определения шаблонов языка Go при форматировании и фильтрации вывода.

`docker logs`
Выводит журналы (logs) для контейнера. Выводится все, что было записано в потоки `STDERR` и `STDOUT` внутри контейнера. Более подробно о журналировании в системе Docker см. в главе 10.

`docker port`
Выводит список отображений открытых портов для заданного контейнера. Дополнительно могут быть заданы номер внутреннего порта контейнера и искомый протокол. Часто используется после выполнения команды `docker run -P <image>` для получения списка назначенных портов. Например:

```
$ ID=$(docker run -P -d redis)
$ docker port $ID
```

```
6379/tcp -> 0.0.0.0:32768
$ docker port $ID 6379
0.0.0.0:32768
$ docker port $ID 6379/tcp
0.0.0.0:32768
```

`docker ps`

Предоставляет общую информацию о работающих контейнерах: имя, идентификатор, состояние. У этой команды большое количество разнообразных аргументов, среди которых наиболее важным является `-a`, позволяющий вывести информацию обо всех контейнерах, а не только о работающих в текущий момент. Также следует отметить аргумент `-q`, возвращающий только идентификаторы контейнеров, что очень удобно для организации потока ввода для других команд, например `docker rm`.

`docker top`

Предоставляет информацию о процессах, выполняющихся внутри заданного контейнера. В действительности эта команда запускает утилиту Unix `ps` на хосте и выбирает для вывода процессы, выполняющиеся в заданном контейнере. Аргументы для этой команды совпадают с аргументами утилиты `ps`, и по умолчанию установлены аргументы `-ef` (но помните о необходимости сохранить поле идентификатора процесса PID при выводе результатов). Например:

```
$ ID=$(docker run -d redis)
$ docker top $ID
UID  PID   PPID  C  STIME  TTY  TIME      CMD
999  9243  1836  0  15:44  ?    00:00:00  redis-server *:6379
$ ps -f -u 999
UID  PID   PPID  C  STIME  TTY  TIME      CMD
999  9243  1836  0  15:44  ?    00:00:00  redis-server *:6379
$ docker top $ID -axZ
LABEL           PID   TTY  STAT  TIME  COMMAND
docker-default  9243  ?    Ssl   0:00  redis-server *:6379
```

Работа с образами

Следующие команды представляют собой инструменты для создания образов и работы с ними:

`docker build`

Создает образ из файла Dockerfile. Подробности использования этой команды см. в разделах «Создание образов из файлов Dockerfile» (глава 3) и «Как создаются образы» текущей главы.

`docker commit`

Создает образ из указанного контейнера. Эта команда иногда может быть полезной, тем не менее в большинстве случаев предпочтительнее пользоваться командой `docker build`, которая с легкостью воспроизводится повторно. По умолчанию контейнеры временно приостанавливаются перед созданием обра-

за, но приостановку можно отменить с помощью аргумента --pause=false. Для настройки метаданных применяются аргументы -a и -m. Например:

```
$ ID=$(docker run -d redis touch /new-file)
$ docker commit -a "Joe Bloggs" -m "Comment" $ID commit:test
ac479108b0fa9a02a7fb290a22dacd5e20c867ec512d6813ed42e3517711a0cf
$ docker images commit
REPOSITORY  TAG   IMAGE ID      CREATED             VIRTUAL SIZE
commit      test  ac479108b0fa  About a minute ago  111 MB
$ docker run commit:test ls /new-file
/new-file
```

docker export

Экспортирует содержимое файловой системы заданного контейнера в виде tar-архива, направляя его в стандартный поток вывода STDOUT. Созданный таким образом архив может быть загружен командой docker import. Следует отметить, что экспортируется только файловая система, но все метаданные, такие как объявленные порты, команды CMD, параметры настройки ENTRYPOINT, будут потеряны. Также отметим, что в экспортируемую файловую систему не включаются какие-либо тома. Сравните с командой docker save.

docker history

Выводит информацию о каждом уровне в образе.

docker images

Выводит список локальных образов, содержащий такую информацию, как имя репозитория, имя тега, размер и др. По умолчанию не выводится информация о промежуточных вспомогательных образах (используемых при создании образов верхнего уровня). В столбце VIRTUAL SIZE показан общий суммарный размер образа с учетом всех нижележащих уровней. Так как эти уровни могут совместно использоваться несколькими образами, добавление их размера во все соответствующие образы не позволяет точно оценить степень использования дискового пространства. Кроме того, образы могут появляться в списке несколько раз, если имеют более одного тега, образы можно различать по идентификатору. Команда имеет несколько аргументов, среди которых следует выделить -q, возвращающий только идентификаторы образов, что удобно для передачи в поток ввода других команд, таких как docker rmi. Например:

```
$ docker images | head -4
REPOSITORY              TAG     IMAGE ID      CREATED       VIRTUAL SIZE
identidock_identidock   latest  9fc66b46a2e6  26 hours ago  839.8 MB
redis                   latest  868be653dea3  6 days ago    110.8 MB
containersol/pres-base  latest  13919d434c95  2 weeks ago   401.8 MB
```

Пример удаления всех «висячих» образов:

```
$ docker rmi $(docker images -q -f dangling=true)
Deleted: a9979d5ace9af55a562b8436ba66a1538357bc2e0e43765b406f2cf0388fe062
```

`docker import`

Создает образ из архивного файла, содержащего файловую систему и созданного командой `docker export`. Архив может быть задан путем к файлу или в форме URL, а также передан через стандартный поток ввода `STDIN` (если указан флаг `-`). Возвращает идентификатор нового созданного образа. Образ может быть помечен тегом, состоящим из имени репозитория и имени тега. Следует отметить, что образ, созданный командой `docker import`, состоит только из одного уровня, и в нем отсутствуют параметры конфигурации Docker, такие как объявленные порты и значения инструкций `CMD`. Сравните с командой `docker load`. Пример «сращивания» образа до одного уровня в результате операций экспорта и импорта:

```
$ docker export 35d171091d78 | docker import - flatten:test
5a9bc529af25e2cf6411c6d87442e0805c066b96e561fbd1935122f988086009
$ docker history flatten:test
IMAGE         CREATED         CREATED BY  SIZE      COMMENT
981804b0c2b2  59 seconds ago              317.7 MB  Imported from -
```

`docker load`

Загружает репозиторий из tar-архива, передаваемого через стандартный поток ввода `STDIN`. Репозиторий может содержать несколько образов и тегов. В отличие от команды `docker import`, в загружаемые образы включены метаданные и история. Подходящие для загрузки архивные файлы создаются командой `docker save`, что делает комбинацию `save`/`load` вполне жизнеспособной альтернативой реестрам для распространения образов и выполнения операций резервного копирования. Пример приведен в описании команды `docker save`.

`docker rmi`

Удаляет заданный образ или несколько образов. Образы определяются идентификатором или комбинацией имен репозитория и тега. Если указано только имя репозитория без тега, то предполагается тег `latest`. Для удаления образов, существующих в нескольких репозиториях, необходимо указать идентификатор образа и воспользоваться аргументом `-f`. Такую команду нужно выполнить по одному разу в каждом репозитории.

`docker save`

Сохраняет именованные образы или репозитории в tar-архив, передаваемый в стандартный поток вывода `STDOUT` (для записи в файл используйте аргумент `-o`). Образы можно задавать по идентификаторам или в форме `repository:tag`. Если задано только имя репозитория, то в архиве будут сохранены все образы из этого репозитория, а не только имеющие тег `latest`. Часто используется в сочетании с командой `docker load` для распространения или резервного копирования образов. Например:

```
$ docker save -o /tmp/redis.tar redis:latest
$ docker rmi redis:latest
Untagged: redis:latest
```

```
Deleted: 868be653dea3ff6082b043c0f34b95bb180cc82ab14a18d9d6b8e27b7929762c
...
$ docker load -i /tmp/redis.tar
$ docker images redis
REPOSITORY    TAG       IMAGE ID        CREATED         VIRTUAL SIZE
redis         latest    0f3059144681    3 months ago    111 MB
```

`docker tag`

Связывает имя репозитория и тега с заданным образом. Образ можно указывать по идентификатору или с помощью имен репозитория и тега (если тег не указан, то предполагается `latest`). Если для нового имени тег не задан, то по умолчанию присваивается тег `latest`. Например:

```
$ docker tag faa2b75ce09a newname ❶
$ docker tag newname:latest amouat/newname ❷
$ docker tag newname:latest amouat/newname:newtag ❸
$ docker tag newname:latest myregistry.com:5000/newname:newtag ❹
```

❶ Добавляет образ с идентификатором `faa2b75ce09a` в репозиторий `newname`, присваивая образу тег `latest` по умолчанию, так как тег не был указан.

❷ Добавляет образ `newname:latest` в репозиторий `amouat/newname`, опять с присваиванием по умолчанию тега `latest`. Этот формат подходит для передачи образов в реестр Docker Hub, если операцию выполняет пользователь `amouat`.

❸ Команда аналогична предыдущей, но в этом случае вместо тега по умолчанию явно присваивается тег `newtag`.

❹ Добавляет образ `newname:latest` в репозиторий `myregistry.com/newname` с тегом `newtag`. Этот формат подходит для передачи образов в реестр, расположенный на сайте http://myregistry.com:5000.

Команды для работы с реестром

Следующие команды используются для работы с реестрами, в том числе и с Docker Hub. Следует помнить, что Docker сохраняет аутентификационную информациюв файле *.dockercfg*, расположенном в вашем домашнем каталоге.

`docker login`

Выполняет процедуру регистрации или входа на заданный сервер реестра. Если сервер не задан, то по умолчанию предполагается Docker Hub. При необходимости в ходе этой процедуры в интерактивном режиме будет запрашиваться дополнительная уточняющая информация, но эту информацию можно передать в виде аргументов.

`docker logout`

Выполняет процедуру выхода из реестра Docker. Если сервер не задан, то по умолчанию предполагается Docker Hub.

`docker pull`

Загружает заданный образ из реестра. Реестр определяется по имени образа, а по умолчанию принимается Docker Hub. Если не задано имя тега, будет за-

гружен образ с тегом `latest` (при наличии такого образа в данном реестре). Для загрузки всех образов из репозитория используется аргумент `-a`.

`docker push`
: Выгружает образ или репозиторий в заданный реестр. При отсутствии тега выгружаются все образы указанного репозитория в заданный реестр, а не только образы с тегом `latest`.

`docker search`
: Выводит список общедоступных репозиториев из реестра Docker Hub, соответствующих заданному шаблону поиска. Результат может содержать не более 25 репозиториев. Также можно определить фильтр по количеству звездочек (положительных оценок) и для автоматизированных сборок. Но в самом общем случае гораздо проще выполнить поиск непосредственно на самом веб-сайте.

Резюме

Из этой главы вы получили большой объем информации. Если вы лишь бегло ознакомились с основными положениями, вы должны иметь достаточно широкое понимание как механизма Docker в целом, так и работы основных его команд. В части II мы рассмотрим практическое применение полученных знаний в конкретном программном проекте на протяжении всего жизненного цикла – от разработки до реального «производственного» применения. Возможно, некоторый материал из этой главы станет более понятным в ходе практической работы.

Часть II

ЖИЗНЕННЫЙ ЦИКЛ ПО ПРИ ИСПОЛЬЗОВАНИИ DOCKER

В части I мы изучили основные принципы организации контейнеров и познакомились с основами их практического применения. В части II практическое применение контейнеров рассматривается более подробно, с использованием Docker для создания, тестирования и развертывания веб-приложения. Будут описаны возможные способы применения контейнеров Docker в процессах разработки, тестирования и эксплуатации. В этой части внимание сосредоточено на организации системы, располагающейся на одном хосте, а подробное описание развертывания и организации контейнеров на нескольких хостах приведено в части III.

После изучения части II вы будете лучше понимать, как включить Docker в процесс разработки ПО, и более уверенно использовать Docker в повседневной работе. Чтобы воспользоваться большинством преимуществ Docker, важно усвоить и принять подход DevOps. В частности, при разработке мы будем думать о том, как ввести ПО в эксплуатацию, освобождаясь от проблемы развертывания ПО в различных программных средах.

Несмотря на то что приложение, которое будет создаваться на протяжении всех глав части II, в учебных целях остается небольшим и очень простым, мы все же в полной мере рассмотрим технологии и практические методики, необходимые для разработки и ввода в эксплуатацию крупномасштабных приложений, сопровождаемых большими группами разработчиков.

Контейнеры не подходят для создания монолитного корпоративного ПО с циклом выпуска версий, измеряемого в неделях или месяцах. Вместо этого мы естественным образом придем к технологии микросервисов и будем изучать такие методики, как непрерывное развертывание (где оно применимо) для безопасного ввода ПО в эксплуатацию несколько раз в день.

Преимущества контейнеров, DevOps, микросервисов и непрерывного развертывания неизбежно приводят к осмыслению реализации цикла с быстрой обратной связью. Ускоряя итерации цикла, мы можем разрабатывать, тестировать и выполнять валидацию высококачественных систем за более короткое время.

Глава 5

Использование Docker в процессе разработки

На протяжении всей части II мы будем разрабатывать простое веб-приложение, которое возвращает неповторяющееся изображение для заданной строки, по аналогии с пиктограммами идентификации (identicons) на сайтах GitHub и StackOverflow, предназначенными для пользователей, которые не выбрали для себя картинки. Приложение будет создаваться с использованием языка программирования Python и программной среды для веб-приложений Flask. Для учебного примера выбран язык Python, потому что он широко применяется, а также из-за его лаконичности и удобочитаемости. Если вы никогда не программировали на Python, не беспокойтесь. Мы сосредоточимся на взаимодействии с Docker, а не на подробностях написания исходного кода на Python[1]. Flask тоже выбран из-за своей простоты – его легко изучать. Мы будем использовать Docker для управления всеми требуемыми зависимостями, поэтому нет необходимости в установке Python и Flask на вашем компьютере.

В этой главе главное внимание уделено организации рабочего потока, основанного на контейнерах, и практическому применению соответствующих инструментов до начала собственно разработки, которая будет рассматриваться в следующей главе.

Традиционное приветствие миру

Начнем с простого веб-сервера, который возвращает тривиальную фразу «Hello World!». Сначала создадим новый каталог *identidock* для хранения нашего проекта. В этом каталоге создадим подкаталог *app* для файлов исходного кода на Python. В подкаталоге *app* создадим файл *identidock.py*:

```
$ tree identidock/
identidock/
```

[1] Если вы хотите узнать больше о языке Python и среде Flask, то прочтите книгу Flask Web Development (http://shop.oreilly.com/product/0636920031116.do) Мигеля Гринберга (Miguel Grinberg) (O'Reilly), особенно если планируете заняться созданием веб-приложений.

```
|__ app
    |__ identidock.py

1 directory, 1 file
```

В файл *identidock.py* введите следующий код:

```
from flask import Flask
app = Flask(__name__) ❶

@app.route('/') ❷
def hello_world():
    return 'Hello World!\n'

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0') ❸
```

❶ Инициализация Flask и настройка объекта приложения.
❷ Создание маршрута , связанного с целевым URL. При любом обращении к этому URL будет вызываться функция `hello_world`.
❸ Инициализация веб-сервера в программной среде Python. Использование адреса `0.0.0.0` (вместо `localhost` или `127.0.0.1`) в качестве аргумента `host` позволяет выполнить привязку всех сетевых интерфейсов. Это необходимо для обеспечения доступа к контейнеру с хоста и из других контейнеров. Оператор `if` в предыдущей строке обеспечивает выполнение текущей строки только в том случае, когда файл вызывается как независимая программа, а не запускается как составная часть более крупного приложения.

Исходный код
Исходные коды для данной главы можно найти на сайте GitHub (https://github.com/using-docker/using_docker_in_dev). Все исходные коды снабжены тегами, обозначающими различные этапы разработки в соответствии с содержимым текущей главы.
Ранее я уже отмечал неудобства при вводе исходного кода вручную по тексту печатной книги или при копировании/вставке из электронной версии книги. Если у вас возникают проблемы, то воспользуйтесь репозиторием GitHub.

Теперь нам нужно поместить написанный код в контейнер и выполнить его. В каталоге *identidock* создайте файл *Dockerfile* со следующим содержимым:

```
FROM python:3.4

RUN pip install Flask==0.10.1
WORKDIR /app
COPY app /app

CMD ["python", "identidock.py"]
```

Этот Dockerfile использует в качестве основного официальный образ Python, содержащий установленную версию Python 3. Поверх этого уровня устанавливается Flask и копируется в наш код. Инструкция `CMD` просто запускает код из файла *identidock.py*.

Варианты официальных образов

Многие официальные репозитории для широко распространенных языков программирования, таких как Python, Go, Ruby, содержат несколько образов для различных целей. В дополнение к образам, соответствующим отдельным версиям, вам могут предложить следующие варианты:

- slim – эти образы представляют собой усеченные версии стандартных образов. Из них исключены многие общеупотребительные пакеты и библиотеки. Это важно, если необходимо уменьшить размер образа для распространения, но при этом, как правило, требуются дополнительные трудозатраты по установке и сопровождению пакетов, уже доступных в стандартном образе;
- onbuild – эти образы используют инструкцию Dockerfile `ONBUILD` для задержки выполнения определенных команд до момента создания нового образа-«потомка», являющегося наследником основного образа. Эти команды обрабатываются как часть инструкции `FROM` для образа-потомка и обычно выполняют такие операции, как копирование дополнительного кода и запуск очередного этапа компиляции. Такие образы могут упростить и ускорить процесс на начальной стадии работы с каким-либо языком, но при долговременной работе они могут создавать ограничения и вносить беспорядок. Вообще говоря, я бы рекомендовал применение onbuild-образов только при первоначальном ознакомлении с репозиторием.

В нашем примере приложения используется только стандартный основной образ для Python 3, мы не рассматриваем каких-либо его вариантов.

Теперь можно создать и запустить наше простейшее приложение:

```
$ cd identidock
$ docker build -t identidock .
...
$ docker run -d -p 5000:5000 identidock
0c75444e8f5f16dfe5aceb0aae074cc33dfc06f2d2fb6adb773ac51f20605aa4
```

Здесь в команду `docker run` передается флаг `-d`, чтобы запустить контейнер в фоновом режиме, но вы можете исключить его, если хотите наблюдать вывод результата работы веб-сервера. Аргумент `-p 5000:5000` сообщает механизму Docker о необходимости перенаправления порта 5000 в контейнере в порт 5000 на хосте.

Протестируем наш контейнер:

```
$ curl localhost:5000
Hello World!
```

IP-адреса Docker-машины

Если вы запускаете Docker с помощью Docker-машины (в случае установки через Docker Toolbox в ОС Mac OS или Windows), то не сможете использовать аргумент `localhost` в качестве URL. Вместо этого потребуется IP-адрес виртуальной машины, на которой работает Docker. Для получения этого адреса можно воспользоваться командой ip Docker-машины. Например:

```
$ curl $(docker-machine ip default):5000
Hello World!
```

В примерах данной книги предполагается, что Docker работает локально, в противном случае необходимо заменять `localhost` на соответствующий IP-адрес.

Сервер работает. Но существует достаточно серьезная проблема, связанная с организацией рабочего потока в его текущем состоянии: при любом, даже минимальном изменении исходного кода придется заново создавать образ и перезапускать контейнер. Хорошо, что для устранения этой проблемы есть простое решение. Можно *смонтировать каталог* хоста с исходными кодами *как отдельную файловую систему* (*bind mount*) внутри контейнера. В приведенном ниже фрагменте кода контейнер, запущенный последним, останавливается и удаляется (если последним запущенным не был контейнер из предыдущего примера, то нужно определить идентификатор контейнера с веб-сервером с помощью команды `docker ps`), после чего запускается новый контейнер с каталогом исходных кодов, смонтированным в каталог */app внутри контейнера*:

```
$ docker stop $(docker ps -lq)
0c75444e8f5f
$ docker rm $(docker ps -lq)
$ docker run -d -p 5000:5000 -v "$(pwd)"/app:/app identidock
```

Аргумент `-v "$(pwd)"/app:/app` позволяет смонтировать подкаталог *app* из текущего рабочего каталога хоста на каталог */app* внутри контейнера. Старое содержимое каталога */app* контейнера заменяется новым, при этом сохраняется возможность записи в этот каталог (если запись нежелательна, можно смонтировать том в режиме только для чтения). Аргументы для ключа `-v` обязательно должны быть абсолютными путевыми именами, поэтому в команде используется значение `$(pwd)` для получения полного пути к текущему каталогу и соблюдения принципа переносимости.

Монтирование каталогов как файловых систем

Когда каталог хоста определяется как том с помощью аргумента `-v HOST_DIR:CONTAINER_DIR` в команде `docker run`, выполняемую при этом операцию часто обозначают как «монтирование каталога как файловой системы» (bind mount), поскольку она связывает каталог (или файл) хоста с каталогом (или файлом) внутри контейнера. Это приводит к некоторой путанице, так как с формальной точки зрения все тома монтируются как файловые системы, но если каталог хоста не был задан явно при монтировании, то потребуются дополнительные усилия, чтобы обнаружить этот каталог.

Следует отметить, что `HOST_DIR` всегда ссылается на каталог, расположенный на компьютере, на котором работает механизм Docker. Если установлено соединение с удаленным демоном Docker, то заданное путевое имя непременно должно существовать на удаленном компьютере. Если используется локальная виртуальная машина, поддерживаемая Docker-машиной (при установке Docker через Toolbox), то будет выполнено кросс-монтирование (монтирование с пересечением границ файловых систем) вашего домашнего каталога для упрощения процесса разработки.

Проверим работоспособность нового контейнера:

```
$ curl localhost:5000
Hello World!
```

В последнем примере мы смонтировали тот же каталог, который был добавлен в образ с помощью команды `COPY`, но различие существенное: сейчас мы используем один и тот же каталог на хосте и внутри контейнера, а не его копию из образа. Именно поэтому теперь можно редактировать файл исходного кода *identidock.py* и сразу же видеть результат внесенных изменений:

```
$ sed -i '' s/World/Docker/ app/identidock.py
$ curl localhost:5000
Hello Docker!
```

Здесь используется утилита `sed` для быстрой замены фрагмента текста в файле *identidock.py*. Если утилита `sed` недоступна или вы никогда не работали с ней, то можно просто открыть файл исходного кода в любом текстовом редакторе и заменить слово `"World"` на `"Docker"`.

Теперь у нас есть вполне приемлемая среда разработки, но не хватает полного комплекта зависимостей – компилятора и библиотек языка Python, включенных в состав Docker-контейнера. В любом случае эта задача остается самой главной. Мы вряд ли будем использовать контейнер из текущего примера для производственной эксплуатации, в основном потому, что он запускает веб-сервер Flask в конфигурации по умолчанию, предназначенной только для разработки и чрезвычайно неэффективной и незащищенной для реального применения. Одним из самых важных преимуществ использования Docker является сведение к минимуму различий между режимом разработки и режимом эксплуатации, поэтому сейчас мы узнаем, что можно сделать в этом направлении.

Но почему бы не воспользоваться virtualenv?

Опытных разработчиков на языке Python может удивить тот факт, что для разработки нашего приложения не используется virtualenv (https://virtualenv.pypa.io/en/latest/). virtualenv представляет собой чрезвычайно полезный инструмент для изоляции программных сред Python. Он позволяет разработчикам одновременно использовать различные версии Python с поддержкой соответствующих библиотек для каждого приложения в отдельности. В большинстве случаев это очень важно и даже неизбежно в процессе разработки на языке Python.

Но при использовании контейнеров этот инструмент менее полезен, так как контейнеры сами предоставляют изолированную среду. Разумеется, если вы не можете отказаться от virtualenv, то продолжайте пользоваться им и внутри контейнера, но, скорее всего, больших преимуществ от этого вы не получите, за исключением тех случаев, когда вероятны конфликты между несколькими приложениями или библиотеками, установленными внутри контейнера.

uWSGI (https://uwsgi-docs.readthedocs.org/en/latest/) – это готовый к промышленной эксплуатации сервер приложений, который также может работать в связке с обычным веб-сервером, таким как nginx. Использование uWSGI вместо веб-

сервера Flask с конфигурацией по умолчанию предоставляет в наше распоряжение контейнер с весьма гибкими свойствами, который можно применять для широкого диапазона задач. Переход к использованию сервера uWSGI осуществляется редактированием всего лишь двух строк в Dockerfile:

```
FROM python:3.4

RUN pip install Flask==0.10.1 uWSGI==2.0.8 ❶
WORKDIR /app
COPY app /app

CMD ["uwsgi", "--http", "0.0.0.0:9090", "--wsgi-file", "/app/identidock.py", \
     "--callable", "app", "--stats", "0.0.0.0:9191"] ❷
```

❶ Добавление uWSGI в список устанавливаемых пакетов Python.
❷ Создание новой команды для запуска uWSGI. Здесь uWSGI получает указание запустить http-сервер, прослушивающий порт 9090 и инициализирующий приложение `app` из файла */app/identidock.py*. Также инициализируется сервер наблюдения за состоянием с прослушиванием порта 9191. Другой способ замены команды `CMD` – передача соответствующих аргументов в команду `docker run`.

После создания нового образа запустим его, чтобы наблюдать различия:

```
$ docker build -t identidock .
...
Successfully build 3133f91af597
$ docker run -d -p 9090:9090 -p 9191:9191 identidock
00d6fa65092cbd91a97b512334d8d4be624bf730fcb482d6e8aecc83b272f130
$ curl localhost:9090
Hello Docker!
```

Если теперь выполнить команду `docker logs` с указанием идентификатора этого контейнера, то мы увидим информацию из журналов для uWSGI, подтверждающую, что сейчас действительно используется сервер uWSGI. Также можно запросить у сервера uWSGI статистические данные о состоянии на http://localhost:9191. Здесь код Python-программы, в обычном режиме инициализирующий веб-сервер по умолчанию, не выполняется, так как он не был запущен непосредственно из командной строки.

Теперь сервер работает правильно, но необходимо выполнить некоторые вспомогательные административные действия. Если внимательно изучить журналы uWSGI, то можно заметить, что сервер совершенно справедливо предупреждает об опасности работы из-под учетной записи root. Это очевидное нарушение безопасности, которое можно легко устранить, определив в Dockerfile пользователя, от имени которого должен запускаться сервер. Кроме того, мы в явной форме объявим порты, открытые контейнером для прослушивания:

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi ❶
RUN pip install Flask==0.10.1 uWSGI==2.0.8
WORKDIR /app
```

```
COPY app /app

EXPOSE 9090 9191 ❷
USER uwsgi ❸

CMD ["uwsgi", "--http", "0.0.0.0:9090", "--wsgi-file", "/app/identidock.py", \
  "--callable", "app", "--stats", "0.0.0.0:9191"]
```

❶ Создание группы и пользователя `uwsgi` с помощью штатных средств Unix.
❷ Использование инструкции `EXPOSE` для объявления портов, доступных для хоста и других контейнеров.
❸ Определение имени пользователя `uwsgi` для всех последующих строк (включая `CMD` и `ENTRYPOINT`).

Пользователи и группы внутри контейнера

Ядро Linux использует идентификаторы пользователей UID и идентификаторы групп GID для идентификации пользователей и определения их прав доступа. Преобразование числовых идентификаторов UID и GID в символьные идентификаторы выполняется операционной системой в пространстве пользователя. Поэтому идентификаторы пользователей UID в контейнере совпадают с аналогичными UID на хосте, но пользователи и группы, созданные внутри контейнера, не передаются на хост. Из-за этого возникает побочный эффект – может возникать беспорядок в правах доступа, а одни и те же файлы могут принадлежать одному пользователю внутри контейнера и другому пользователю вне контейнера. В качестве примера рассмотрим изменение владельца конкретного файла:

```
$ ls -l test-file
-rw-r--r--    1 docker   staff 0 Dec 28 18:26  test-file
$ docker run -it -v $(pwd)/test-file:/test-file debian bash
root@e877f924ea27:/# ls -l test-file
-rw-r--r--    1 1000  staff 0 Dec 28 18:26 test-file
root@e877f924ea27:/# useradd -r test-user
root@e877f924ea27:/# chown test-user test-file
root@e877f924ea27:/# ls -l /test-file
-rw-r--r--    1 test-user staff 0 Dec 28 18:26 /test-file
root@e877f924ea27:/# exit
exit
docker@boot2docker:~$ ls -l test-file
-rw-r--r--    1 999      staff 0 Dec 28 18:26  test-file
```

После создания измененного образа обычным способом проверим конфигурацию с новым пользователем:

```
$ docker build -t identidock .
...
$ docker run identidock whoami
uwsgi
```

Обратите внимание: мы заменили заданную по умолчанию инструкцию CMD, инициализирующую веб-сервер, на команду `whoami`, которая возвращает текущее имя активного пользователя внутри контейнера.

Всегда определяйте имя пользователя с помощью инструкции USER
Необходимо включать инструкцию `USER` для определения пользователя во все файлы `Dockerfile` (или для изменения пользователя в скриптах для инструкций `ENTRYPOINT` или `CMD`). Если этого не сделать, то внутри контейнера все процессы будут запускаться от имени суперпользователя root. Так как идентификаторы пользователя в контейнере и на хосте одинаковы, после взлома контейнера извне нарушитель получит права суперпользователя root на хост-компьютере.
В настоящее время ведется разработка механизма автоматического преобразования идентификатора root внутри контейнера в идентификатор пользователя с большим UID на хосте, но на момент публикации книги (Docker версии 1.8) эта разработка еще не завершена.

Теперь команды внутри контейнера не запускаются от имени суперпользователя root. Запустим контейнер еще раз, но с другими аргументами:

```
$ docker run -d -P --name port-test identidock
```

В этот раз не заданы конкретные порты хоста для привязки. Вместо этого используется аргумент `-P`, определяющий автоматическое назначение механизмом Docker произвольно выбираемых портов с большими номерами на хосте для каждого объявленного открытым порта в контейнере. Прежде чем получить доступ к сервису, поддерживаемому данным контейнером, нужно узнать выбранные номера портов на хосте:

```
$ docker port port-test
9090/tcp -> 0.0.0.0:32769
9191/tcp -> 0.0.0.0:32768
```

Теперь мы знаем, что порт контейнера 9090 связан с портом хоста 32769, а порт 9191 – с портом 32768, и можем получить доступ к нашему сервису (отметим, что на вашей системе могут быть выбраны другие номера портов хоста):

```
$ curl localhost:32769
Hello Docker!
```

На первый взгляд последнее действие может выглядеть ненужным (в нашем простом примере так и есть), но если на одном хосте работает большое количество контейнеров, то гораздо проще передать механизму Docker функцию автоматического назначения свободных портов, чем самому следить за правильностью их распределения.

Итак, наш веб-сервис работает почти так, как должен работать при реальной эксплуатации. Осталось еще немало аспектов, требующих дополнительной настройки, например настройка параметров uWSGI для рациональной организации процессов и потоков, но главный шаг от имитационного сервера, по умолчанию предназначенного для отладки Python-программ, к настоящему веб-серверу уже сделан.

Возникает новая проблема: потерян доступ к инструментам разработки, таким как вывод отладочной информации и перезагрузка измененного кода в режиме реального времени, которые поддерживались веб-сервером Python по умолча-

нию. Можно свести к минимуму различия между средами разработки и эксплуатации, но в любом случае продолжают существовать ключевые пункты, для которых всегда требуются некоторые изменения. В идеальном случае хотелось бы использовать один и тот же образ как для разработки, так и для эксплуатации, но при этом иметь возможность изменения набора параметров, определяющих функциональные возможности, в зависимости от условий работы этого образа. Это возможно, если воспользоваться переменными среды и написать простой скрипт для выбора комплекта функциональных возможностей в зависимости от контекста.

В том же каталоге, где находится *Dockerfile*, создайте файл *cmd.sh* со следующим содержимым:

```
#!/bin/bash
set -e

if [ "$ENV" = 'DEV' ]; then
  echo "Running Development Server" # Запуск сервера для разработки
  exec python "identidock.py"
else
  echo "Running Production Server" # Запуск сервера для эксплуатации
  exec uwsgi --http 0.0.0.0:9090 --wsgi-file /app/identidock.py \
             --callable app --stats 0.0.0.0:9191
fi
```

Задача этого скрипта вполне очевидна. Если значение переменной `ENV` равно `DEV`, то запускается веб-сервер для отладки, в противном случае будет использоваться «настоящий» сервер для эксплуатации[1]. Команда `exec` нужна для того, чтобы не создавать нового процесса и обеспечить получение и обработку любых сигналов (таких как `SIGTERM`) единственным процессом uwsgi, а не избыточным родительским процессом.

Использование файлов конфигурации и вспомогательных скриптов
Для упрощения работы я включил все описанное выше в Dockerfile. Но по мере роста и развития приложения имеет смысл вынести некоторые фрагменты и скрипты во внешние файлы поддержки, если есть такая возможность. В частности, зависимости для команды pip следует переместить в файл requirements.txt, а для хранения параметров конфигурации сервера uWSGI можно создать отдельный .ini-файл.

Необходимо отредактировать Dockerfile, чтобы задействовать приведенный выше скрипт:

```
FROM python:3.4

RUN grouppadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask==0.10.1 uWSGI==2.0.8
```

[1] Теперь некоторые переменные, например номера портов, дублируются в нескольких файлах. Это можно исправить с помощью передаваемых аргументов или переменных среды.

```
WORKDIR /app
COPY app /app
COPY cmd.sh / ❶

EXPOSE 9090 9191
USER uwsgi

CMD ["/cmd.sh"] ❷
```

❶ Добавление файла скрипта в контейнер.
❷ Вызов скрипта из инструкции CMD.

Прежде чем проверить новую версию, нужно остановить все ранее запущенные контейнеры. Следующие команды останавливают все работающие контейнеры и удаляют их с хоста, поэтому не выполняйте их, если некоторые работающие контейнеры необходимо сохранить:

```
$ docker stop $(docker ps -q)
c4b3d240f187
9be42abaf902
78af7d12d3bb

$ docker rm $(docker ps -aq)
1198f8486390
c4b3d240f187
9be42abaf902
78af7d12d3bb
```

Теперь можно создать новый образ со скриптом выбора режима работы и протестировать его:

```
$ chmod +x cmd.sh
$ docker build -t identidock .
...
$ docker run -e "ENV=DEV" -p 5000:5000 identidock
Running Development Server
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
```

Задача выполнена. Теперь при запуске с аргументом `-e "ENV=DEV"` мы получаем сервер для разработки и отладки, при запуске без этого аргумента сервер начинает работу в режиме эксплуатации.

Серверы для разработки

Python-сервер, предлагаемый по умолчанию, может оказаться непригодным для разработки, особенно в тех случаях, когда объединяется несколько контейнеров. Сервер uWSGI также можно запускать в режиме разработки и отладки. При этом сохраняется возможность переключения в различные режимы работы, поэтому можно активизировать такие функциональные возможности uWSGI, как перезагрузку измененного кода в реальном времени для разработки и отладки, и отменять их в режиме эксплуатации.

Автоматизация с использованием Compose

Для упрощения работы можно добавить еще одну небольшую функцию автоматизации. Инструмент Docker Compose (http://docs.docker.com/compose/) предназначен для быстрой настройки и запуска различных вариантов сред разработки Docker. Важно отметить, что Compose использует файлы YAML (YAML Ain't Markup Language; язык сериализации данных) для хранения конфигурации групп контейнеров, позволяя разработчикам избавиться от рутинной и череватой опечатками ручной работы и не тратить сил на дублирование уже существующего решения. В настоящий момент предельная простота нашего приложения не позволяет по достоинству оценить преимущества автоматизации, но приложения обычно очень быстро становятся большими и сложными. Compose освободит нас от необходимости сопровождать все вспомогательные скрипты, предназначенные для организации работы, включая запуск, установление соединений, обновление и остановку контейнеров.

Если Docker установлен через Toolbox, то Compose уже доступен на вашей системе. В противном случае выполните инструкции по установке Compose с вебсайта Docker (http://docs.docker.com/compose/install/). В этой главе используется Compose версии 1.4.0, но поскольку применяется только основная функциональность, подойдут все версии после 1.2.

В каталоге *identidock* создадим файл *docker-compose.yml* со следующим содержимым:

```
identidock: ❶
  build: . ❷
  ports: ❸
   - "5000:5000"
  environment: ❹
   ENV: DEV
  volumes: ❺
   - ./app:/app
```

❶ В первой строке объявляется имя создаваемого контейнера. В одном YAML-файле можно определить несколько контейнеров (часто называемых сервисами на диалекте Compose).

❷ Ключ `build` сообщает Compose, что образ для данного контейнера должен быть создан из Dockerfile, расположенного в текущем каталоге (`.`). Каждое определение контейнера должно содержать либо ключ `build`, либо ключ `image`. Для ключей `image` задается тег или идентификатор образа, из которого создается контейнер, по аналогии с аргументом `image` в команде `docker run`.

❸ Ключ `ports` полностью аналогичен аргументу `-p` в команде `docker run` и служит для объявления открытых портов. Здесь связывается порт 5000 в контейнере с портом 5000 на хосте. Кавычки не обязательны, но рекомендуется их использование, чтобы избежать вероятной путаницы при синтаксическом разборе YAML-выражений, например 56:56 как числа в шестидесятиричной системе.

❹ Ключ `environment` полностью аналогичен аргументу `-e` в команде `docker run` и служит для определения значений переменных среды в контейнере. Здесь для переменной `ENV` определяется значение `DEV`, чтобы запустить веб-сервер Flask в режиме разработки.

❺ Ключ `volumes` полностью аналогичен аргументу `-v` в команде `docker run` и служит для определения томов. Здесь определено монтирование подкаталога *app* из текущего каталога на каталог */app* внутри контейнера точно так же, как мы делали раньше, чтобы можно было вносить изменения в код прямо с хоста.

В YAML-файлах для Compose можно определять многие другие ключи, которые в большинстве своем полностью соответствуют аргументам команды `docker run`.

Теперь при запуске команды `docker-compose up` будет получен в точности тот же результат, который наблюдался после выполнения предыдущей команды `docker run`:

```
$ docker-compose up
Creating identidock_identidock_1...
Attaching to identidock_identidock_1
identidock_1 | Running Development Server
identidock_1 | * Running on http://0.0.0.0:5000/
identidock_1 | * Restarting with reloader
```

Проверим работу сервера из другого терминала:

```
$ curl localhost:5000
Hello Docker!
```

Для завершения работы этого приложения воспользуйтесь комбинацией клавиш **Ctrl**+**C**, чтобы остановить контейнер.

Для переключения в эксплуатационный режим сервера uWSGI необходимо изменить определения ключей `environment` и `ports` в YAML-файле. Можно отредактировать существующий файл *docker-compose.yml* или создать новый YAML-файл для режима эксплуатации и передавать требуемое имя файла в команду `docker-compose` с помощью флага `-f` или через переменную среды `COMPOSE_FILE`.

Порядок работы Compose

При работе с Compose наиболее часто используются следующие команды. Назначение большинства из них легко определить по имени, почти все они имеют аналоги среди команд механизма Docker, тем не менее краткие описания этих команд приведены ниже.

`up`

Запуск всех контейнеров, определенных в Compose-файле. Вывод журнальных записей объединяется в один поток. В большинстве случаев используется аргумент `-d` для запуска Compose в фоновом режиме.

`build`

Пересоздание всех образов, созданных из файлов Dockerfile. Команда `up` будет создавать образы, только если они не существовали ранее, поэтому команду `build` следует использовать при необходимости обновления образов.

`ps`
: Вывод информации о состоянии контейнеров, управляемых Compose.

`run`
: Одноразовый запуск контейнера с выполнением одной команды (не в качестве сервиса). Также запускаются все контейнеры, с которыми должны быть установлены соединения, если не задан аргумент `--no-deps`. (Команды, передаваемые через `run`, заменяют команды, определенные в файле конфигурации сервиса. Кроме того, по умолчанию команда `run` не создает портов, определенных в файле конфигурации сервиса.)

`logs`
: Вывод журнальных записей с цветной подсветкой, объединенный для всех контейнеров, управляемых Compose.

`stop`
: Останов контейнеров без их удаления.

`rm`
: Удаление остановленных контейнеров. Следует помнить о том, что аргумент `-v` позволяет удалить все тома, управляемые механизмом Docker.

Обычный порядок работы начинается с выполнения команды `docker-compose up -d` для запуска приложения. Команды `docker-compose logs` и `ps` могут использоваться для проверки состояния приложения и как вспомогательное средство при отладке.

После внесения изменений в исходный код нужно выполнить `docker-compose build`, затем `docker-compose up -d`. При этом будет создан новый образ и заменен работающий контейнер. Отметим, что Compose сохраняет все ранее существовавшие тома из старых контейнеров, таким образом, базы данных и кэши остаются неизменными при переходе к новым версиям контейнеров (это может привести к беспорядку, поэтому будьте осторожны при замене контейнеров). Если создание нового образа не требуется, но внесены изменения в Compose YAML-файл, то выполните команду `docker-compose up -d`, чтобы заменить контейнер на точно такой же, но с новыми настройками. Если нужно принудительно остановить весь механизм Compose и заново создать все контейнеры, то воспользуйтесь флагом `--force-recreate`.

После завершения сеанса работы с приложением выполните команду `docker-compose stop` для его остановки. Тот же самый комплект контейнеров будет повторно запущен при выполнении команды `docker-compose start` или `up`, если не был изменен исходный код. Для окончательного удаления набора контейнеров приложения используйте команду `docker-compose rm`.

Подробное описание всех команд Compose смотрите на странице справочного руководства сайта Docker https://docs.docker.com/compose/reference/.

Резюме

После изучения данной главы мы имеем в своем распоряжении полноценную рабочую программную среду и можем начать разработку приложения. Мы узнали:

- как использовать официальные образы для быстрого создания переносимого и гибко настраиваемого комплекта для разработки без дополнительной установки каких-либо инструментальных средств в системе хоста;
- как использовать тома для динамического внесения изменений в исходный код работающих контейнеров;
- как одновременно организовать и поддерживать среду разработки и эксплуатационную среду в одном контейнере;
- как использовать инструмент Compose для автоматизации процесса разработки.

Docker обеспечивает нас удобной средой разработки со всеми необходимыми инструментальными средствами. Вместе с тем мы можем оперативно тестировать создаваемые приложения в среде, имитирующей эксплуатационные условия.

Остается необходимость доработки многих аспектов, особенно касающихся тестирования и непрерывной интеграции/доставки, но мы будем рассматривать эти темы в нескольких следующих главах одновременно с продолжением разработки нашего приложения.

Глава 6

Создание простого веб-приложения

В этой главе мы превратим примитивную программу «Hello World!» в простое веб-приложение, которое генерирует уникальное изображение для пользователей при вводе ими некоторого произвольного текста. Такие изображения иногда называют *идентификационными пиктограммами* (*identicons*), они используются для идентификации пользователей при помощи уникального изображения, сгенерированного по имени пользователя или по IP-адресу. Результатом изучения главы будет работоспособное приложение с базовой функциональностью, которую мы будем развивать в следующих главах. В процессе создания этого приложения мы увидим, как нужно компоновать контейнеры Docker для создания полнофункциональной системы, и почему такой подход естественным образом приводит к использованию микросервисов.

Идентификационные пиктограммы, или идентиконы

Идентификационные пиктограммы (идентиконы) – это изображения, автоматически генерируемые по некоторому числовому значению, обычно по хэш-коду IP-адреса или по имени пользователя. Это своеобразное визуальное представление конкретного объекта для упрощения его идентификации. Вариантами использования могут быть идентификационные изображения для пользователей на каком-либо веб-сайте, получаемые посредством хэширования их имен или IP-адресов, а также значки для избранного (favicons) для веб-сайтов.

Идентификационные пиктограммы придумал и разработал Дон Парк (Don Park) в начале 2007 года, для того чтобы проще различать комментаторов своего блога. Исходный код остается доступным на соответствующей странице GitHub (https://github.com/donpark/identicon).

Позже появились альтернативные реализации с различными графическими стилями. Наиболее известными создателями идентификационных пиктограмм являются Stack Overflow и GitHub (рис. 6.1, слева), использующие автоматически сгенерированные идентиконы для пользователей, которые не установили какое-либо изображение для своей учетной записи. Stack Overflow пользуется идентиконами,

сгенерированными сервисом Gravatar (рис. 6.1, справа)[1]. GitHub генерирует собственные идентиконы.

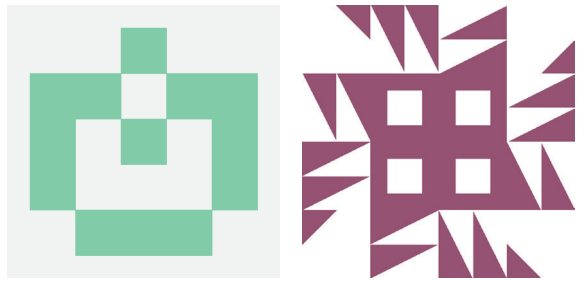

Рис. 6.1. Слева – типичная идентификационная пиктограмма сайта GitHub; справа – типичная идентификационная пиктограмма сервиса Gravatar

Если вы на практике выполняли все примеры из предыдущей главы, то должны были сформировать проект со следующей структурой:

```
identidock/
|__ Dockerfile
|__ app
|   |__ identidock.py
|__ cmd.sh
|__ docker-compose.yml
```

Но даже если этого проекта нет, не стоит беспокоиться. Вы можете взять нужный код со страницы данной книги на сайте GitHub (https://github.com/using-docker/creating-a-simple-web-app/). Например:

```
$ git clone -b v0 https://github.com/using-docker/creating-a-simple-web-app/
...
```

Можно также перейти на страницу версий в проекте GitHub для загрузки требуемых файлов.

Тег v0 обозначает версию кода, соответствующую концу предыдущей главы. В дальнейшем теги будут соответствовать обновлениям кода, сделанным на протяжении данной главы.

Управление версиями

Предполагается, что читатель данной книги умеет работать с инструментальным средством для управления версиями Git, по крайней мере знает как записывать и извлекать версии исходного кода и клонировать репозитории. В следующей главе мы рассмотрим интеграцию Docker Hub с сервисами GitHub и BitBucket. При недостатке опыта работы с Git ознакомьтесь с бесплатным учебником https://try.github.io.

[1] Который, в свою очередь, среди прочих использует разработки проекта WP_Identicon (http://scott.sherrilmix.com/blog/blogger/wp_identicon/).

Создание основной веб-страницы

На первом этапе займемся созданием простейшей веб-страницы для нашего приложения. Для удобства будем возвращать HTML-код только в форме строки[1]. Заменим содержимое файла *identidock.py* следующим исходным кодом:

```
from flask import Flask

app = Flask(__name__)
default_name = 'Joe Bloggs'

@app.route('/')
def get_identicon():

    name = default_name

    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{}">
              <input type="submit" value="submit">
              </form>
              <p>You look like a:
              <img src="/monster/monster.png"/>
              '''.format(name)
    footer = '</body></html>'

    return header + body + footer

if __name__ == '__main__':
    app.run( debug=True, host='0.0.0.0' )
```

В действительности этот код делает не намного больше, чем примитивная программа «Hello World!». Здесь лишь изменен возвращаемый текст – он стал небольшой веб-страницей, содержащей форму для ввода имени пользователя. Функция `format` заменяет подстроку `"{}"` реальным значением переменной `name`, которой пока просто присваивается имя по умолчанию «Joe Bloggs».

После выполнения команды `docker-compose up -d` и обращения в браузере по адресу http://localhost:5000 можно увидеть страницу, показанную на рис. 6.2.

Рис. 6.2. Вид сервиса identidock на первом этапе разработки

[1] Более эффективным решением было бы использование механизма шаблонов, подобного Jinja2, входящего в состав сервера Flask.

Отсутствие изображения вполне ожидаемо, так как мы пока не добавили код для его генерации. Кроме того, не работает кнопка отправки введенного текста (submit).

На этом этапе разработки следовало бы принять разумное решение, обеспечивающее автоматизированное тестирование и, возможно, даже непрерывную интеграцию/доставку. Но, для того чтобы подробно объяснить каждый шаг, разработка будет продолжена без тестирования и непрерывной интеграции, которые будут постепенно вводиться в последующих главах.

Использование преимуществ существующих изображений

Пришло время заставить нашу программу выдавать какой-либо реальный результат. Для этого необходимы функция или сервис, принимающие строку и возвращающие уникальное изображение. Эта функция или сервис будут вызываться с именем пользователя, переданным с веб-страницы, для выполнения замены шаблона отсутствующего изображения реальным изображением.

В нашем случае воспользуемся существующим Docker-сервисом изображений *dnmonster*. Если говорить в общих чертах, то сервис dnmonster предоставляет прикладной программный интерфейс RESTful. Этот сервис можно без затруднений заменить другим механизмом генерации идентиконов, особенно если он поддерживает интерфейс RESTful и может быть размещен в контейнере.

Для вызова сервиса необходимо внести соответствующие изменения в существующий код, и прежде всего нужно добавить новую функцию `get_identicon`:

```
from flask import Flask, Response ❶
import requests ❷

app = Flask(__name__)
default_name = 'Joe Bloggs'


@app.route('/')
def mainpage():

    name = default_name

    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{}">
              <input type="submit" value="submit">
              </form>
              <p>You look like a:
              <img src="/monster/monster.png"/>
              '''.format(name)
    footer = '</body></html>'

    return header + body + footer
```

```
@app.route('/monster/<name>')
def get_identicon( name ):

    r = requests.get('http://dnmonster:8000/monster/' + name + '?size=80') ❸
    image = r.content

    return Response( image, mimetype='image/png' ) ❹

if __name__ == '__main__':
    app.run( debug=True, host='0.0.0.0' )
```

❶ Импорт из пакета Flask модуля Response, используемого для передачи изображений.
❷ Импорт библиотеки requests (http://docs.python-requests.org/en/latest/), используемой для организации диалога с сервисом dnmonster.
❸ Создание HTTP-запроса GET, отправляемого сервису dnmonster. Запрашивается имя идентификационного изображения как значение переменной name, при этом размер изображения должен быть равен 80 пикселам.
❹ Оператор return немного усложняется из-за использования функции Response, сообщающей серверу Flask о том, что возвращается изображение в формате PNG, а не HTML-код и не текст.

Далее необходимо внести некоторые изменения в Dockerfile, чтобы обеспечить в новом коде поддержку соответствующих библиотек:

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask==0.10.1 uWSGI==2.0.8 requests==2.5.1 ❶
WORKDIR /app
COPY app /app
COPY cmd.sh /

EXPOSE 9090 9191
USER uwsgi

CMD ["/cmd.sh"]
```

❶ Добавляется библиотека requests, используемая в обновленном исходном коде.

Теперь все готово для запуска контейнера dnmonster и установления соединения с ним из контейнера нашего приложения. Для лучшего понимания того, что происходит при этом, сейчас мы выполним обычные команды Docker, а инструментом Compose воспользуемся позже. Поскольку с образом dnmoster мы работаем впервые, его необходимо загрузить из репозитория Docker Hub:

```
$ docker build -t identidock .
...
$ docker run -d --name dnmonster amouat/dnmonster:1.0
Unable to find image 'amouat/dnmonster:1.0' locally
(Образ 'amouat/dnmonster:1.0' не найден на локальной системе)
1.0: Pulling from amouat/dnmonster
(1.0: Загрузка из amouat/dnmonster)
```

```
...
Status: Downloaded newer image for amouat/dnmonster:1.0
(Состояние: Загружена более новая версия образа amouat/dnmonster:1.0)
e695026b14f7d0c48f9f4b110c7c06ab747188c33fc80ad407b3ead6902feb2d
```

Запуск контейнера приложения выполняется почти так же, как и в предыдущей главе, но на этот раз добавляется аргумент `--link dnmonster:dnmonster` для установления соединения между контейнерами. Это позволяет обращаться к URL http://dnmonster:8080/ непосредственно в исходном коде Python:

```
$ docker run -d -p 5000:5000 -e "ENV=DEV" --link dnmonster:dnmonster identidock
16ae698a9c705587f6316a6b53dd0268cfc3d263f2ce70eada024ddb56916e36
```

Подробнее о соединениях см. раздел «Соединение между контейнерами» (глава 4).

Если снова открыть браузер и перейти по адресу http://localhost:5000, то получим нечто похожее на рис. 6.3.

Рис. 6.3. Первое идентификационное изображение

Выглядит не слишком впечатляюще, но ведь мы только что сгенерировали самую первую идентификационную пиктограмму. Кнопка отправки введенного текста все еще не работает, и пользователь лишен возможности ввести свое имя, но мы исправим это в ближайшее время. Теперь вернемся к использованию Compose, чтобы избавиться от необходимости запоминания предыдущих команд `docker run`. В файл *docker-compose.yml* вносятся следующие изменения:

```
identidock:
  build .
  ports:
   - "5000:5000"
  environment:
    ENV: DEV
  volumes:
   - ./app:/app
  links: ❶
   - dnmonster

dnmonster: ❷
  image: amouat/dnmonster:1.0
```

❶ Объявление установления соединения из контейнера identidok с контейнером dnmonster. Compose обеспечивает правильный порядок запуска контейнеров для установления соединения.
❷ Определение нового контейнера dnmonster. Здесь достаточно сообщить Compose о необходимости использования образа `amouat/dnmonster:1.0` из репозитория Docker Hub.

Перед запуском приложения мы должны удалить все старые контейнеры:

```
$ docker rm $(docker stop ps -q)
 ...
```

Обратите внимание, что эта команда остановит все запущенные контейнеры, не только identidock. Затем мы можем пересоздать и запустить приложение при помощи Compose:

```
$ docker-compose build
 ...
$ docker-compose up -d
 ...
```

После этого для работающего приложения можно обновлять исходный код без перезапуска контейнеров.

Чтобы обеспечить правильное функционирование кнопки, необходимо обработать POST-запрос к серверу и использовать переменную `form` (содержащую имя пользователя) для генерации требуемого изображения. Кроме того, мы немного усложним код и выполним хэширование данных, введенных пользователем. Это позволит в некоторой степени сохранить анонимность предоставляемых данных, таких как адреса электронной почты, и привести эти данные в форму, подходящую для записи в URL (не нужно будет экранировать такие символы, как пробелы и т. п.). В нашем учебном приложении хэширование не столь важно, так как мы работаем только с именами, но данный пример показывает, как использовать этот сервис в других сценариях для защиты более важной информации, предоставляемой пользователями.

В файл *identicon.py* внесите следующие изменения:

```
from flask import Flask, Response, request
import requests
import hashlib ❶

app = Flask(__name__)
salt = "UNIQUE_SALT" ❷
default_name = 'Joe Bloggs'

@app.route('/', methods=['GET', 'POST']) ❸
def mainpage():

    name = default_name
    if request.method == 'POST': ❹
    name = request.form['name']
```

```
    salted_name = salt + name
    name_hash = hashlib.sha256( salted_name.encode() ).hexdigest() ❺

    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{0}">
              <input type="submit" value="submit">
              </form>
              <p>You look like a:
              <img src="/monster/{1}"/>
              '''.format( name, name_hash ) ❻
    footer = '</body></html>'

    return header + body + footer

@app.route('/monster/<name>')
def get_identicon( name ):

    r = requests.get( 'http://dnmonster:8080/monster/' + name + '?size=80' )
    image = r.content

    return Response( image, mimetype='image/png' )

if __name__ == '__main__':
    app.run( debug=True, host='0.0.0.0' )
```

❶ Импорт библиотеки, которая будет использоваться для хэширования данных, вводимых пользователем. Так как это стандартная библиотека, для ее установки вносить изменения в Dockerfile не нужно.

❷ Определение значения переменной `salt`, используемой в хэш-функции. При изменении этого значения разные сайты могут генерировать различные идентификационные пиктограммы для одних и тех же входных данных.

❸ По умолчанию сервер Flask определяет пути только для ответов на HTTP-запросы `GET`. Форма в исходном коде отправляет HTTP-запрос `POST`, поэтому необходимо добавить именованный аргумент `methods` к списку путей и в явной форме определить, что в данном случае будут обрабатываться оба запроса `POST` и `GET`.

❹ Если значение `request.method` равно значению `"POST"`, то этот запрос является результатом щелчка по кнопке подтверждения отправки данных (submit). В этом случае необходимо обновить переменную `name`, присвоив ей текстовое значение, введенное пользователем.

❺ Выполнение операции хэширования введенных данных с использованием алгоритма SHA256.

❻ Изменение URL изображения с учетом полученного выше хэшированного значения. При попытке загрузки файла изображения браузер будет вызывать функцию `get_identicon`, учитывая заданный путь и это хэшированное значение.

После сохранения новой версии исходного кода в файле *identicon.py* отладочный веб-сервер Python должен обнаружить изменения и автоматически перезагрузиться. Теперь можно наблюдать полноценную работу текущей версии нашего веб-приложения и видеть изображения, генерируемые для вводимых имен (см. рис. 6.4).

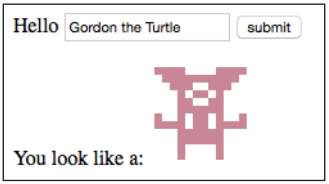

Рис. 6.4. Идентификационная пиктограмма для имени Gordon the Turtle

dnmonster

Сервис dnmonster представляет собой приложение, написанное с помощью Node.js и упакованное в Docker-контейнер. Это адаптация приложения Кевина Годена (Kevin Gaudin) monsterid.js (https://github.com/KevinGaudin/monsterid.js), перенесенная из программной браузерной среды JavaScript в программную среду Node.js. В свою очередь, приложение monsterid.js основано на программе Андреаса Гора (Andreas Gohr) MonsterID (http://www.splitbrain.org/projects/monsterid), которая создает вычисляемые 8-битовые изображения монстров в стиле RetroAvatar (http://retroavatar.appspot.com/). Исходный код приложения dnmonster размещен на GitHub (https://github.com/amouat/dnmonster).

В отличие от monsterid.js, сервис dnmonster не выполняет хэширования введенных данных, предоставляя возможность реализации этой операции вызывающей стороне (рис. 6.5).

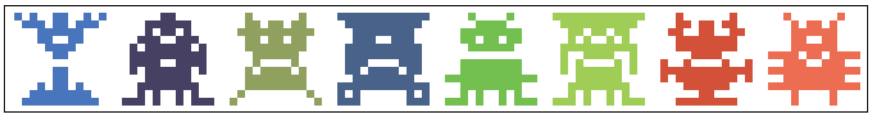

Рис. 6.5. Примеры идентификационных пиктограмм, сгенерированных сервисом dnmonster

Дополнительное кэширование

Пока все идет как нужно. И все же у текущей версии приложения есть один «ужасный» недостаток (монстры здесь ни при чем) – при каждом запросе изображения происходит весьма неэффективное, с точки зрения расхода вычислительных ресурсов, обращение к сервису dnmonster. Но в этом нет постоянной необходимости – основная идея использования идентификационных пиктограмм состоит в том, что для конкретных введенных данных (имени) изображение остается одним и тем же, поэтому следовало бы кэшировать первый полученный результат.

Для этой цели будет использоваться Redis – организованное в оперативной памяти хранилище данных типа «ключ-значение». Такая схема хорошо подходит для задач, подобных нашей, когда объем хранимой информации невелик и нет причин для беспокойства о долговременном ее сохранении (если запись потеряна

или удалена, изображение просто генерируется повторно). Можно было бы добавить сервер Redis в существующий контейнер identidock, но проще и правильнее (технологически) разместить его в новом контейнере. При этом мы получаем доступ ко всем преимуществам официального образа Redis, уже подготовленного в репозитории Docker Hub, и избавляемся от неудобств, связанных с организацией работы нескольких процессов внутри контейнера.

Запуск нескольких процессов внутри контейнера

В подавляющем большинстве контейнеров запускается только один процесс. Когда необходимо организовать работу нескольких процессов, наилучшим решением являются запуск нескольких контейнеров и установление соединений между ними, как показано в примере, рассматриваемом в текущем разделе.

Тем не менее иногда без нескольких процессов в одном контейнере обойтись невозможно. В таких случаях лучше всего воспользоваться менеджером процессов, например supervisord (http://supervisord.org/) или runit (http://smarden.org/runit/), для управления запуском и контроля выполнения процессов. Кроме того, можно написать простой скрипт инициализации требуемых процессов, но при этом следует помнить, что в этом случае только на вас возлагается ответственность за корректный останов и удаление процессов, а также за правильную обработку и перенаправление всех сигналов.

Более подробно об использовании supervisord в контейнерах можно узнать из статьи на сайте Docker (https://docs.docker.com/articles/using_supervisord/).

Чтобы использовать кэш, в первую очередь необходимо изменить исходный код:

```
from flask import Flask, Response, request
import requests
import hashlib
import redis ❶

app = Flask(__name__)
cache = redis.StrictRedis( host='redis', port=6379, db=0 ) ❷
salt = "UNIQUE_SALT"
default_name = 'Joe Bloggs'

@app.route('/', methods=['GET', 'POST'])
def mainpage():

    name = default_name
    if request.method == 'POST':
        name = request.form['name']

    salted_name = salt + name
    name_hash = hashlib.sha256( salted_name.encode() ).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
```

```
            Hello <input type="text" name="name" value="{0}">
            <input type="submit" value="submit">
            </form>
            <p>You look like a:
            <img src="/monster/{1}"/>
            '''.format( name, name_hash )
    footer = '</body></html>'

    return header + body + footer

@app.route('/monster/<name>')
def get_identicon( name ):

    image = cache.get( name ) ❸
    if image is None: ❹
        print( "Cache miss (промах кэша)", flush=True ) ❺
        r = requests.get( 'http://dnmonster:8080/monster/' + name + '?size=80' )
        image = r.content
        cache.set( name, image ) ❻

    return Response( image, mimetype='image/png' )

if __name__ == '__main__':
    app.run( debug=True, host='0.0.0.0' )
```

❶ Импорт модуля Redis.
❷ Настройка кэша Redis. Мы будем использовать соединения Docker, чтобы обеспечить доступность имени хоста `redis`.
❸ Проверка наличия текущего значения переменной `name` в кэше.
❹ При промахе кэша (то есть при отсутствии текущего значения в кэше) Redis возвращает значение `None`. В этом случае изображение генерируется как обычно, а кроме того...
❺ Выводится некоторая отладочная информация об отсутствии кэшированного изображения и...
❻ Сгенерированный образ добавляется в кэш и связывается с заданным именем.

Поскольку теперь используются новый модуль и новый контейнер, придется внести изменения в файлы *Dockerfile* и *docker-compose.yml*. Обновленная версия *Dockerfile*:

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask==0.10.1 uWSGI==2.0.8 requests==2.5.1 redis==2.10.3 ❶
WORKDIR /app
COPY app /app
COPY cmd.sh /

EXPOSE 9090 9191
USER uwsgi

CMD ["/cmd.sh"]
```

❶ Необходимо установить библиотеку поддержки клиента Redis для языка Python.

Обновленная версия *docker-compose.yml*:

```
identidock:
  build: .
  ports:
   - "5000:5000"
  environment:
    ENV: DEV
  volumes:
    - ./app:/app
  links:
    - dnmonster
    - redis ❶

dnmonster:
  image: amouat/dnmonster:1.0

redis:
  image: redis:3.0 ❷
```

❶ Директива установления соединения с контейнером Redis.
❷ Создание контейнера Redis на основе официального образа.

После остановки работающего приложения identidock можно выполнить команды `docker-compose build` и `docker-compose up`, чтобы запустить новую версию. Поскольку мы не внесли никаких изменений, влияющих на функциональность приложения, видимый результат работы новой версии не будет отличаться от предыдущего. Чтобы окончательно убедиться в том, что работает именно новый код, можно проверить поток вывода отладочной информации или, если есть возможность, применить инструменты контроля и наблюдения, например Prometheus, описанный в главе 10, и собственными глазами увидеть то, что происходит при выполнении приложения.

Микросервисы

Разработка приложения identidock происходила в соответствии с архитектурными принципами *микросервиса* (*microservice*), когда системы компонуются из нескольких небольших и независимых сервисов. Такой подход часто противопоставляется *монолитным* (*monolithic*) *архитектурам*, где система является составной частью одного крупномасштабного сервиса. Несмотря на то что identidock представляет собой упрощенное учебное приложение, на его примере четко прослеживаются разнообразные характеристики микросервисного подхода.

Если бы мы применили монолитную архитектуру, то пришлось бы разрабатывать аналоги dnmonster, Redis и identidock на одном языке программирования и запускать их как единый компонент в одном контейнере. При правильном проектировании монолитному приложению следовало бы разделить эти компоненты на отдельные библиотеки и по возможности пользоваться уже существующими библиотеками.

Мы обратились к противоположному подходу, при котором приложение identidock состоит из веб-приложения на языке Python, обменивающегося данными с сервисом на языке JavaScript и с хранилищем данных типа «ключ-значение», написанным на C, при этом компоненты распределены по трем контейнерам. При изучении следующих глав книги вы узнаете, как с минимальными трудозатратами подключить к identidock большее количество сервисов, в том числе *обратный прокси-сервер* (*reverse proxy*) в главе 9 и средства наблюдения и фиксации событий в журналах в главе 10.

Технология микросервисов обладает рядом преимуществ. Намного проще масштабировать программную среду микросервисов, распространяя ее на большее количество сетевых компьютеров. Отдельные микросервисы можно быстро и просто заменять на более производительные аналоги, а также без затруднений откатываться на предыдущие варианты при возникновении непредвиденных проблем в остальных частях системы. В независимых друг от друга микросервисах можно использовать различные языки программирования, что позволяет разработчикам выбирать языки, наилучшим образом соответствующие конкретным задачам.

Описываемый подход не лишен недостатков, в первую очередь это дополнительные накладные расходы для всех распределенных компонентов, так как обмен данными происходит по сети, а не посредством вызовов библиотечных функций. Приходится пользоваться дополнительными инструментальными средствами, подобными Compose, чтобы обеспечить правильный порядок запуска компонентов и корректное установление соединений между ними. Общая организация программной среды и обнаружение нужных сервисов становятся чрезвычайно важными задачами, требующими правильного решения.

Современные интернет-приложения способны извлекать огромные выгоды от возможностей динамического масштабирования, предоставляемых микросервисами, и это подтверждено на практике такими известными компаниями, как Netflix, Amazon и SoundCloud. Поэтому микросервисы уже фактически представляют собой быстро развивающуюся весьма важную и значимую архитектуру, которая тем не менее, как это обычно бывает, не претендует на роль универсальной «серебряной пули»[1].

Резюме

Теперь у нас есть полноценно работающая версия веб-приложения. Она остается весьма простой и все же обладает достаточной функциональностью, чтобы использовать несколько контейнеров и наглядно демонстрировать принципы разработки с помощью контейнеров. Вы научились многократно использовать существующие образы как в качестве основы для создания собственного образа (базовый образ

[1] Более подробно о преимуществах и недостатках микросервисов можно узнать из цикла статей Мартина Фаулера (Martin Fowler) по этой теме, особенно из статьи «Microservices» (http://martinfowler.com/articles/microservices.html).

Python), так и в качестве «черных ящиков», предоставляющих некоторые сервисы (образ dnmonster).

Но более важно то, что вы своими глазами увидели, как использование контейнеров естественным образом приводит к идее объединения небольших, четко определенных сервисов, взаимодействующих друг с другом, в более крупную систему – к идее применения технологии микросервисов.

Глава 7

Распространение образов

После создания собственных образов у автора возникает естественное желание сделать их доступными для сотрудников, для серверов непрерывной интеграции ПО и/или для конечных пользователей. Существует несколько способов распространения образов: можно заново создавать их из соответствующих файлов Dockerfile, скачивать из реестров или использовать команду `docker load` для установки образа из архивного файла.

В этой главе мы более подробно рассмотрим различия между этими способами и выберем наилучшие варианты организации распространения образов как внутри замкнутой группы сотрудников, так и для большой группы независимых конечных пользователей. Кроме того, будут описаны операции присваивания тегов и выгрузки образа identidock, для того чтобы им могли воспользоваться другие участники нашего рабочего процесса, а также для предоставления возможности его загрузки другими пользователями.

Исходные коды для этой главы доступны на странице книги в репозитории GitHub. Тег v0 определяет состояние кода в конце предыдущей главы, а следующие теги соответствуют версиям кода, получаемым в процессе дальнейшей разработки учебного примера в данной главе. Для получения начальной версии кода выполните команду:

```
$ git clone -b v0 https://github.com/using-docker/image-dist/
...
```

Кроме того, можно скачать код, соответствующий любому тегу, со страницы Releases проекта в репозитории GitHub (https://github.com/using-docker/image-dist/).

Именование образов и репозиториев

В разделе «Работа с реестрами» главы 3 описывалось, как правильно присваивать теги образам и выгружать образы в удаленные репозитории. При распространении образов очень важно использовать точные и содержательные имена и теги. Еще раз отметим, что имена и теги присваиваются при создании образа или устанавливаются специальной командой `docker tag`:

```
$ cd identidock
$ docker build -t "identidock:0.1" . ❶
$ docker tag "identidock:0.1" "amouat/identidock:0.1" ❷
```

❶ Определение имени репозитория `identidock` и тега `0.1`.
❷ Устанавливается соответствие имени `amouat/identidock` с образом, который ссылается на имя пользователя `amouat` в репозитории Docker Hub.

Опасность использования тега latest

Не следует слишком часто применять тег `latest`. Docker использует этот тег по умолчанию, если явно не задан какой-либо другой тег, но `latest` не содержит полезной информации. Во многих репозиториях этот тег служит обозначением самой свежей стабильной версии образа, но это совершенно не обязательно, поскольку определяется лишь неофициальным соглашением.

Образы с тегом `latest`, как и все прочие образы, не будут обновляться автоматически при выгрузке новой версии в реестр – остается необходимость явного выполнения команды `docker pull` для получения обновленных версий.

Если в команде `docker run` или `docker pull` указано имя образа без тега, то Docker использует образ с тегом `latest`, а при отсутствии такого образа сообщит об ошибке.

Тег `latest` слишком часто вводит в заблуждение пользователей, поэтому рекомендуется полностью отказаться от его применения, особенно в открытых массовых репозиториях.

При именовании тегов необходимо соблюдать несколько правил. Имя тега должно состоять из букв верхнего и нижнего регистров, чисел и символов «точка» (.) и «дефис» (-). Длина имени тега – от 1 до 128 символов. Запрещено использовать в качестве первого символа имени точку и дефис.

Имена репозиториев и тегов чрезвычайно важны при организации процесса разработки. Механизм Docker устанавливает минимальные ограничения для допустимых имен и позволяет создавать и удалять имена в любое время. Поэтому группа разработки должна самостоятельно разработать приемлемую схему присваивания имен.

Docker Hub

Самое простое и очевидное решение, обеспечивающее общий доступ к разработанным образам, – использование Docker Hub. Docker Hub – реестр, работающий в режиме онлайн и поддерживаемый компанией Docker Inc. Этот реестр предоставляет свободные репозитории для открытых образов, но пользователи могут оплатить закрытые, защищенные репозитории.

Другие варианты закрытого хостинга

Docker Hub – это не единственный вариант при выборе хранилища частных закрытых репозиториев в облачной среде. На момент выпуска данной книги основным конкурентом является реестр quay.io, предлагающий чуть более расширенные функциональные возможности, по сравнению с Docker Hub, по вполне приемлемым ценам.

Образ identidock можно выгрузить в репозиторий без особых трудностей. Предположим, что у вас уже есть учетная запись в реестре Docker Hub[1], тогда эта операция выполняется прямо из командной строки:

```
$ docker tag identidock:latest amouat/identidock:0.1 ❶
$ docker push amouat/identidock:0.1 ❷
The push refers to a repository [docker.io/amouat/identidock] (len: 1)
(Операция выгрузки обращается к репозиторию [docker.io/amouat/identidock] (размер: 1))
76899e56d187: Image successfully pushed (Образ успешно выгружен)
...
0.1: digest: sha256:8aecd14cb97cc4333fdffe903aec1435a1883a44ea9f25b45513d4c2...
```

❶ В первую очередь необходимо создать псевдоним (алиас) для образа в пространстве имен Docker Hub. Требуемая обязательная форма `<username>/<repositoryname>`, где `<username>` – имя вашей учетной записи в реестре Docker Hub (в данном примере `amouat`), а `<repositoryname>` – имя репозитория, который должен существовать в реестре Hub. Кроме того, предоставляется возможность присваивания данному образу тега `0.1`.

❷ Выгрузка образа с использованием только что созданного псевдонима. Если указанный репозиторий не существовал ранее, то он создается, и выполняется выгрузка образа с учетом заданного тега.

Теперь образ identidock общедоступен, и любой пользователь может загрузить его командой `docker pull`.

При переходе на сайт Docker Hub можно найти свой репозиторий по URL, соответствующему определенной схеме, например https://registry.hub.docker.com/u/amouat/identidock/. Если войти в учетную запись, то предоставляется возможность выполнения разнообразных задач администрирования, таких как создание описания для репозитория, включение других пользователей в группу сотрудников и настройка параметров методики разработки webhook.

Если нужно обновить репозиторий, то достаточно повторить выполнение команд установки тега и выгрузки образа для требуемого образа. При использовании существующего тега предыдущий образ будет перезаписан. Это удобно, но что, если необходимо просто обновлять образы при внесении изменений в их исходный код? Такой вариант использования встречается весьма часто, поэтому в реестре Docker Hub реализована концепция *автоматических сборок* (*automated builds*).

Автоматические сборки

Настроим функцию автоматической сборки в реестре Docker Hub для образа identidock. После настройки Hub будет заново создавать образ identidock и сохранять его в соответствующем репозитории при каждом внесении изменений в исходный код. Для этого потребуется создать и настроить репозиторий исходного кода на GitHub или Bitbucket. Можно просто выгрузить исходный код, который написан нами к настоящему моменту, или «клонировать» официальный код, размещенный на странице книги в GitHub (https://github.com/using-docker/image-dist/).

---

[1] Если нет, создайте ее на сайте https://hub.docker.com/.

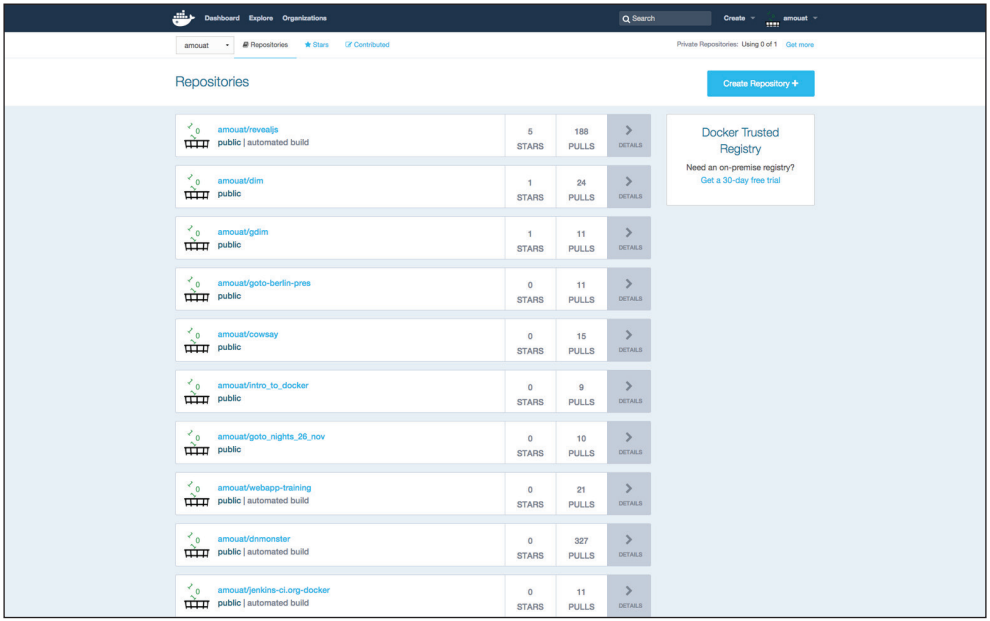

Рис. 7.1. Домашняя страница пользователя на сайте реестра Docker Hub

Автоматические сборки конфигурируются не из командной строки, а через веб-интерфейс сайта Docker Hub. После входа в свою учетную запись на этом сайте в правом верхнем углу должно появиться спускающееся меню с заголовком **Create** (Создать). Нужно выбрать пункт **Create Automated Build** (Создать автоматическую сборку) и указать расположение репозитория с исходным кодом приложения identidock[1]. После выбора репозитория исходного кода выводится страница параметров конфигурации для автоматической сборки. По умолчанию имя создаваемого репозитория совпадает с именем репозитория исходного кода, но его следует изменить на что-то более информативное, например `identidock_auto`. Рекомендуется добавить краткое описание репозитория, скажем «Automatic build for identidock» («Автоматическая сборка для identidock»). В поле первого тега оставьте без изменений значение `Branch`, затем введите имя `master`, чтобы отслеживать исходный код из главной ветви. В поле **Dockerfile Location** (Расположение Dockerfile) введите */identidock/Dockerfile*, если вы клонировали мой репозиторий исходного кода. Поле самого последнего тега определяет имя, присваиваемое данному образу в реестре Docker Hub. Можно оставить тег `latest` или заменить его на более информативный, например `auto`. После установки всех параметров щелкните по кнопке **Create** (Создать). Выводится страница сборки для нового репозитория. Первую сборку можно запустить, щелкнув по кнопке **Trigger a Build** (Начать

[1] Перед этим необходимо создать учетную запись на GitHub или Bitbucket, если вы не сделали этого раньше.

сборку). После завершения процедуры сборки можно скачать новую версию образа (если сборка завершилась успешно).

Протестировать функцию автоматизированной сборки можно посредством внесения небольших изменений в исходный код. В нашем случае добавим файл *README*, который Docker Hub также будет использовать для вывода информации об этом репозитории. В каталоге *identidock* создайте файл *README.md* с кратким описанием, например таким[1]:

```
identidock
==========
Simple identicon server based on monsterid from Kevin Gaudin.
(Простой сервер идентификационных пиктограмм на основе программы monsterid Кевина Годена.)
From "Using Docker" by Adrian Mouat published by O'Reilly Media.
(Из книги «Использование Docker», автор Эдриэн Моуэт, издательство O'Reilly Media.)
```

Сохраните файл и отправьте его в репозиторий исходного кода:

```
$ git add README.md
$ git commit -m "Added README"
[master d8f3317] Added README
 1 file changed, 6 insertions(+)
 create mode 100644 identidock/README.md
$ git push
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 456 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To git@github.com:using-docker/image-dist.git
   c81ff68..d8f3317  master -> master
```

Сделайте небольшую паузу, после чего перейдите на страницу сборки репозитория, где вы должны увидеть процесс или результат сборки новой версии образа.

Если по какой-либо причине в процессе сборки возникла ошибка, то можно попытаться выяснить ее причину по записям в журнале, щелкнув по кнопке **Build Code** (Код для сборки) на вкладке **Build Details** (Подробности сборки). Кроме того, в любой момент можно начать новую процедуру сборки с помощью кнопки **Trigger a Build** (Начать сборку).

Описанный выше способ создания и распространения образов подходит не для всех проектов. Создаваемые образы открыты для всех, если вы не оплатили закрытого репозитория, и вы находитесь в полной зависимости от работоспособности реестра Docker Hub – если сервер «падает», то вы лишаетесь возможности обновления своих образов, а пользователи не могут их загружать. Кроме того, возникает проблема производительности: при необходимости оперативно создавать

[1] Если вы клонировали существующий репозиторий исходного кода, то этот файл уже существует. В этом случае измените текст в существующем файле.

и перемещать образы через программный канал (pipeline) вы вряд ли согласитесь с ростом накладных расходов при передаче файлов из Docker Hub и вынужденным интервалом ожидания в очереди на сборку. Для проектов с открытым исходным кодом и для относительно небольших проектов Docker Hub выполняет свою задачу превосходно. Но для более крупномасштабных и более серьезных проектов вам, вероятнее всего, потребуются другие решения.

Распространение с ограничением доступа

Кроме использования реестра Docker Hub, существуют и другие возможности. Можно делать все вручную, выполняя операции экспорта и импорта образов или просто создавая новые версии образов из файлов Dockerfile на каждом Docker-хосте. Оба решения не являются оптимальными: создание образов из Dockerfile, повторяемое многократно, – процесс медленный, к тому же при этом могут возникать различия между образами на разных хостах. Процедуры экспортирования и импортирования образов в известной степени не надежны и являются источниками потенциальных ошибок. Остается одна возможность: использовать отдельный реестр, сопровождение которого обеспечивается самим автором или независимым провайдером.

Начнем с описания бесплатного решения – организация собственного реестра, после этого кратко рассмотрим некоторые коммерческие предложения.

Организация собственного реестра

Частный реестр Docker существенно отличается от официального реестра Docker Hub. Оба реестра поддерживают реализацию прикладного программного интерфейса, позволяющего пользователям выгружать, загружать и выполнять поиск образов, но Docker Hub – это удаленный сервис с закрытым исходным кодом, тогда как частный реестр представляет собой приложение с открытым исходным кодом, работающее на локальной системе. Кроме того, Docker Hub обеспечивает поддержку учетных записей пользователей, ведение статистики и веб-интерфейс, а в частном реестре Docker все эти функции отсутствуют.

Контроль за рабочим процессом

Версия реестра v2 является стабильной, но некоторые важные функциональные возможности все еще находятся в разработке. Поэтому в текущем разделе главное внимание уделено основным вариантам использования, без углубления в подробности применения более сложных функций. Полную, самую свежую версию документации по реестру можно найти на соответствующей странице проекта Docker в репозитории Git Hub (https://github.com/docker/distribution).

В этой главе рассматривается только версия 2 реестра, способная работать с Docker-демоном версии 1.6 и более поздних. Если необходима поддержка более старых версий Docker, то следует использовать предыдущую версию реестра (также возможна одновременная работа обеих версий реестра во время переходного периода). По сравнению с первой версией, версия 2 реестра предоставляет улуч-

шенные функциональные возможности по обеспечению безопасности, надежности и эффективности, поэтому настоятельно рекомендуется как можно скорее перейти к использованию версии 2.

Простейшим способом создания локального реестра является использование официального образа. Быстрый запуск реестра осуществляется командой:

```
$ docker run -d -p 5000:5000 registry:2
...
75fafd23711482bbee7be50b304b795a40b7b0858064473b88e3ddcae3847c37
```

Теперь у нас есть работающий реестр, и мы можем присваивать образам соответствующие теги и выгружать их в этот реестр. При использовании механизма `docker-machine` остается возможность указания адреса `localhost`, поскольку он правильно интерпретируется механизмом Docker (в отличие от клиента), который работает на том же хосте, что и реестр:

```
$ docker tag amouat/identidock:0.1 localhost:5000/identidock:0.1
$ docker push localhost:5000/identidock:0.1
The push refers to a repository [localhost:5000/identidock] (len: 1)
...
0.1: digest: sha256:d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc2885...
```

Если сейчас мы удалим локальную версию, то в любой момент можно извлечь ее из реестра:

```
$ docker rmi localhost:5000/identidock:0.1
Untagged: localhost:5000/identidock:0.1
$ docker pull localhost:5000/identidock:0.1
0.1: Pulling from identidock
(0.1: Загрузка из репозитория identidock)
...
76899e56d187: Already exists (Образ уже существует)
Digest: sha256:d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc28852e173108
Status: Downloaded newer image for localhost:5000/identidock:0.1
(Состояние: Загружен более новый образ для localhost:5000/identidock:0.1)
```

Механизм Docker обнаружил, что уже существует образ с тем же содержимым, поэтому в действительности происходит только повторное добавление тега. Обратите внимание на то, что реестр сгенерировал аутентификационный *дайджест* (*digest*) для этого образа. Это уникальное хэш-значение на основе содержимого образа и его метаданных. Образы можно извлекать из реестра по дайджесту, например:

```
$ docker pull localhost:5000/identidock@sha256:\
d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc28852e173108
sha256:d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc28852e173108: Pul...
...
76899e56d187: Already exists
Digest: sha256:d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc28852e173108
Status: Downloaded newer image for localhost:5000/identidock@sha256:d20affe5...
```

Главным преимуществом использования дайджеста является абсолютная гарантия того, что извлекается в точности тот образ, который нужен пользователю. При извлечении (загрузке) по тегу можно оказаться в ситуации, когда имя тегированного образа было изменено в какой-то момент, но пользователь не знает об этом. Кроме того, использование дайджестов обеспечивает целостность образа, и пользователь может быть уверен в том, что образ не был поврежден во время передачи или во время хранения. В разделе «Подтверждение происхождения образов» главы 13 более подробно рассматриваются обеспечение безопасности при работе с образами и процедура определения их происхождения.

Главное обоснование использования собственного частного реестра – необходимость организации централизованного хранилища для группы разработчиков или для всей организации. Это означает, что потребуется возможность загрузки образов из реестра, выполняемой удаленным демоном Docker. Но при попытке обращения извне к локальному реестру, который мы только что запустили, выводится следующее сообщение об ошибке:

```
$ docker pull 192.168.1.100:5000/identidock:0.1 ❶
Error response from daemon: unable to ping registry endpoint
(Демон вернул ошибку: не получен ping-ответ от целевого хоста реестра)
https://192.168.99.100:5000/v0/
v2 ping attempt failed with error: Get https://192.168.99.100:5000/v2/:
(v2 попытка ping – ошибка:)
tls: oversized record received with length 20527
(tls: получена запись с размером, превышающим допустимый, – длина 20527)
v1 ping attempt failed with error: Get https://192.168.99.100:5000/v1/_ping:
tls: oversized record received with length 20527
```

❶ Здесь аргумент `localhost` заменен на IP-адрес сервера. Ошибка будет возникать при любой попытке загрузки образа с помощью демона как на другом компьютере, так и на той же системе, где работает реестр.

Попробуем разобраться в том, что произошло. Демон Docker запретил соединение с удаленным хостом, так как этот хост не имеет действительного сертификата TLS (Transport Layer Security). До этого установление соединения разрешалось только потому, что в механизме Docker предусмотрено особое исключение для загрузки с серверов, расположенных на локальном хосте (то есть по адресу localhost). Возникшую проблему можно решить одним из следующих трех способов:

1. Перезапустить каждый демон Docker, которому требуется доступ к нашему реестру, с аргументом `--insecure-registry 192.168.1.100:5000` (разумеется, нужно указывать адрес и номер порта, который вы выбрали для своего сервера).
2. Установить на хосте реестра подписанный сертификат от аккредитованного центра сертификации, как во всех случаях, когда необходимо управлять доступом к веб-сайту по протоколу HTTPS.
3. Установить на хосте реестра самоподписанный сертификат и скопировать его на все хосты демонов Docker, которым должен быть предоставлен доступ к этому реестру.

Первый способ самый простой, но здесь он не рассматривается из соображений сохранения безопасности. Второй вариант является наилучшим, но требует получения сертификата от аккредитованного центра сертификации, за который обычно нужно заплатить. Третий способ обеспечивает безопасность, но для его реализации необходимо вручную выполнить операции копирования сертификата на хосты каждого демона.

Для создания собственного самоподписанного сертификата можно воспользоваться утилитой OpenSSL. Все операции должны быть выполнены на компьютере, который предполагается использовать в качестве сервера реестра в течение длительного времени. Сертификаты для данного примера протестированы на виртуальной машине Ubuntu 14.04, работающей в облачной инфраструктуре Digital Ocean, но в других операционных средах, вероятнее всего, будут получены другие результаты.

```
root@reginald:~# mkdir registry_certs
root@reginald:~# openssl req -newkey rsa:4096 -nodes -sha256 \
>     -keyout registry_certs/domain.key -x509 -days 365 \
      -out registry_certs/domain.crt  ❶
Generating a 4096 bit RSA private key
(Генерация 4096-битового закрытого ключа по алгоритму RSA)
.....................................................++
.....................................................++
writing new private key to 'registry_certs/domain.key'
(запись нового закрытого ключа в файл 'registry_certs/domain.key')
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
(Далее необходимо ввести информацию, которая будет включена в ваш запрос сертификата.)
What you are about to enter is what is called a Distinguished Name or a DN.
(Вы должны ввести так называемое имя для распознавания (DN).)
There are quite a few fields but you can leave some blank
(Некоторые поля можно оставить незаполненными)
For some fields there will be a default value,
(Для некоторых полей имеются значения по умолчанию,)
If you enter '.', the field will be left blank.
(При вводе символа точки '.' поле будет оставлено пустым.)
-----
Country Name (2 letter code) [AU]:
(Название страны (двухбуквенный код))
State or Province Name (full name) [Some-State]:
(Название штата или провинции (полностью))
Locality Name (eg, city) []:
(Название места расположения (например, город))
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
(Название организации (компании и т. п.))
Organizational Unit Name (eg, section) []:
(Название подразделения организации (например, отдела))
```

```
Common Name (e.g. server FQDN or YOUR name) []:reginald ❷
(Общедоступное открытое имя (например, имя FQDN-сервера или ВАШЕ имя))
Email Address []:
root@reginald:~# ls registry_certs/
domain.crt  domain.key ❸
```

❶ Создается самоподписанный сертификат x509 и 4096-битовый закрытый ключ по алгоритму RSA. Сертификат подписан с помощью дайджеста SHA256 и действителен в течение 365 дней. Далее OpenSSL запрашивает дополнительную информацию, и можно ввести требуемые ответы или оставить значения по умолчанию.

❷ Общедоступное открытое имя важно – оно должно совпадать с именем, используемым для доступа к серверу, при этом нельзя вводить IP-адрес (здесь `reginald` – имя моего сервера).

❸ После завершения процесса мы получаем файл сертификата *domain.crt*, который будет совместно использоваться всеми клиентами, и закрытый ключ *domain.key*, который следует хранить в безопасном месте, исключив возможность постороннего доступа к нему.

Обращение к реестру по IP-адресу

Если для обращения к реестру требуется использование IP-адреса, то процедура немного усложняется. IP-адрес нельзя применять вместо общедоступного открытого имени. Необходимо настроить альтернативные имена SAN (Subject Alternative Names) для IP-адреса или нескольких адресов, которые будут использоваться. Вообще говоря, я бы не рекомендовал такого подхода. Все-таки лучше выбрать имя своего сервера и обеспечить доступ по этому имени внутри системы (в худшем случае можно просто вручную добавить адрес и имя сервера в файл */etc/hosts*). Это гораздо проще сделать, к тому же не потребуется операция замены тегов для всех образов при изменении IP-адреса.

Теперь необходимо скопировать сертификат в каждую систему демона Docker, которому потребуется доступ к реестру[1]. Копирование должно быть выполнено в файл */etc/docker/certs.d/<адрес_реестра>/ca.crt*, где *<адрес_реестра>* – это адрес (имя) и номер порта конкретного сервера реестра. Также потребуется перезапустить демон Docker. Например:

```
root@reginald:~# sudo mkdir -p /etc/docker/certs.d/reginald:5000
root@reginald:~# sudo cp registry_certs/domain.crt \
                 /etc/docker/certs.d/reginald:5000/ca.crt  ❶
root@reginald:~# sudo service docker restart
docker stop/waiting
docker start/running, process 3906
```

❶ Для запуска на удаленном хосте необходимо передать CA-сертификат на хост Docker с помощью утилиты `scp` или аналогичной ей. При использовании открытого, заверенного CA-сертификата этот шаг можно пропустить.

[1] Этот шаг можно пропустить, если вы используете сертификат, подписанный аккредитованным центром сертификации.

Теперь можно запустить реестр[1]:

```
root@reginald:~# docker run -d -p 5000:5000 \
            -v $(pwd)/registry_certs:/certs \  ❶
            -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
            -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \  ❷
            --restart=always --name registry registry:2
...
b79cb734d8778c0e36934514c0a1ed13d42c342c7b8d7d4d75f84497cc6f45f4
```

❶ Размещение в контейнере сертификатов как тома.
❷ Можно определить переменные среды для настройки конфигурации реестра, которая позволяет использовать созданные ранее сертификаты.

Выполним операции извлечения (загрузки) образа, замены его тега и возврата (выгрузки) обратно в реестр, чтобы убедиться в работоспособности новой версии реестра:

```
root@reginald:~# docker pull debian:wheezy
wheezy: Pulling from library/debian
ba249489d0b6: Pull complete
19de96c112fc: Pull complete
library/debian:wheezy: The image you are pulling has been verified.
Important: image verification is a tech preview feature and should not be
relied on to provide security.
Digest: sha256:90de9d4ecb9c954bdacd9fbcc58b431864e8023e42f8cc21782f2107054344e1
Status: Downloaded newer image for debian:wheezy
root@reginald:~# docker tag debian:wheezy reginald:5000/debian:local  ❶
root@reginald:~# docker push reginald:5000/debian:local
The push refers to a repository [reginald:5000/debian] (len: 1)
19de96c112fc: Image successfully pushed
ba249489d0b6: Image successfully pushed
local: digest: sha256:3569aa2244f895ee6be52ed5339bc83e19fafd713fb1138007b987...
```

❶ Имя "reginald" необходимо заменить на имя вашего сервера реестра.

Итак, мы получили реестр с возможностью удаленного доступа к нему с обеспечением безопасной работы и безопасного хранения образов. При тестировании реестра с удаленных компьютеров не забудьте скопировать сертификат в файл */etc/docker/certs.d/<адрес_реестра>/ca.crt* на компьютерах с работающими механизмами Docker и проверьте, может ли механизм Docker корректно определить адрес нового реестра[2].

---

[1] Перед этим рекомендуется удалить все ранее запущенные экземпляры реестра.

[2] Нельзя заменять имя реестра каким-либо IP-адресом, так как это приведет к критической ошибке из-за несоответствия сертификату. Вместо этого необходимо отредактировать файл */etc/hosts* или настроить сервис доменных имен DNS, чтобы обеспечить правильное преобразование имен.

Для Docker существует огромное количество параметров конфигурации, которые можно применять для общей начальной и более тонкой настройки собственного реестра в конкретном варианте использования. Параметры настройки реестра сохраняются внутри образа в YAML-файле, который может быть заменен томом. Кроме того, значения параметров могут быть заменены во время выполнения путем присваивания значений переменным среды, как в предыдущем примере, где были определены переменные среды `REGISTRY_HTTP_TLS_KEY` и `REGISTRY_HTTP_TLS_CERTIFICATE`. Во время написания книги файл конфигурации размещался в каталоге */go/src/github.com/docker/distribution/cmd/registry/config.yml*, но, вероятнее всего, этот путь будет заменен на более простой. По умолчанию конфигурация определена для среды разработки, поэтому требуются значительные изменения для режима эксплуатации. Более подробное описание процесса конфигурирования реестра и примеры файлов конфигурации можно найти в дистрибутивной версии проекта на GitHub.

В следующих разделах описываются основные возможности и параметры, необходимые для настройки реестра.

Хранилище

По умолчанию образ реестра использует драйвер файловой системы, который вполне ожидаемо сохраняет все данные и образы в соответствующей файловой системе. Это удачный выбор для среды разработки и, возможно, приемлемый для многих вариантов эксплуатации. Необходимо объявить том в ранее определенном корневом каталоге и связать его с надежным хранилищем файлов. Например, включение следующего кода в файл *config.yml* настраивает реестр для использования работающего драйвера файловой системы и для сохранения данных в подкаталоге */var/lib/registry*, который должен быть объявлен как том:

```
storage:
    filesystem:
        rootdirectory: /var/lib/registry
```

Для сохранения данных в облачной среде можно воспользоваться драйвером Amazon S3 или Microsoft Azure.

Кроме того, предлагаются поддержка для хранилища распределенных объектов Ceph и использование Redis как кэша в оперативной памяти для ускорения уровня доступа.

Аутентификация

До сих пор мы рассматривали доступ к реестру по протоколу TLS, но оставили без внимания процедуру аутентификации пользователей. Возможно, это оправдано, если используются только открытые образы или реестр доступен только в частной закрытой сети, но большинству организаций необходим доступ, строго ограниченный только пользователями, прошедшими процедуру аутентификации.

Существуют два способа организации процедуры аутентификации:

1. Установить и настроить защищающий реестр прокси-сервера, например nginx, который будет отвечать за аутентификацию пользователей. Пример такой конфигурации приведен в официальной документации GitHub-проекта (https://docs.docker.com/registry/nginx/), где используются функциональные возможности nginx по аутентификации пользователь/пароль. После настройки можно использовать команду `docker login` для аутентификации в реестре.
2. Токен-аутентификация с использованием JSON Web Tokens. При применении этого метода реестр будет отказывать в обслуживании клиентам, не предъявившим корректного токена, но будет перенаправлять их на сервер аутентификации. Токены можно получить на сервере аутентификации, после чего клиент сможет получить доступ к реестру. Docker не предоставляет сервера аутентификации, и на момент публикации книги было известно единственное решение с открытым кодом от компании Cesanta Software (https://github.com/cesanta/docker_auth). Другие возможности в настоящее время: развернуть собственную библиотеку токенов на основе JSON Web Token или заплатить за одно из коммерческих решений, описанных ниже в разделе «Коммерческие реестры». Несмотря на очевидную сложность и трудоемкость настройки собственного сервиса аутентификации, этот вариант часто является более предпочтительным для крупных или распределенных организаций.

Протокол HTTP

В этом разделе описывается конфигурирование HTTP-интерфейса для реестра. Это важно для настройки правильной работы реестра. В частности, потребуется установить местонахождение TLS-сертификата и ключа для реестра, в предыдущем примере это сделано с помощью переменных среды `REGISTRY_HTTP_TLS_KEY` и `REGISTRY_HTTP_TLS_CERTIFICATE`.

Обычный файл конфигурации может выглядеть приблизительно так:

```
http:
    addr: reginald:5000  ❶
    secret: DD100CC4-1356-11E5-A926-33C19330F945  ❷
    tls:  ❸
        certificate: /certs/domain.crt
        key: /certs/domain.key
```

❶ Адрес реестра.

❷ Произвольная строка использована для заверения информации о состоянии, сохраненной клиентами. Она предназначена для защиты от поддельных сертификатов и ключей. В идеальном случае должно быть сгенерировано случайное значение.

❸ Настройка сертификатов, как в примерах, приводимых ранее. Указанные файлы должны быть доступными в контейнере реестра, или монтироваться как том, или копироваться в этот контейнер.

Прочие параметры настройки

Следует отметить большое разнообразие других параметров, которые можно использовать для настройки связующего (промежуточного) ПО, системы уведомлений, ведения журналов и кэширования. Более подробную информацию можно найти в вышеупомянутой официальной документации GitHub-проекта.

Коммерческие реестры

Если необходимо более солидное решение с возможностью управления через веб-интерфейс, то можно предложить Docker Trusted Registry (https://www.docker.com/docker-trusted-registry) и CoreOS Enterprise Registry (https://coreos.com/products/enterprise-registry/). Это полноценные коммерческие решения для локального использования, которое будет защищено сетевым экраном организации/предприятия.

Оба варианта предлагают широкий спектр функциональных возможностей, обеспечивающих удобное хранение образов, а также предоставляют инструментальные средства для работы с Docker-образами в группе, например инструменты для детального управления доступом и графический пользовательский интерфейс для операций установки, настройки и прочих задач администрирования.

Сокращение размера образа

Вы, вероятно, уже заметили, что Docker-образы могут иметь достаточно большой размер – многие образы достигают размера в несколько сотен мегабайт, что влечет за собой рост затрат времени на загрузку и выгрузку образов. В известной степени это компенсируется иерархической структурой образов: если у вас имеется базовый (родительский) уровень образа, то необходимо загрузить только новые уровни-потомки.

Тем не менее во многих случаях следует попытаться уменьшить размеры образов, хотя это не так просто сделать. Самым простым и очевидным решением кажется удаление ненужных файлов из образа. К сожалению, это не помогает. Вспомните о том, что образ формируется из нескольких уровней, причем каждый уровень создается отдельной командой из соответствующего файла Dockerfile и его родительских файлов Dockerfile. Общий конечный размер образа представляет собой сумму размеров всех его уровней. Удаление файла на одном уровне ничего не даст, так как он продолжает оставаться нетронутым на родительских уровнях. Чтобы убедиться в этом на конкретном примере, рассмотрим следующий Dockerfile:

```
FROM debian:wheezy

RUN dd if=/dev/zero of=/bigfile count=1 bs=50MB  ❶
RUN rm /bigfile
```

❶ Это простой и быстрый способ создания файла заданного размера.

Теперь создадим образ и исследуем его:

```
$ docker build -t filetest .
...
$ docker images filetest  ❶
REPOSITORY  TAG     IMAGE ID      CREATED       VIRTUAL SIZE
filetest    latest  e2a98279a101  8 second ago  135 MB
$ docker history filetest  ❷
IMAGE        ...  CREATED BY                                      SIZE      ...
e2a98279a101      /bin/sh -c rm /bigfile                          0 B
5d0f04380012      /bin/sh -c dd if=/dev/zero of=/bigfile count=   50 MB
c90d655b99b2      /bin/sh -c #(nop) CMD [/bin/bash]               0 B
30d39e59ffe2      /bin/sh -c #(nop) ADD file:3f1a40df75bc5673ce   85.01 MB
511136ea3c5a                                                      0 B
```

❶ Здесь можно видеть, что образ имеет общий размер 135 Мб, ровно на 50 Мб больше, чем основной образ.

❷ Команда `docker history` дает полную картину. Первые две строки описывают уровни, созданные приведенным выше файлом Dockerfile. Отчетливо видно, что команда `dd` создала уровень размером 50 Мб, а поверх него новый уровень создала команда `rm`.

Для сравнения рассмотрим несколько измененный Dockerfile:

```
FROM debian:wheezy

RUN dd if=/dev/zero of=/bigfile count=1 bs=50MB && rm /bigfile
```

Еще раз создадим и исследуем образ:

```
$ docker build -t filetest .
...
$ docker images filetest
REPOSITORY  TAG     IMAGE ID      CREATED         VIRTUAL SIZE
filetest    latest  40a9350a4fa2  34 seconds ago  85.01 MB
$ docker history filetest
IMAGE        ...  CREATED BY                                      SIZE      ...
40a9350a4fa2      /bin/sh -c dd if=/dev/zero of=/bigfile count=   0 B
c90d655b99b2      /bin/sh -c #(nop) CMD [/bin/bash]               0 B
30d39e59ffe2      /bin/sh -c #(nop) ADD file:3f1a40df75bc5673ce   85.01 MB
511136ea3c5a                                                      0 B
```

В этом случае размер образа, по сравнению с основным, не увеличился. Если файл удаляется на том же уровне, на котором он создан, то такой файл не включается в образ. Поэтому достаточно часто встречаются файлы Dockerfile, которые загружают tar-архивы или архивы других форматов, распаковывают их и сразу же удаляют архивный файл в одной инструкции `RUN`. Например, в официальный образ MongoDB включена следующая инструкция (URL сокращен для удобства форматирования):

```
RUN curl -SL "https://$MONGO_VERSION.tgz" -o mongo.tgz \
    && curl -SL "https://$MONGO_VERSION.tgz.sig" -o mongo.tgz.sig \
```

```
    && gpg --verify mongo.tgz.sig \
    && tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
    && rm mongo.tgz*
```

Похожая методика применима и для исходного кода – иногда в одной строке можно увидеть команды загрузки, компиляции в бинарный код и удаления ненужных файлов.

По той же причине нет смысла выполнять операцию очистки после работы менеджера пакетов следующим образом:

```
RUN rm -rf /var/lib/apt/lists/*
```

Действительно «почистить» образ можно так (этот пример также взят из официального Dockerfile для mongo):

```
RUN apt-get update \
    && apt-get install -y curl numactl \
    && rm -rf /var/lib/apt/lists/*
```

Вы также можете вернуться к разделу «Основные образы» главы 4, где обсуждается правильный выбор образа с минимальным размером.

Существует еще одна возможность уменьшения размера образа в затруднительных положениях. Если выполнить операцию экспорта контейнера `docker export`, затем применить к результату команду `docker import`, то получится образ, содержащий только один уровень. Например:

```
$ docker create identidock:latest
fe165be64117612c94160c6a194a0d8791f4c6cb30702a61d4b3ac1d9271e3bf
$ docker export $(docker ps -lq) | docker import -
146880a742cbd0e92cd9a79f75a281f0fed46f6b5ece0219f5e1594ff8c18302
$ docker tag 146880a identidock:import
$ docker images identidock
REPOSITORY  TAG     IMAGE ID      CREATED        VIRTUAL SIZE
identidock  import  146880a742cb  5 minutes ago  730.9 MB
identidock  0.1     76899e56d187  23 hours ago   839.5 MB
identidock  latest  1432cc6c20e5  4 days ago     839 MB
$ docker history identidock:import
IMAGE         CREATED        CREATED BY  SIZE      COMMENT
146880a742cb  11 minutes ago             730.9 MB  Imported from -
```

Эта операция немного уменьшает размер образа, но для нее характерны следующие недостатки:

- необходимо повторно выполнить все инструкции Dockerfile, такие как `EXPOSE`, `CMD`, `PORTS`, которые не связаны с данной файловой системой;
- теряются все метаданные, связанные с исходным образом;
- после импорта невозможно использовать пространство совместно с другими образами, основанными на том же родительском образе.

Происхождение образов

При распространении и эксплуатации образов важно знать, как установить *происхождение* (*provenance*) образов, то есть источник, из которого они получены, и автора (авторов). При загрузке необходима уверенность в том, что данный образ действительно создан тем автором, который объявлен, что не была произведена подмена или искажение образа, что это в точности тот образ, который подготовил и протестировал его создатель.

Компания Docker предлагает решение под названием Docker content trust (https://docs.docker.com/security/trust/content_trust/), которое во время написания книги находилось в стадии тестирования, поэтому по умолчанию недоступно. Более подробно об этом см. в разделе «Подтверждение происхождения образов» главы 13.

Резюме

Эффективное распространение образов является важнейшим компонентом любого рабочего процесса с использованием Docker. В этой главе подробно рассматривались самые простые и очевидные решения этой задачи: реестр Docker Hub и частные реестры. Также обсуждались некоторые вопросы, связанные с распространением образов, в том числе необходимость правильного присваивания имен и тегов образам, а также возможности уменьшения размеров образов.

В следующей главе будет рассматриваться переход к следующему этапу рабочего процесса использования образов – к созданию и сопровождению сервера непрерывной интеграции.

Глава 8

Непрерывная интеграция и тестирование с использованием Docker

Из этой главы вы узнаете, как можно использовать Docker и Jenkins для создания рабочего потока *непрерывной интеграции* (*continuous integration – CI*) для создания и тестирования учебного приложения. Также здесь рассматриваются другие аспекты тестирования с применением Docker и дается краткое описание методики тестирования архитектуры микросервисов.

При тестировании контейнеров и микросервисов возникают некоторые достаточно серьезные трудности различного характера. Для микросервисов модульное тестирование (юнит-тестирование) выполняется просто, но при интеграционном тестировании и тестировании системы в целом возникают затруднения из-за постоянно увеличивающегося количества сервисов и сетевых соединений. Имитация сетевых сервисов становится более надежной, чем общепринятая, широко распространившаяся практика имитации классов в кодовой базе монолитных приложений на Java или C#. Сохранение кода тестов в образах позволяет поддерживать переносимость и целостность контейнеров, но увеличивает их размер.

Исходные коды для этой главы доступны на странице книги в репозитории GitHub. Тег `v0` определяет состояние кода в конце предыдущей главы, а следующие теги соответствуют версиям кода, получаемым в процессе дальнейшей разработки учебного примера в данной главе. Для получения начальной версии кода выполните команду:

```
$ git clone -b v0 https://github.com/using-docker/image-dist/
...
```

Кроме того, можно скачать код, соответствующий любому тегу, со страницы Releases проекта в репозитории GitHub (https://github.com/using-docker/image-dist/).

Включение модульных тестов в identidock

В первую очередь нужно добавить несколько модульных тестов в кодовую базу нашего приложения identidock. Эти тесты будут проверять основную функциональность исходного кода identidock, не затрагивая внешних сервисов[1].

Начнем с создания файла *identidock/app/tests.py* со следующим содержимым:

```
import unittest
import identidock

class TestCase( unittest.TestCase ):

    def setUp( self ):
        identidock.app.config["TESTING"] = True
        self.app = identidock.app.test_client()

    def test_get_mainpage( self ):
        page = self.app.post( "/", data=dict(name="Moby Dick") )
        assert page.status_code == 200
        assert 'Hello' in str( page.data )
        assert 'Moby Dick' in str( page.data )

    def test_html_escaping( self ):
        page = self.app.post( "/", data=dict(name='"><b>TEST</b><!--') )
        assert '<b>' not in str( page.data )

if __name__ == '__main__':
    unittest.main()
```

Это очень простой тестовый файл с тремя методами:

- `setUp` – инициализация тестовой версии веб-приложения с использованием сервера Flask;
- `test_get_mainpage` – тестирование метода, вызывающего URL / с передачей в поле `name` значения `"Moby Dick"`. Затем тест проверяет код возврата этого метода на равенство значению `200`, при этом полученные данные должны содержать строки `'Hello'` и `'Moby Dick'`;
- `test_html_escaping` – проверка правильности экранирования HTML-элементов в потоке ввода.

Теперь выполним эти тесты:

```
$ docker build -t identidock .
...
$ docker run identidock python tests.py
.F
======================================================================
```

[1] Многие разработчики являются приверженцами методики разработки через тестирование (test-driven development – TDD), при которой сначала пишутся тесты, а потом тестируемый код. В данной книге этот подход не используется, чтобы сохранить последовательный и понятный стиль изложения материала.

```
FAIL: test_html_escaping (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 19, in test_html_escaping
    assert '<b>' not in str(page.data)
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.010s

FAILED (failures=1)
```

Неудачное завершение. Первый тест прошел, но при выполнении второго возникла ошибка, так как при вводе пользовательских данных не обеспечено корректное экранирование. Это серьезная угроза безопасности, которая в крупномасштабных приложениях может привести к утечкам данных и создать возможность для внешних атак типа «межсайтовый скриптинг» (cross-site scripting – XSS). Чтобы наблюдать этот эффект при работе приложения, запустите identidock и попробуйте ввести, например, такое имя (включая двойные кавычки): `"><b>pwned!</b><!--"`. Атакующий получает потенциальную возможность включить вредоносный скрипт в наше приложение и обманным путем заставить пользователей запустить его.

К счастью, эту ошибку легко исправить. Нужно только добавить в исходный код приложения функцию соответствующей обработки пользовательского ввода, которая заменяет HTML-элементы и кавычки на escape-коды. После редактирования исходный код *identidock.py* должен выглядеть следующим образом:

```
from flask import Flask, Response, request
import requests
import hashlib
import redis
import html

app = Flask( __name__ )
cache = redis.StrictRedis( host='redis', port=6379, db=0 )
salt = "UNIQUE_SALT"
default_name = 'Joe Bloggs'

@app.route( '/', methods=['GET', 'POST'] )
def mainpage():

    name = default_name
    if request.method == 'POST':
        name = html.escape( request.form['name'], quote=True )  ❶

    salted_name = salt + name
    name_hash = hashlib.sha256( salted_name.encode() ).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{0}">
```

```
            <input type="submit" value="submit">
            </form>
            <p>You look like a:
            <img src="/monster/{1}"/>
            '''.format( name, name_hash )
    footer = '</body></html>'

    return header + body + footer

@app.route( '/monster/<name>' )
def get_identicon( name ):

    name = html.escape( name, quote=True )  ❶
    image = cache.get( name )
    if image is None:
        print( "Cache miss (Промах кэша)", flush=True )
        r = requests.get( 'http://dnmonster:8000/monster/' + name + '?size=80' )
        image = r.content
        cache.set( name, image )

    return Response( image, mimetype='image/png' )

if __name__ == '__main__':
    app.run( debug=True, host='0.0.0.0' )
```

❶ Метод `html.escape()` используется для преобразования введенных пользователем данных в требуемую форму.

Еще раз создадим и протестируем приложение:

```
$ docker build -t identidock .
...
$ docker run identidock python tests.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.009s

OK
```

Все проблемы решены. Это можно проверить, если перезапустить identidock в новых контейнерах (не забудьте выполнить команду `docker-compose build`) и вводить различные «некорректные» имена, которые могут создать угрозу безопасности[1]. Если бы мы использовали настоящий механизм обработки шаблонов вместо простого объединения строк, то экранирование выполнялось бы автоматически без нашего участия, что устранило бы наблюдаемую выше проблему.

---

[1] Неловко признаться, но я не обращал внимания на эту проблему до этапа рецензирования данной книги. В очередной раз получен урок: важно тестировать даже код, кажущийся простым и ясным, а лучше всего повторно использовать уже существующий, проверенный и апробированный код и инструментальные средства его сопровождения во всех случаях, когда это возможно.

Теперь, когда у нас есть несколько тестов, необходимо расширить возможности файла *cmd.sh*, чтобы обеспечить поддержку автоматического выполнения этих тестов. Отредактируйте содержимое *cmd.sh* следующим образом:

```
#!/bin/bash
set -e

if [ "$ENV" = 'DEV' ]; then
  echo "Running Development Server"
  exec python "identidock.py"
elif [ "$ENV" = 'UNIT']; then
  echo "Running Unit Tests"
  exec python "tests.py"
else
  echo "Running Production Server"
  exec uwsgi --http 0.0.0.0:9090 --wsgi-file /app/identidock.py \
             --callable app --stats 0.0.0.0:9191
fi
```

После пересборки образа можно запускать тесты, всего лишь изменив значение переменной среды:

```
$ docker build -t identidock .
...
$ docker run -e ENV=UNIT identidock
Running Unit Tests
..
----------------------------------------------------------------------
Ran 2 tests in 0.010s

OK
```

Можно было бы написать гораздо больше модульных тестов. В частности, у нас нет тестов для метода `get_identicon()`. Но для модульного тестирования этого метода потребовалось бы использование тестовых версий сервисов dnmonster и Redis или *тест-дублеров* (*test double*). Тест-дублер заменяет реальный сервис и в большинстве случаев представляет собой либо *заглушку* (*stub*), которая просто возвращает фиксированный ответ (например, заглушка для сервиса, информирующего о курсах акций, может всегда возвращать фиксированное значение «42»), либо *объект-имитацию* (*mock*), для которого можно запрограммировать ожидаемое поведение при вызове (например, для конкретной транзакции обращение к объекту-имитации будет происходить только один раз). Для получения более подробной информации о тест-дублерах см. документацию по модулю Python mock (https://docs.python.org/3/library/unittest.mock.html), а также сайты специализированных HTTP-инструментов, таких как Pact (https://github.com/real-estate-com-au/pact), Mountebank (http://www.mbtest.org/) и Mirage (https://mirage.readthedocs.org/).

Включение тестов в образы

В этой главе мы включили тесты для приложения identidock в соответствующий образ, что полностью соответствует философии Docker в отношении использования одного и того же образа в процессах разработки, тестирования и эксплуатации. Это также означает, что мы можем без затруднений выполнить тесты для образов, работающих в различных средах, – полезная возможность при отладке. Недостатком является увеличение размера образа – приходится включать исходный код тестов и все зависимости, такие как библиотеки поддержки тестирования. В свою очередь, это приводит к увеличению потенциальной *поверхности атаки* (attack surface), то есть взломщик получает возможность воспользоваться тестовыми утилитами или кодом тестов для проникновения в эксплуатируемую систему.

В большинстве случаев преимущества, определяемые простотой и надежностью использования единого образа, оказываются более значимыми, чем недостатки, связанные с небольшим увеличением размера и потенциальной угрозой для безопасности.

Следующий этап – организация автоматического тестирования на сервере непрерывной интеграции, и мы рассмотрим возможности автоматического тестирования, когда проверка выполняется в процессе разработки на стадии управления исходным кодом до перемещения его в стабильную версию или в версию для эксплуатации.

Использование контейнеров для быстрого тестирования

Все тесты, в особенности модульные тесты, должны выполняться быстро, чтобы поощрять разработчиков запускать их как можно чаще, при этом не теряя времени на ожидание результатов. Контейнеры предоставляют быстрый способ загрузки хорошо подготовленной и изолированной среды, которая может оказаться удобной для выполнения тестов, изменяющих программную среду. Например, допустим, что имеется набор тестов, использующий определенный сервис[1], на котором предварительно подготовлены некоторые тестовые данные. Каждый тест, обращающийся к этому сервису, с большой вероятностью каким-либо образом изменяет программную среду, добавляя, удаляя или модифицируя данные. Один из способов создания подобных тестов состоит в восстановлении исходного состояния данных после каждого прогона тестов, но это не всегда возможно: если отдельный тест (или процедура восстановления) завершился с ошибкой, то тестовые данные искажаются для всех последующих тестов, затрудняя поиск источника ошибки, кроме того, при этом требуется знание механизма работы тестируемого

[1] Подобные тесты более характерны для интеграционного тестирования или для тестирования системы в целом, чем для модульного тестирования, тем не менее их можно применять для модульного тестирования в тестовой конфигурации без объектов-имитаторов. Многие эксперты модульного тестирования рекомендуют заменять некоторые компоненты, например базы данных, объектами-имитаторами, но в тех ситуациях, когда компонент стабилен и надежен, часто проще и разумнее использовать сам этот компонент.

сервиса (он перестает быть «черным ящиком»). Другой способ – удалять сервис после каждого теста и запускать новый для каждого следующего теста. При использовании в процессе тестирования виртуальных машин такая процедура, вероятнее всего, окажется слишком медленной, но при использовании контейнеров это решение вполне приемлемо.

Контейнеры предоставляют преимущества и в другой дисциплине тестирования – при запуске сервисов в различных средах/конфигурациях. Если проверяемое ПО должно работать в различных дистрибутивах Linux с различными установленными СУБД, то нужно создать отдельный образ для каждой конфигурации, и тесты будут выполняться очень быстро. Скрытым недостатком такого подхода является то, что он не учитывает различия в ядрах, устанавливаемых в разнообразных дистрибутивах.

Создание контейнера для сервера Jenkins

Jenkins – это широко известный сервер непрерывной интеграции с открытым исходным кодом. Существуют и другие серверы непрерывной интеграции, и альтернативные решения, но здесь мы используем именно Jenkins для своего веб-приложения просто из-за его распространенности. Нужно настроить сервер Jenkins таким образом, чтобы при любых изменениях в нашем проекте identidock он автоматически проверял внесенные изменения, создавал новые образы и выполнял для них определенные операции тестирования, как описанные выше модульные тесты, так и некоторые тесты системы в целом. Кроме того, должен создаваться отчет о результатах этих тестов.

Решение этой задачи будет основано на образе из официального репозитория Jenkins. В примерах данной книги используется версия 1.609.3, хотя следует отметить, что новые версии сервера выпускаются достаточно часто, поэтому вы можете попробовать любую более новую версию, но, возможно, потребуются некоторые изменения для сохранения работоспособности примеров.

Чтобы контейнер Jenkins мог формировать образы, необходимо смонтировать размещенный на хосте Docker-сокет[1] внутри контейнера, тем самым позволяя серверу Jenkins успешно создавать образы-«братья» того же уровня. Другое решение состоит в использовании методики Docker-в-Docker (*Docker-in-Docker – DinD*), при которой Docker-контейнер может создавать собственные контейнеры-потомки. Эти две методики сравниваются на рис. 8.1.

---

[1] *Docker-сокет* – это точка входа (endpoint), используемая для обмена данными между клиентом и демоном. По умолчанию это *IPC-сокет* (*IPC – interprocess communication – взаимодействие процессов*), доступный через файл */var/run/docker.sock*, но Docker также поддерживает TCP-сокеты, определяемые по сетевому адресу, и сокеты механизма systemd. В этой главе предполагается использование сокета по умолчанию */var/run/docker.sock*. Так как доступ к такому сокету организован через дескриптор файла, можно просто смонтировать эту точку входа как том внутри контейнера.

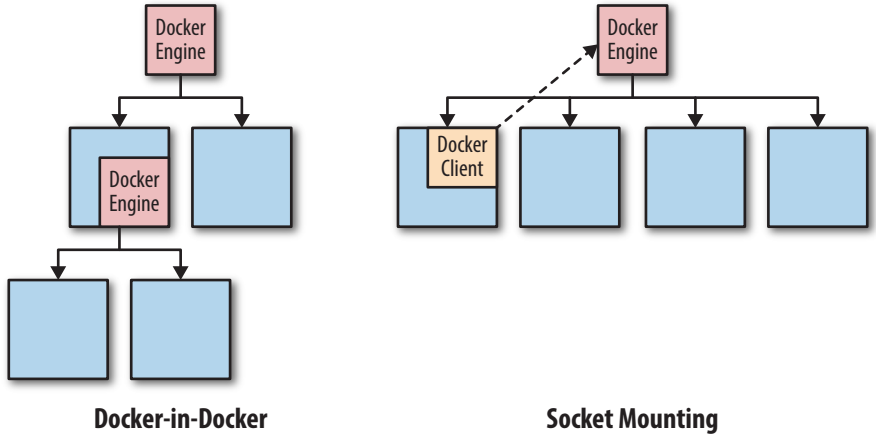

Рис. 8.1. Сравнение методики Docker-в-Docker и методики монтирования сокета

Docker-in-Docker

Методика Docker-in-Docker (Docker-in-Docker – DinD) представляет собой обычный запуск механизма Docker внутри контейнера. Для обеспечения работоспособности такого подхода необходима особая конфигурация, в основном касающаяся запуска контейнера в привилегированном режиме и решения некоторых проблем с файловой системой. Но гораздо проще воспользоваться проектом DinD Джерома Петаццони (Jérême Petazzoni), размещенным на GitHub (https://github.com/jpetazzo/dind), и объяснить все требуемые операции. Работу можно начать немедленно, если взять образ DinD, который Джером поместил в реестр Docker Hub:

```
$ docker run --rm --privileged -t -i -e LOG=file jpetazzo/dind
ln: failed to create symbolic link '/sys/fs/cgroup/systemd/name=systemd': Operation not
permitted
(ln: невозможно создать символическую ссылку '/sys/fs/cgroup/systemd/name=systemd': Операция запрещена)
root@02306db64f6a:/# docker run busybox echo "Hello New World!"
Unable to find image 'busybox:latest' locally
(Образ 'busybox:latest' в локальной системе не найден)
Pulling repository busybox
(Загрузка образа busybox из репозитория)
d7057cb02084: Download complete (Загрузка завершена)
cfa753dfea5e: Download complete
Status: Download newer image for busybox:latest
(Состояние: загружен более новый образ busybox:latest)
Hello New World!
```

Главное различие между методикой DinD и методикой монтирования сокета состоит в том, что контейнеры, созданные по методике DinD, полностью изолированы от контейнеров хоста. При выполнении команды `docker ps` внутри DinD-контейнера будут показаны только контейнеры, созданные демоном DinD Docker. При исполь-

зовании методики монтирования сокета команда `docker ps` покажет все контейнеры, вне зависимости от места запуска команды.

Вообще говоря, я предпочитаю более простую методику монтирования сокета, но при определенных условиях может потребоваться дополнительная изоляция, обеспечиваемая методикой DinD. При выборе методики DinD необходимо помнить о следующих вещах:

- используется отдельный кэш, поэтому первоначальные процедуры создания образов выполняются медленнее, и приходится повторно загружать все образы. Локальный реестр или зеркало удаленного реестра могут немного улучшить положение. Не пытайтесь смонтировать кэш сборки образов с хоста – механизм Docker разрешает доступ только одному объекту, поэтому при совместном использовании двумя экземплярами могут происходить непредсказуемые и крайне нежелательные события;
- контейнер должен запускаться в привилегированном режиме, что делает его менее защищенным, по сравнению с методикой монтирования сокета (если взломщик получает доступ, то у него появляется возможность смонтировать любое устройство, в том числе и физические накопители). Поэтому можно порекомендовать разработчикам Docker в будущем обеспечить поддержку более продуманного разделения привилегий и прав доступа, которая позволяла бы самим пользователям выбирать устройства, доступные DinD-контейнерам;
- методика DinD использует тома в каталоге /var/lib/docker, и если вы забываете удалять тома после завершения работы контейнеров, то это может привести к быстрому заполнению дискового пространства.

Чтобы больше узнать обо всех нюансах использования методики DinD, обратитесь к статье Джерома Петаццони в его проекте на GitHub http://bit.ly/1WtECmm.

Чтобы смонтировать сокет с хоста, необходимо убедиться в том, что пользователь сервера Jenkins внутри контейнера обладает требуемыми правами доступа. В новом каталоге *identijenk* создайте Dockerfile со следующим содержимым:

```
FROM jenkins:1.609.3

USER root
RUN echo "deb http://apt.dockerproject.org/repo debian-jessie main" \
          > /etc/apt/sources.list.d/docker.list \
      && apt-key adv --keyserver hkp://p80.pool.sks-keyserver.net:80 \
          --recv-keys 58118E89F3A912897C070ADBF76221572C52609D \
      && apt-get update \
      && apt-get install -y apt-transport-https \
      && apt-get install -y sudo \
      && apt-get install -y docker-engine \
      && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers
USER jenkins
```

Этот Dockerfile скачивает основной образ Jenkins, устанавливает бинарный файл Docker и добавляет права выполнения `sudo`-команд без указания пароля для пользователя `jenkins`. Отметим, что пользователь `jenkins` преднамеренно не добав-

лен в группу пользователей Docker, поэтому все последующие команды Docker будут выполняться с префиксом sudo.

Не используйте группу docker

Вместо многократного повторения префикса sudo можно было бы добавить пользователя jenkins в группу docker хоста. Но при этом возникает проблема: на хосте непрерывной интеграции потребуются поиск и использование идентификатора GID группы docker, а также внесение фиксированного значения GID в код Dockerfile. Такой Dockerfile лишается свойства переносимости, так как значения GID для группы docker будут различными на разных хостах. Для устранения этой проблемы предпочтительнее использовать механизм sudo.

Создание образа:

```
$ docker build -t identijenk .

...
Successfully built d0c716682562
```

Проверка его работоспособности:

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock identijenk sudo docker ps
CONTAINER ID  IMAGE       COMMAND                CREATED        STATUS  ...
a36b75062e06  identijenk  "/bin/tini -- /usr/lo"  1 seconds ago  Up Less tha...
```

В приведенной выше команде docker run смонтирован Docker-сокет для установления соединения с демоном Docker на том же хосте. В более старых версиях Docker общепринятой практикой было монтирование еще и бинарного файла Docker вместо установки механизма Docker внутри контейнера. Преимущество такого подхода заключалось в поддержке синхронизации версий Docker на хосте и внутри контейнера. Но с версии 1.7.1 для Docker началось использование динамических библиотек, таким образом, все зависимости также необходимо было монтировать внутри контейнера. Поэтому вместо устранения многочисленных проблем с поиском и обновлением библиотек, требуемых для монтирования, решили просто устанавливать механизм Docker внутри образа.

Теперь у нас есть механизм Docker, работающий внутри контейнера, и можно устанавливать другой комплект ПО, необходимый для обеспечения создания образов на сервере Jenkins. Dockerfile нужно изменить следующим образом:

```
FROM jenkins:1.609.3

USER root
RUN echo "deb http://apt.dockerproject.org/repo debian-jessie main" \
          > /etc/apt/sources.list.d/docker.list \
      && apt-key adv --keyserver hkp://p80.pool.sks-keyserver.net:80 \
          --recv-keys 58118E89F3A912897C070ADBF76221572C52609D \
      && apt-get update \
      && apt-get install -y apt-transport-https \
      && apt-get install -y sudo \
      && apt-get install -y docker-engine \
```

```
      && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers

RUN curl -L https://github.com/docker/compose/releases/download/1.4.1/\
      docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose; \
      chmod +x /usr/local/bin/docker-compose  ❶

USER jenkins
COPY plugins.txt /usr/share/jenkins/plugins.txt  ❷
RUN /usr/local/bin/plugins.sh /usr/share/jenkins/plugins.txt
```

❶ Установка компонента Docker Compose, с помощью которого будут автоматически создаваться и запускаться образы.

❷ Копирование и обработка файла *plugins.txt*, в котором определен список подключаемых модулей для установки на сервере Jenkins.

В том же каталоге, где расположен Dockerfile, создайте файл *plugins.txt* со следующим содержимым:

```
scm-api:0.2
git-content:1.16.1
git:2.3.5
greenballs:1.14
```

Первые три подключаемых модуля предназначены для настройки интерфейса, используемого при организации доступа к проекту Identidock в репозитории Git Hub. Модуль `greenballs` заменяет синие значки, по умолчанию обозначающие успешное завершение процесса создания образа, на зеленые.

Теперь мы почти готовы к запуску контейнера Jenkins и конфигурированию самого процесса создания образов, но сначала необходимо создать контейнер данных для постоянного хранения параметров конфигурации:

```
$ docker build -t identijenk .
...
$ docker run --name jenkins-data identijenk echo "Jenkins Data Container"
Jenkins Data Container
```

Для контейнера данных использован образ Jenkins, поэтому мы можем быть уверены в правильности установки прав доступа. Контейнер завершает работу сразу после выполнения команды `echo`, но пока он не удален, его можно использовать, указывая как аргумент для `--volumes-from`. Более подробно контейнеры данных описаны в разделе «Управление данными с помощью томов и контейнеров данных» главы 4.

После этого контейнер Jenkins готов к запуску:

```
$ docker run -d --name jenkins -p 8080:8080 --volumes-from jenkins-data \
    -v /var/run/docker.sock:/var/run/docker.sock identijenk
75c4b300ade6a62394a328153b918c1dd58c5f6b9ac0288d46e02d5c593929dc
```

Если в браузере перейти по адресу `http://localhost:8080`, то можно наблюдать инициализацию сервера Jenkins. Немного позже мы настроим процедуры создания и тестирования образа для нашего проекта identidock. Но сначала необходимо

внести небольшое изменение в сам проект. В текущий момент файл *docker-compose.yml* для этого проекта инициализирует версию для разработки, но мы должны разработать набор тестов системы в целом, выполняемый для версии, максимально адаптированной для эксплуатации. Поэтому нужно создать новый файл *jenkins.yml*, который будет использоваться для запуска эксплуатационной версии identidock внутри контейнера Jenkins:

```
identidock:
  build: .
  expose:
   - "9090"  ❶
  environment:
    ENV: PROD  ❷
  links:
   - dnmonster
   - redis

dnmonster:
  image: amouat/dnmonster:1.0

redis:
  image: redis:3.0
```

❶ Поскольку Jenkins располагается в «соседнем» контейнере, нет необходимости объявлять открытыми порты на хосте, чтобы установить соединение с ними. Здесь команда `expose` включена главным образом в целях документирования, и даже без нее контейнер identidock будет доступным из контейнера Jenkins, но при условии, что вы не изменили параметров сетевых настроек, принятых по умолчанию.

❷ Установка среды как предназначенной для эксплуатации.

Этот файл нужно добавить в репозиторий identidock, из которого будет извлекать исходный код сервер Jenkins, то есть в собственный частный репозиторий, если вы ранее сконфигурировали его, или в существующий репозиторий на Git Hub (https://github.com/using-docker/identidock).

Теперь сервер Jenkins полностью готов к началу процедуры конфигурирования и сборки образов. Откройте веб-интерфейс Jenkins по адресу http://localhost:8080 и выполните следующие инструкции:

1. Щелкните по ссылке **create new jobs** (создать новые задания).
2. В поле **Item name** (Имя элемента) введите `identidock`, выберите вариант **Freestyle project** (Проект в свободном стиле) и щелкните по кнопке **ОК**.
3. Настройте параметры **Source Code Management** (Управление исходным кодом). При использовании общедоступного репозитория GitHub нужно выбрать вариант «Git» и ввести URL своего репозитория. Если используется собственный частный репозиторий, то потребуется настройка специальных удостоверений (некоторые репозитории, в том числе BitBucket, предлагают *ключи развертывания* (*deployment keys*), которые можно использовать для настройки доступа только для чтения как средства защиты от несанкциони-

рованного доступа). Кроме того, можно воспользоваться готовой версией, доступной на GitHub (https://github.com/using-docker/identidock).

4. Щелкните по кнопке **Add build step** (Добавить этап создания) и выберите вариант **Execute shell** (Выполнение скрипта командной оболочки). В поле ввода текстового блока **Command** (Команда) введите следующий текст скрипта:

```
# Аргументы для Compose, определяемые по умолчанию
COMPOSE_ARGS=" -f jenkins.yml -p jenkins "

# Необходимо остановить и удалить все старые контейнеры
sudo docker-compose $COMPOSE_ARGS stop  ❶
sudo docker-compose $COMPOSE_ARGS rm --force -v

# Создание (сборка) системы
sudo docker-compose $COMPOSE_ARGS build --no-cache
sudo docker-compose $COMPOSE_ARGS up -d

# Выполнение модульного тестирования
sudo docker-compose $COMPOSE_ARGS run --no-deps --rm -e ENV=UNIT identidock ERR=$?

# Выполнение тестирования системы в целом, если модульное тестирование завершилось
успешно
if [ $ERR -eq 0 ]; then
  IP=$(sudo docker inspect -f {{.NetworkSettings.IPAddress}}
          jenkins_identidock_1)  ❷
  CODE=$(curl -sL -w "%{http_code}" $IP:9090/monster/bla -o /dev/null) || true  ❸
  if [ $CODE -ne 200 ]; then
    echo "Site returned " $CODE
    ERR=1
  fi
fi

# Останов и удаление системы
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

return $ERR
```

❶ Еще раз отметим использование `sudo` при вызове Docker Compose, так как пользователь `jenkins` не является членом группы `docker`.

❷ Команда `docker inspect` применяется для определения IP-адреса контейнера identidock.

❸ Утилита `curl` используется для доступа к сервису identidock, при этом проверяется код возврата HTTP: если возвращен код, не равный 200, то сервис работает правильно. Также отметим, что здесь используется путь `/monster/bla` для подтверждения возможности соединения identidock с сервисом dnmonster.

Приведенный выше код скрипта также можно взять из репозитория GitHub (https://github.com/using-docker/ci-testing). Обычно подобные скрипты должны проверяться вместе с другим кодом в системе управления версиями, но для нашего простого примера достаточно просто скопировать его на сервер Jenkins.

Теперь необходимо протестировать все в комплексе, для этого щелкните по кнопке **Save** (Сохранить), затем по кнопке **Build Now** (Создать). Подробности

процесса создания можно наблюдать, щелкнув по кнопке идентификатора сборки и выбрав пункт **Console Output** (Вывод на консоль). На рис. 8.2 показан один из возможных вариантов вывода результатов создания образа.

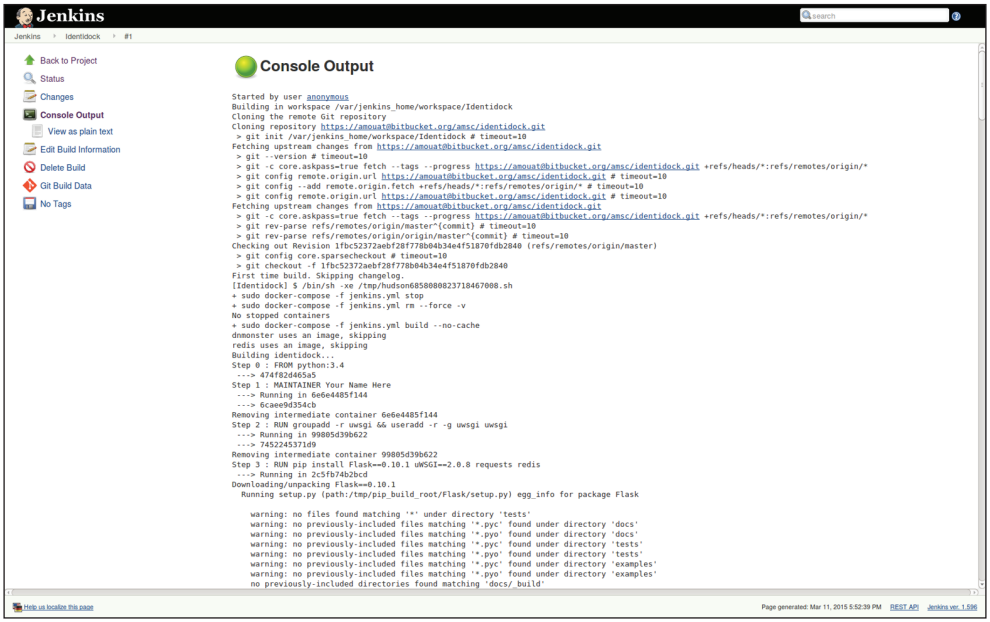

Рис. 8.2. Успешное создание образа с помощью сервера Jenkins

Итак, мы полностью подготовили и настроили запуск нашего Docker-приложения, управление автоматическим выполнением модульных тестов, а также создали минимальный набор (smoke test) для тестирования системы в целом. Если бы это было реальное приложение, то следовало бы заняться формированием полного комплекта тестов, подтверждающих правильное функционирование приложения и способных обрабатывать всевозможные наборы входных данных, но для нашего простого учебного примера сделанного выше вполне достаточно.

Создание образа по триггеру

В предыдущем разделе процедура создания образа запускалась вручную при щелчке по кнопке **Build Now** (Создать). Можно усовершенствовать эту операцию, определив ее автоматический запуск по результатам проверки проекта в репозитории GitHub. Для этого в конфигурации identidock нужно разрешить использование метода **Poll SCM** (Опрос системы управления версиями) и в текстовом поле ввести строку `"H/5 * * * *"`. После этого сервер Jenkins каждые пять минут будет проверять репозиторий исходного кода и при обнаружении изменений запускать процедуру создания образа.

Такое простое решение работает вполне удовлетворительно, тем не менее оно нерационально использует ресурсы, к тому же сборка образа постоянно запаздывает на несколько минут. Более эффективным решением является определение такой конфигурации репозитория исходного кода, при которой сервер Jenkins уведомляется о внесении изменений. Это можно сделать с помощью методики Webhook, доступной и на BitBucket, и на GitHub, но для реализации потребуется доступ к серверу Jenkins через Интернет.

Использование образа из Docker Hub

Здесь у читателя может возникнуть вопрос: «А зачем мы вообще создаем образ?» Если вы следовали инструкциям предыдущего раздела, то у вас должна быть настроена процедура автоматического создания образов в реестре Docker Hub, выполняющая регулярные проверки репозитория исходного кода. Можно получить дополнительные преимущества, используя функциональные возможности Webhooks в реестре Docker Hub для автоматической инициализации процедуры создания образа сервером Jenkins после успешного завершения процесса создания образа в репозитории Docker Hub. Затем с помощью специального скрипта можно загрузить этот образ, а не создавать его заново. Но и в этом случае потребуется доступ к серверу Jenkins через Интернет.

Такое решение может оказаться удобным для небольших проектов, создающих одиночные Docker-образы, но для крупномасштабных проектов с большой вероятностью потребуются более высокая скорость и более строгие меры обеспечения безопасности при управлении процедурой создания образов.

Выгрузка образа в реестр

После успешного тестирования образа identidock необходимо провести его по оставшейся части непрерывного рабочего процесса. Первый этап – присваивание тегов и выгрузка в реестр. Далее образ будет передан на следующий этап – конечные процедуры подготовки и эксплуатация.

Присваивание осмысленных тегов

Правильное присваивание тегов очень важно для реализации управления и установления происхождения образов в непрерывном рабочем процессе, основанном на использовании контейнеров. Ошибки могут сильно затруднить ввод образов в эксплуатацию и даже сделать его невозможным – это проявляется при возврате к процедурам создания, при отладке, а сопровождение становится неоправданно сложным. Для любого конкретного образа должна сохраняться возможность точного определения соответствующего ему Dockerfile и контекста, который был использован при создании этого образа[1].

[1] Отметим, что это вовсе не значит, что вы сможете воспроизвести точно такой же контейнер, так как зависимости для него могли измениться. Более подробное описание решения этой проблемы см. в разделе «Повторно воспроизводимые и надежные файлы Dockerfile» главы 13.

Теги можно перезаписать и изменить в любое время. Именно поэтому вся ответственность за правильную организацию процесса присваивания тегов и нумерации версий образов ложится на автора-разработчика.

Для нашего учебного приложения мы присвоим образу два тега: хэш-значение из git-репозитория и `newest`. Тег `newest` всегда будет обозначать самую последнюю сборку, которая успешно прошла все тесты, а хэш-значение git-репозитория можно использовать для восстановления комплекта файлов любой версии образа. Я преднамеренно отказываюсь от тега `latest` по причинам, изложенным в примечании «Опасность использования тега latest» главы 7. Отредактируйте скрипт сборки для сервера Jenkins следующим образом:

```
# Аргументы для Compose, определяемые по умолчанию
COMPOSE_ARGS=" -f jenkins.yml -p jenkins "

# Необходимо остановить и удалить все старые контейнеры
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

# Создание (сборка) системы
sudo docker-compose $COMPOSE_ARGS build --no-cache
sudo docker-compose $COMPOSE_ARGS up -d
# Выполнение модульного тестирования
sudo docker-compose $COMPOSE_ARGS run --no-deps --rm -e ENV=UNIT identidock ERR=$?
# Выполнение тестирования системы в целом, если модульное тестирование завершилось успешно
if [ $ERR -eq 0 ]; then
  IP=$(sudo docker inspect -f {{.NetworkSettings.IPAddress}} jenkins_identidock_1)
  CODE=$(curl -sL -w "%{http_code}" $IP:9090/monster/bla -o /dev/null) || true
  if [ $CODE -ne 200 ]; then
    echo "Test passed - Tagging"
    HASH=$(git rev-parse --short HEAD)  ❶
    sudo docker tag -f jenkins_identidock amouat/identidock:$HASH  ❷
    sudo docker tag -f jenkins_identidock amouat/identidock:newest  ❷
    echo "Pushing"
    sudo docker login -e joe@bloggs.com -u jbloggs -p jbloggs123  ❸
    sudo docker push amouat/identidock:$HASH  ❹
    sudo docker push amouat/identidock:newest  ❹
  else
    echo "Site returned " $CODE
    ERR=1
  fi
fi

# Останов и удаление системы
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

return $ERR
```

❶ Получение короткой версии хэш-значения от git.
❷ Добавление тегов.
❸ Авторизованный вход в реестр.
❹ Выгрузка образов в реестр.

Обратите внимание на необходимость переименования тега для соответствия с названием репозитория, в который вы намерены выгрузить образы. Например, если ваш репозиторий работает на хосте `myhost:5000`, то нужно использовать имя `myhost:5000/identidock:newest`. Аналогичным образом в команде `docker login` следует изменить адрес e-mail, имя и пароль на действительно существующие.

При создании нового образа вы наверняка заметите, что теперь скрипт присваивает теги и выгружает образы в заданный реестр, обеспечивая полную готовность к следующему этапу непрерывного рабочего процесса. Удачное решение для нашего учебного приложения, которое с большой вероятностью подойдет и для большинства проектов. Но поскольку реальные проекты имеют тенденцию к постоянному усложнению, скорее всего, вы будете использовать больше тегов и определять более информативные имена. Команда `git describe` может оказать существенную помощь при генерации более осмысленных имен, основанных на тегах.

Поиск всех тегов для образа

Для любого образа каждый тег хранится отдельно. Это означает, что для обнаружения всех тегов для некоторого образа необходимо составить полный список образов, определяемый идентификатором этого образа в качестве фильтра. Например, для поиска всех тегов для образа с тегом *amouat/identidock:newest* выполняется следующая команда:

```
$ docker images --no-trunc | grep $(docker inspect -f {{.Id}}
    amouat/identidock:newest)
amouat/identidock    51f6152    96c7b4c094c8f76ca82b6206f...
amouat/identidock    newest     96c7b4c094c8f76ca82b6206f...
jenkins_identidock   latest     96c7b4c094c8f76ca82b6206f...
```

Здесь мы видим, что этот образ также имеет тег `51f6152`.

Следует помнить, что тег можно увидеть, только если он существует в кэше образов. Например, если вы загружаете `debian:latest`, то не получите тега `debian:7`, даже если их идентификаторы одинаковы (положение на момент публикации книги). Если имеются образы `debian:latest` и `debian:7` и вы загружаете новую версию `debian:latest`, это никак не повлияет на образ с тегом `debian:7`, который останется связанным с предыдущим идентификатором образа.

Конечные процедуры подготовки и эксплуатация

После того как образ протестирован, снабжен тегами и выгружен в реестр, необходимо передать его на следующий этап непрерывного рабочего процесса – *конечные процедуры подготовки* (*staging*) или *эксплуатацию* (*production*). Это можно сделать несколькими способами, в том числе с применением уведомлений по методике webhook (https://docs.docker.com/registry/notifications/) реестра или используя для этой цели сервер Jenkins.

Беспорядочный рост количества образов

При эксплуатации системы необходимо решить проблему *беспорядочного роста количества образов* (*image sprawl*). Сервер Jenkins должен периодически избавляться от ненужных образов, также необходимо управление количеством образов

в реестре, чтобы избежать его быстрого заполнения устаревшими и промежуточными версиями образов. Первый способ решения этой проблемы заключается в удалении всех образов, которые старше определенной даты, возможно, с помещением их в хранилище резервных копий, если позволяет дисковое пространство[1]. Другой способ предполагает использование более развитых инструментальных средств, например CoreOS Enterprise Registry или Docker Trusted Registry, в которые включены дополнительные функциональные возможности для управления репозиториями.

Проверка соответствия

Важно точно знать, что вы тестируете именно тот образ контейнера, который будет использоваться для реальной эксплуатации. Не рекомендуется создавать для тестирования образ из Dockerfile, затем повторно создавать образ для эксплуатации – вы должны быть абсолютно уверены в том, что в эксплуатацию вводится тот же образ, который был протестирован, без каких-либо скрытых отличий. Поэтому очень важно использовать такую версию реестра или хранилища образов, которая обеспечивает одновременную поддержку этапов тестирования, конечной подготовки и эксплуатации.

Использование Docker для поддержки вспомогательных серверов Jenkins

При активной работе с репозиторием образов потребуется постоянное наращивание ресурсов для выполнения тестов. Сервер Jenkins использует концепцию «вспомогательных сборочных серверов», которая позволяет сформировать целый комплекс серверов Jenkins для выполнения задач по созданию образов из исходных кодов, располагающихся на удаленных хостах.

Если необходимо использовать Docker для динамической поддержки таких вспомогательных серверов, то следует обратить внимание на дополнительный подключаемый Docker-модуль для сервера Jenkins (https://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin).

Организация резервного копирования для сервера Jenkins

Поскольку для сервиса Jenkins мы используем контейнер данных, резервное копирование для сервера Jenkins выполняется достаточно просто:

```
$ docker run --volumes-from jenkins-data -v $(pwd):/backup \
         debian tar -zcvf /backup/jenkins-data.tar.gz /var/jenkins_home
```

---

[1] На момент публикации данной книги для локально размещенных реестров Docker это проще предложить, чем сделать, поскольку функция удаления из реестра еще не реализована. Кроме того, существует еще несколько проблем, которые подробно описаны в технологической дорожной карте для процедуры распространения образов (https://github.com/docker/distribution/blob/master/ROADMAP.md).

В результате выполнения этой команды создается файл *jenkins-data.tar.gz* в каталоге *$(pwd)/backup*. Можно остановить или временно приостановить контейнер Jenkins перед выполнением этой команды. После этого вы можете использовать команду, подобную приведенной ниже, для создания нового контейнера данных с последующей распаковкой в него резервной копии:

```
$ docker run --name jenkins-data2 identijenk echo "New Jenkins Data Container"
$ docker run --volumes-from jenkins-data2 -v $(pwd):/backup \
         debian tar -xzvf /backup/backup.tar
```

К сожалению, при таком подходе необходимо точно знать все точки монтирования вашего контейнера. Процедуру можно автоматизировать средствами исследования и наблюдения за контейнером, а кроме того, можно также воспользоваться такими инструментами, как docker-backup (https://github.com/discordianfish/docker-backup), для выполнения этой задачи. В будущих версиях Docker ожидается более солидная поддержка операций резервного копирования и других задач этого этапа рабочего процесса.

Хостинговые решения для непрерывной интеграции

Для поддержки процесса непрерывной интеграции существует множество хостинговых решений: от самых простых, когда компании выполняют обязанности по сопровождению сервера Jenkins, установленного в облачной среде, до комплексных специализированных решений, таких как Travis (https://travis-ci.org/), Wercker (http://wercker.com/), CircleCI (https://circleci.com/) и drone.io (https://drone.io/). Большинство этих решений в большей степени ориентировано на выполнение модульных тестов для предопределенного набора языков программирования, нежели на организацию тестирования для системы контейнеров. Тем не менее наблюдается рост интереса указанных компаний к сфере контейнеризации, и я надеюсь на скорое появление специализированных решений для тестирования Docker-контейнеров.

Тестирование и микросервисы

При использовании Docker предоставляется отличный шанс для перехода на архитектуру микросервисов. Тестируя архитектуру микросервисов, вы обнаружите, что количество уровней тестирования больше того, которое казалось возможным. Поэтому вы сами должны решать, что и как тестировать. В качестве основы для рабочей среды тестирования можно предложить следующие компоненты:

- *модульные тесты* (*юнит-тесты*) – для каждого сервиса[1] необходимо иметь полный набор модульных тестов, которые должны проверять правильную

[1] Обычно создается один контейнер для каждого сервиса или несколько контейнеров для каждого сервиса, если необходимо предоставить больше ресурсов.

функциональность небольших, изолированных частей приложения. Для устранения зависимостей от других сервисов можно воспользоваться тест-дублерами. Поскольку количество тестов достаточно велико, важно обеспечение их выполнения с максимально возможной скоростью, чтобы тестирование проводилось как можно чаще и разработчикам не приходилось бы подолгу ждать результатов. Модульные тесты должны составлять самую большую часть тестового комплекта для системы;

- *тесты компонентов* – это может быть тестирование на уровне внешнего интерфейса отдельных сервисов или тестирование на уровне подсистемы, состоящей из группы сервисов. В обоих случаях, вероятнее всего, обнаружатся зависимости от других сервисов, которые можно заменить тест-дублерами, как уже было сказано ранее. При тестировании также могут оказаться полезными определение метрик и фиксация событий в журнале через прикладной программный интерфейс проверяемого сервиса (сервисов), но при этом следует обеспечить хранение тестовой информации в отдельном пространстве имен (например, можно использовать другой URL-префикс), отделенном от основного рабочего интерфейса (API);
- *тесты для системы в целом* – эти тесты подтверждают работоспособность всей системы. Поскольку выполнение системных тестов требует существенных затрат ресурсов и времени, их количество должно быть минимальным – нет смысла в проведении многочасового тестирования, серьезно задерживающего процедуры развертывания и исправления ошибок (следует рассмотреть вариант тестирования по расписанию, кратко описанный ниже). Некоторые компоненты системы иногда просто невозможно протестировать, или их тестирование влечет за собой непомерно высокие затраты, и такие компоненты также необходимо заменять тест-дублерами (реальный запуск ядерных ракет для их тестирования – не самое лучшее решение). Тестовый комплект для нашего учебного примера identidock соответствует методике тестирования системы в целом: проверяется вся система полностью без использования тест-дублеров.

Кроме того, можно рассмотреть использование в тестовом комплекте следующих дополнительных компонентов:

- *тесты по договору с заказчиком* (*потребителем*) – эти тесты, также называемые контрактами, управляемыми заказчиком, формируются потребителем или заказчиком сервиса и предназначены главным образом для определения ожидаемых входных и выходных данных. Кроме того, тесты могут выявлять побочные эффекты (изменение состояния) и оценивать производительность. С каждым потребителем конкретного сервиса должен быть заключен отдельный договор. Главное преимущество таких тестов заключается в том, что они позволяют разработчикам сервиса точно определять ситуации, связанные с нарушением требований заказчиков: если тест, определяемый договором, не прошел, то становится очевидной необходимость изменения функциональности сервиса или необходимость обсуждения разработчиками и заказчиками возможности изменения договора;

- *интеграционные тесты* – это тесты для проверки правильности работы каналов обмена информацией между всеми компонентами системы. Этот тип тестирования становится особенно важным для архитектуры микросервисов, где объем работ по коммуникации и координации всех компонентов на порядок выше, по сравнению с монолитными архитектурами. Но, вероятнее всего, для большинства каналов обмена информацией будет вполне достаточно тестов компонентов и тестов для системы в целом;
- *тестирование по расписанию* – при создании образов важно поддерживать высокую скорость потока непрерывной интеграции, поэтому часто не хватает времени для проведения глубокого, всеобъемлющего тестирования, такого как обнаружение неработающих конфигураций или проверка версий для различных платформ. Выполнение подобных тестов можно запланировать на ночное (нерабочее) время, когда высвобождаются дополнительные вычислительные мощности.

Многие из перечисленных выше тестов можно классифицировать как *дореестровые* (*preregistry*) и *послереестровые* (*postregistry*) в зависимости от того, выполняются ли они до или после добавления образа в реестр. Например, модульное тестирование является дореестровым: если модульные тесты провалены, то образ не должен выгружаться в реестр. То же самое справедливо для некоторых тестов по договору с заказчиком и для некоторых тестов компонентов. С другой стороны, образ может быть выгружен в реестр до того, как появится возможность его тестирования как системы в целом. Если послереестровый тест не прошел, то возникает вопрос: что делать дальше? Для всех новых образов должен быть запрещен ввод в эксплуатацию (или они должны быть отозваны из эксплуатации, если уже были развернуты), но в действительности ошибка могла возникнуть из-за более старых версий образов или вследствие неправильного взаимодействия новых образов. Для выявления и устранения этого типа ошибок может потребоваться более глубокое исследование, необходимо тщательно продумать процесс их обработки.

Тестирование в процессе эксплуатации

Наконец, пришло время задуматься о тестировании в процессе эксплуатации. Это не так страшно, как может показаться на первый взгляд. Эта методика особенно полезна в тех случаях, когда мы работаем с большим количеством пользователей, с разнообразными средами и конфигурациями, которые очень трудно протестировать заранее.

Одну из самых распространенных методик иногда называют *blue/green-развертывание* (*blue/green deployment*). Например, нужно обновить существующий работающий сервис (назовем его blue-версией), заменив его новой версией – green-версией. Вместо прямой замены blue-версии на green-версию можно организовать их совместную работу в течение некоторого времени. После установки и запуска green-версии на нее переключается рабочий трафик. Затем мы постоянно наблюдаем за системой, выявляя все неожиданности в ее поведении, такие как рост количества ошибок или задержки. Если работа новой версии признана неудовлетво-

рительной, то мы просто переключаемся обратно на blue-версию, возвращая ее в эксплуатацию. Если новая версия удовлетворяет всем требованиям и работает правильно, то blue-версию можно отключить.

Другие методики основаны на похожем принципе – старая и новая версии должны работать совместно. По *методике A/B*, или *многовариантном тестировании* (*multivariate testing*), две (и более) версии сервиса работают вместе в течение тестового периода, а пользователи распределяются между версиями случайным образом. Собираются и анализируются статистические данные, и на основе результатов, полученных в конце периода тестирования, одна из версий выбирается для эксплуатации. При *постепенном развертывании* (*ramped deployment*) новая версия сервиса становится доступной только небольшому подмножеству пользователей. Если эти пользователи не сталкиваются с проблемами при работе, то новую версию постепенно делают доступной все большему количеству пользователей. По методике *скрытого развертывания* (*shadowing*) обе версии сервиса отвечают на все запросы, но реально используются только результаты, получаемые от старой, стабильной версии. При сравнении результатов, полученных от обеих версий, можно убедиться, что поведение новой версии идентично поведению старой версии (или предсказуемо отличается в лучшую сторону). Методика скрытого развертывания особенно удобна при тестировании новых версий, в которых нет принципиальных изменений функциональности, только улучшены некоторые характеристики, например увеличена производительность.

Резюме

Основная мысль этой главы: контейнеры наилучшим образом подходят для использования в рабочем процессе непрерывной интеграции/доставки. Всегда помните о главном: один и тот же образ можно использовать на разных этапах этого рабочего процесса, а не создавать многократно образы на отдельных этапах. Следует использовать возможности применения существующих инструментов непрерывной интеграции для контейнеров, если это не связано с большим количеством проблем. Впрочем, в ближайшем будущем в этой области наверняка появятся новые специализированные инструментальные средства.

При использовании крупномасштабной архитектуры микросервисов необходимо больше времени уделять организации процесса тестирования и более тщательно изучить методики, кратко описанные в данной главе.

Глава 9

Развертывание контейнеров

Настало время взглянуть на задачу с бизнес-точки зрения – пора подумать о том, как организовать эксплуатацию Docker-контейнеров в реальном мире. На момент публикации данной книги едва ли не каждый рассуждает о применении Docker, многие экспериментируют с Docker, но лишь немногие действительно занимаются промышленной эксплуатацией Docker. Критики иногда отмечают это как полное отсутствие успеха у Docker, но при этом забывают о нескольких важных вещах. Принимая во внимание «юный возраст» этой программной среды, весьма обнадеживающим фактом является использование ее в промышленной эксплуатации многими организациями (в том числе Spotify, Yelp и Baidu), а также постоянное расширение списка преимуществ, предоставляемых всем, кто использует Docker при разработке и тестировании.

На текущий момент уже сформировалось мнение о вполне возможном и обоснованном применении контейнеров в промышленной эксплуатации. Большие проекты и организации могут начинать с малого и постепенно укрупняться, но контейнеры уже сейчас представляют собой реально осуществимое и очевидное решение для многих проектов.

В соответствии с текущим положением большинство общеизвестных способов развертывания контейнеров основано на предварительной установке виртуальных машин и последующем запуске контейнеров в этих виртуальных машинах. Это не самое лучшее решение – возникают значительные накладные расходы, замедляется масштабирование, пользователям приходится работать одновременно со множеством отделенных друг от друга контейнеров. Главная причина запуска контейнеров внутри виртуальных машин – достаточно простое обеспечение безопасности. Здесь особенно важно то, что клиенты лишены возможности доступа к данным и сетевому трафику других клиентов, ведь сами по себе контейнеры предоставляют весьма слабую степень изоляции в настоящее время. Более того, если контейнер захватит все ресурсы ядра или станет причиной краха всей системы, это нарушит работоспособность всех контейнеров, запущенных на том же хосте. Даже большинство специализированных решений – Google Container

Engine (GKE) и Amazon EC2 Container Service (ECS) – продолжает использовать виртуальные машины внутри своих механизмов. На сегодняшний день существуют только два исключения из этого правила – Giant Swarm и Triton компании Joyent, которые будут более подробно рассматриваться ниже.

В этой главе будет показано, как можно развернуть наше простое веб-приложение в различных облачных средах, а также с помощью специализированных сервисов для Docker-хостинга. Мы также рассмотрим некоторые аспекты и методики ввода контейнеров в эксплуатацию как в облачной среде, так и с использованием локальных ресурсов.

Исходные коды для этой главы доступны на странице книги в репозитории GitHub. Мы больше не будем заниматься повторными сборками Python-кода из предыдущих глав, но продолжим пользоваться ранее созданными образами. Вы можете выбрать для дальнейшей работы одну из своих версий образа `identidock` или просто взять нужный образ из репозитория `amouat/identidock`.

Для получения начальной версии кода для данной главы выполните команду с использованием тега `v0`:

```
$ git clone -b v0 https://github.com/using-docker/deploying-containers
...
```

Следующие теги соответствуют версиям кода, получаемым в процессе дальнейшей разработки учебного примера в данной главе.

Кроме того, можно скачать код, соответствующий любому тегу, со страницы Releases проекта в репозитории GitHub (https://github.com/using-docker/deploying-containers/releases).

Предоставление ресурсов с помощью Docker Machine

Самым быстрым и простым способом предоставления новых ресурсов и запуска на них контейнеров является использование *механизма Docker Machine*. Этот механизм может создавать серверы, устанавливать на них Docker и формировать конфигурацию локального Docker-клиента для доступа к контейнерам. В комплект Docker Machine включены драйверы для работы с большинством основных облачных провайдеров, включая AWS, Google Compute Engine, Microsoft Azure и Digital Ocean, а также программные средства VMWare и VirtualBox.

Внимание, бета-версия

Во время написания данной книги была доступна только бета-версия Docker Machine (я протестировал Docker Machine версии 0.4.1). Поэтому при работе с ней возможны возникновение неожиданных ошибок и отсутствие некоторых функциональных возможностей, тем не менее эта версия остается работоспособной и достаточно стабильной. К сожалению, вполне вероятно, что в будущих версиях изменится синтаксис команд, по сравнению с приведенными здесь. По этой причине не рекомендуется использовать Docker Machine в реальной эксплуатации, хотя для тестирования и экспериментальных разработок механизм вполне пригоден.

(Следует добавить, что это предупреждение относится почти ко всему материалу, изложенному в данной книге. Время от времени я считаю необходимым напоминать об этом.)

Рассмотрим применение Docker Machine на практическом примере для процедуры загрузки приложения identidock в облачную среду с последующим его запуском. Для этого сначала необходимо установить Docker Machine на локальный компьютер. Если вы устанавливали Docker через Docker Toolbox, то механизм Docker Machine должен быть уже доступен. В противном случае можно скачать с GitHub (https://github.com/docker/machine/releases) пакет бинарных файлов, которые должны быть размещены в одном из каталогов, входящих в набор путей поиска (например, */usr/local/bin/docker-machine*). После этого должны стать доступными все команды Docker Machine:

```
$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM
default            virtualbox   Running   tcp://192.168.99.100:2376
```

Выводимая информация будет зависеть от того, на каких хостах обнаружен действующий механизм Docker Machine, возможно, не появится ни одной строки. В приведенном выше случае механизм найден в локальной виртуальной машине `boot2docker`. Далее необходимо добавить хост в облачную среду. В следующем примере будет использоваться Digital Ocean, но работа с AWS и другими облачными провайдерами практически ничем не отличается. Для продолжения действий нужно выполнить процедуру регистрации в режиме онлайн и сгенерировать токен личного доступа (перейдите на страницу https://cloud.digitalocean.com/settings/applications). Использование ресурсов платное, поэтому после завершения работы не забудьте остановить и удалить Docker Machine:

```
$ docker-machine create --driver digitalocean --digitalocean-access-token 4820... \
    identihost-do
Creating SSH key...
(Создание ключа SSH...)
Creating Digital Ocean droplet...
(Создание дроплета Digital Ocean...)
To see how to connect Docker to this machine, run: docker-machine env identi...
(Чтобы узнать, как установить соединение Docker с этой машиной, выполните команду: docker-machine env identi...)
```

Теперь Docker-хост в облаке Digital Ocean создан. Далее необходимо определить локального клиента для установления соединения, используя для этого команду, содержащуюся в выводимой информации:

```
$ docker-machine env identihost-do
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://104.236.32.178:2376"
export DOCKER_CERT_PATH="/Users/amouat/.docker/machine/machines/identihost-do"
export DOCKER_MACHINE_NAME="identihost-do"
# Run this command to configure your shell:
(# Выполните следующую команду, чтобы сконфигурировать свою командную оболочку:)
```

```
# eval "$(docker-machine env identihost-do)"
$ eval "$(docker-machine env identihost-do)"
$ docker info
Containers: 0
Images: 0
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 0
 Dirperm1 Supported: false
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.13.0-57-generic
Operating System: Ubuntu 14.04.3 LTS
CPUs: 1
Total Memory: 490 MiB
Name: identihost-do
ID: PLDY:REFM:PU5B:PRJK:L4QD:TRKG:RWL6:5T6W:AVA3:2FXF:ESRC:6DCT
Username: amouat
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
(ПРЕДУПРЕЖДЕНИЕ: Ограничение свопинга не поддерживается)
Labels:
 provider=digitalocean
```

Из выведенной информации можно понять, что установлено соединение с Ubuntu-хостом, работающим в облачной среде Digital Ocean. Если сейчас запустить команду `dcoker run hello-world`, то она будет выполнена на удаленном сервере.

Для запуска identidock можно воспользоваться файлом *docker-compose.yml*, приведенным в конце главы 6, или взять следующий файл *docker-compose.yml*, позволяющий работать с образом identidock из реестра Docker Hub:

```
identidock:
  image: amouat/identidock:1.0
  ports:
   - "5000:5000"
   - "9000:9000"
  environment:
    ENV: DEV
  links:
   - dnmonster
   - redis
  dnmonster:
    image: amouat/dnmonster:1.0
  redis:
    image: redis:3
```

Следует отметить, что если Compose-файл содержит инструкцию `build`, то создание образа будет происходить на удаленном сервере. При этом все монтируемые

тома необходимо удалить, так как ссылки будут определяться на диске удаленного сервера, а не на локальном компьютере.

Запустим Compose обычным образом:

```
$ docker-compose up -d  ❶
...
Creating identidock_identidock_1...
$ curl $(docker-machine ip identihost-do):5000  ❷
<html><head><title>Hello...
```

❶ Это займет некоторое время, поскольку сначала нужно загрузить и создать требуемые образы.

❷ Команду `docker-machine ip` можно использовать для определения места расположения нашего работающего Docker-хоста.

Теперь identidock работает в облачной среде и доступен всем[1]. Мы на удивление быстро установили и запустили приложение, но кое-что еще не сделано. В первую очередь обратим внимание на то, что данное приложение использует порт 5000 веб-сервера Python, предназначенного для разработки. Следует заменить его версией, предназначенной для промышленной эксплуатации, но, кроме того, неплохо было бы разместить перед приложением обратный прокси-сервер или балансировщик нагрузки, который позволил бы корректировать инфраструктуру identidock без изменения внешнего IP-адреса. Сервер nginx обеспечивает поддержку балансировки нагрузки, поэтому на нем можно без затруднений создать несколько экземпляров identidock, совместно использующих трафик.

Оперативное тестирование identidock

На протяжении всей книги выполняется обращение с помощью утилиты `curl` к сервису identidock, чтобы убедиться в его работоспособности. Но простое получение основной страницы – далеко не лучший тест, это просто подтверждение того, что контейнер identidock создан и запущен. Более подходящим тестом является получение идентификационного изображения, свидетельствующее, что контейнеры identidock и dnmonster активны и могут обмениваться данными. Подобный тест может быть, например, таким:

```
$ curl localhost:5000/monster/gordon | head -c 4
PNG
```

Здесь используется Unix-утилита `head` для извлечения первых четырех символов файла изображения, чтобы не загромождать терминал выводом бинарных данных.

Использование прокси-сервера

Начнем с создания обратного прокси-сервера, использующего nginx, за которым можно разместить сервис identidock. Для этого создадим новый каталог *identiproxy*, в котором разместим следующий файл *Dockerfile*:

---

[1] Некоторые провайдеры, в том числе AWS, могут потребовать предварительно открыть порт 5000 в сетевом экране.

```
FROM nginx:1.7

COPY default/conf /etc/nginx/conf.d/default.conf
```

Также создадим файл *default.conf* со следующим содержимым:

```
server {
    listen   80;
    server_name  45.55.251.164;  ❶

    location / {
        proxy_pass  http://identidock:9000;  ❷
        proxy_next_upstream error timeout invalid_header http_500 http_502 http_503 http_504;
        proxy_redirect off;
        proxy_buffering off;
        proxy_set_header        Host            45.55.251.164;  ❶
        proxy_set_header        X-Real-IP       $remote_addr;
        proxy_set_header        X-Forwarded-For  $proxy_add_x_forwarded_for;
    }
}
```

❶ Этот адрес необходимо заменить реальным IP-адресом вашего Docker-хоста или соответствующим доменным именем.

❷ Перенаправление всего трафика в контейнер identidock. Для этого будут использоваться связи между контейнерами.

Если Docker Machine все еще работает и указывает на сервер в облачной среде, можно попробовать создать новый образ на удаленном сервере:

```
$ docker build --no-cache -t identiproxy:0.1 .
Sending build context to Docker daemon 3.072 kB
(Передача контекста создания Docker-демону, 3.072 Кб)
Sending build context to Docker daemon
Step 0 : FROM nginx:1.7
 ---> 637d3b2f5fb5
Step 1 : COPY default.conf /etc/nginx/conf.d/default.conf
 ---> 2e82d9a1f506
Removing intermediate container 5383f47e3d1e
(Удаление вспомогательного контейнера 5383f47e3d1e)
Successfully built 2e82d9a1f506
(Успешно создан контейнер 2e82d9a1f506)
```

Теперь мы общаемся с удаленным Docker-механизмом, и новый образ действительно существует на удаленном сервере, а не на локальном компьютере, где происходила разработка.

После этого можно вернуться в каталог *identidock* и создать новый файл конфигурации Compose для тестирования удаленного сервиса. Содержимое файла *prod.yml*:

```
proxy:
  image: identiproxy:0.1  ❶
  links:
```

```
    - identidock
  ports:
   - "80:80"
identidock:
  image: amouat/identidock:1.0
  links:
   - dnmonster
   - redis
  environment:
    ENV: PROD  ❷
dnmonster:
  image: amouat/dnmonster:1.0  ❶
redis:
  image: redis:3  ❶
```

❶ Обратите внимание на использование тегов для всех образов. В режиме эксплуатации следует внимательнее следить за версиями запускаемых контейнеров. Здесь использование тега `latest` особенно нежелательно, так как он затрудняет или даже делает невозможным определение версии приложения, запускаемого в контейнере.

❷ Отметим отсутствие объявления портов для контейнера identidock (теперь это необходимо только для контейнера proxy). Кроме того, изменилось значение переменной среды – теперь она определяет эксплуатационную версию веб-сервера.

Использование ключевого слова extends в механизме Compose

Для более подробного определения содержимого YAML-файлов можно воспользоваться ключевым словом `extends`, позволяющим совместно использовать параметры конфигурации в различных средах. Например, можно создать файл common.yml со следующим содержимым:

```
identidock:
  image: amouat/identidock:1.0
  environment:
    ENV: DEV
dnmonster:
  image: amouat/dnmonster:1.0
redis:
  image: redis:3
```

Затем можно переписать файл prod.yml следующим образом:

```
proxy:
  image: identiproxy:0.1
  links:
    - identidock
  ports:
   - "80:80"
identidock:
  extends:
    file: common.yml
```

```
      service: identidock
    environment:
      ENV: PROD
  dnmonster:
    extends:
      file: common.yml
      service: dnmonster
  redis:
    extends:
      file: common.yml
      service: redis
```

Здесь ключевое слово `extends` позволяет взять соответствующие параметры конфигурации из общего файла. Значения параметров в файл prod.yml заменяют значения параметров из common.yml. Значения в секциях `links` и `volumes-from` не наследуются, чтобы избежать непредвиденных аварийных ситуаций. Поэтому в нашем случае использование `extends` на самом деле приводит к увеличению количества записей в файле prod.yml, хотя и сохраняется важное преимущество автоматического включения любых изменений, сделанных в общем файле. Главная причина отсутствия ключевого слова `extends` в данной книге проста: необходимость сохранения независимости всех примеров друг от друга.

Теперь нужно остановить старую версию и запустить новую:

```
$ docker-compose stop
Stopping identidock_identidock_1... done
Stopping identidock_redis_1... done
Stopping identidock_dnmonster_1... done
Starting identidock_dnmonster_1...
Starting identidock_redis_1...
Starting identidock_identidock_1...
Starting identidock_proxy_1...
```

Протестируем новую версию: теперь она должна отвечать через заданный по умолчанию порт 80, а не через порт 9090:

```
$ curl $(docker-machine ip identihost-do)
<html><head><title>Hello...
```

Все работает правильно. Теперь контейнер расположен за прокси-сервером, который предоставляет возможность балансировки нагрузки для группы экземпляров identidock или возможность перемещения identidock на новый хост без нарушения корректности IP-адреса (поскольку прокси-сервер остается на старом хосте и в его конфигурацию вносится новое значение). Кроме того, повышается степень защиты, так как доступ к контейнеру приложения возможен только через прокси-сервер, а сам контейнер вообще не открывает никаких портов, доступных напрямую из Интернета.

Ситуацию можно еще немного улучшить. Вызывает беспокойство тот факт, что IP-адрес хоста и имя контейнера жестко вписано в образ прокси-сервера, и если

потребуется использовать имя, отличающееся от identidock, или воспользоваться identiproxy для другого сервиса, то придется создавать новый образ или переписывать конфигурацию с использованием тома. Эти параметры лучше определить как переменные среды. Мы не можем использовать переменные среды напрямую в nginx, но можно написать скрипт, генерирующий конфигурационный файл во время выполнения, непосредственно перед запуском nginx. Вернемся в каталог *identiproxy* и отредактируем файл *default.conf*, разместив заменяемые шаблоны вместо жестко определенных значений переменных:

```
server {
    listen  80;
    server_name  {{NGINX_HOST}};

    location / {
        proxy_pass   {{NGINX_PROXY}};
        proxy_next_upstream error timeout invalid_header http_500 http_502 http_503 http_504;
        proxy_redirect off;
        proxy_buffering off;
        proxy_set_header        Host               {{NGINX_HOST}};
        proxy_set_header        X-Real-IP          $remote_addr;
        proxy_set_header        X-Forwarded-For    $proxy_add_x_forwarded_for;
    }
}
```

Затем нужно создать файл скрипта *entrypoint.sh*, который будет выполнять требуемые подстановки:

```
#!/bin/bash
set -e

sed -i "s|{{NGINX_HOST}}|$NGINX_HOST|;s|{{NGINX_PROXY}}|$NGINX_PROXY|" \
  /etc/nginx/conf.d/default.conf  ❶
cat /etc/nginx/conf.d/default.conf  ❷
exec "$@"  ❸
```

❶ Для выполнения подстановок применяется утилита sed. Возможно, это слегка шаблонно, но в данном случае хорошо подходит для наших целей. Отметим использование символов '|' вместо обычных слешей, чтобы не возникали проблемы со слешами в записях URL.

❷ Вывод полученного результата в системные журналы, это полезно для отладки.

❸ Выполнение команды, переданной через инструкцию CMD. По умолчанию контейнер Nginx определяет инструкцию CMD, запускающую nginx в фоновом режиме, но можно написать другую инструкцию CMD, которая в реальном времени выполняет другие команды или запускает командную оболочку, если это необходимо.

Теперь нужно отредактировать Dockerfile, включив в него новый скрипт:

```
FROM nginx:1.7

COPY default.conf /etc/nginx/conf.d/default.conf
COPY entrypoint.sh /entrypoint.sh
```

```
ENTRYPOINT ["/entrypoint.sh"]
CMD ["nginx", "-g", "daemon off;"]  ❶
```

❶ Эта команда запускает прокси-сервер. Она будет передана как аргумент в скрипт *entrypoint.sh*, если при выполнении `docker run` не задано других команд.

Сделайте новый скрипт выполняемым и создайте новый образ. Сейчас мы назовем его просто proxy, чтобы не разбираться с подробностями identidock:

```
$ chmod +x entrypoint.sh
$ docker build -t proxy:1.0 .
...
```

Чтобы начать работу с новым образом, вернитесь в каталог *identidock* и отредактируйте соответствующим образом файл *prod.yml*:

```
proxy:
  image: proxy:1.0
  links:
    - identidock
  ports:
   - "80:80"
  environment:
    - NGINX_HOST=45.55.251.164  ❶
    - NGINX_PROXY=http://identidock:9090
identidock:
  image: amouat/identidock:1.0
  links:
    - dnmonster
    - redis
  environment:
    ENV: PROD
dnmonster:
  image: amouat/dnmonster:1.0
redis:
  image: redis:3
```

❶ Этой переменной нужно присвоить реальный IP-адрес или имя вашего хоста.

Если теперь остановить работу старой версии и перезапустить приложение, то будет использоваться новый, только что созданный образ. Это все, что нужно для нашего простого веб-приложения, но при использовании соединений Docker в настоящий момент мы ограничены конфигурацией с одним хостом и не можем перейти к архитектуре на основе нескольких хостов (которая необходима для обеспечения устойчивости к критическим сбоям и для масштабирования) без применения более развитых сетевых средств и возможностей обнаружения сервисов, которые мы будем подробно рассматривать в главах 11 и 12.

После проверки работоспособности приложения можно остановить его командой:

```
$ docker-compose -f prod.yml stop
...
$ docker-compose -f prod.yml rm
...
```

Если вы намерены прекратить использование облачного ресурса и отключиться от него, то просто выполните следующие команды:

```
$ docker-machine stop identihost-do
$ docker-machine rm identihost-do
```

Это дает полную гарантию того, что корректно освобождены все ресурсы веб-интерфейса данного провайдера облачной среды.

А теперь рассмотрим некоторые альтернативы механизму Compose.

Присваивание значения переменной COMPOSE_FILE

Вместо того чтобы каждый раз в командной строке явно определять аргумент `-f prod.yml` для механизма Compose, можно один раз присвоить требуемое значение соответствующей переменной среды. Например:

```
$ COMPOSE_FILE=prod.yml
$ docker-compose up -d
...
```

После этого вместо определенного по умолчанию файла docker-compose.yml будет использоваться файл prod.yml.

Усовершенствованная генерация файла конфигурации

Методика использования шаблонов при создании файлов конфигурации для контейнеров Docker является достаточно широко распространенной в процессе контейнеризации приложений, особенно в тех случаях, когда сами приложения не обеспечивают поддержку переменных среды. При переходе от простых примеров, приведенных в данной книге, к реальным приложениям вам наверняка потребуется более мощный обработчик шаблонов, такой как Jinga2 или механизм обработки шаблонов языка Go, чтобы избежать непредвиденных ошибок, вызванных конфликтами в регулярных выражениях.

Проблема возникает настолько часто, что Джейсон Уайлдер (Jason Wilder) создал утилиту dockerize (https://github.com/jwilder/dockerize) для автоматизации процесса контейнеризации. Утилита генерирует файлы конфигурации из файла шаблонов и набора переменных среды, затем вызывает обычное приложение. Такой подход можно использовать для обертывания приложения в скрипты запуска, вызываемые из инструкции `CMD` или `ENTRYPOINT`.

Но Джейсон расширил возможности в еще большей степени, создав утилиту-генератор docker-gen (https://github.com/jwilder/docker-gen). Эта утилита может использовать значения из метаданных контейнера (например, IP-адрес), а также переменные среды. Кроме того, docker-gen может работать постоянно, реагируя на события механизма Docker, такие как создание нового контейнера, и обновляя соответствующим образом файлы конфигурации. Хорошим примером является

созданный Джейсоном контейнер nginx-proxy, автоматически добавляющий контейнеры с помощью переменной среды `VIRTUAL_HOST` для создания группы хостов с балансировкой нагрузки.

Варианты выполнения

После создания системы, полностью готовой к эксплуатации[1], пора подумать о запуске этой системы на реальном сервере[2]. До сих пор мы рассматривали механизмы Compose и Machine, но поскольку оба этих проекта являются относительно новыми и находятся в стадии активной разработки, будет разумно применять с большой осторожностью эти средства при эксплуатации большинства приложений, за исключением разве что небольших вспомогательных проектов (на момент выпуска данной книги подобные предостережения содержались и на вебсайте Docker). Оба проекта быстро развиваются и разрабатывают эксплуатационные функциональные возможности, и чтобы узнать о направлениях их развития, можно обратиться к дорожным картам в их репозиториях на GitHub, из которых можно понять, насколько эти средства готовы к промышленной эксплуатации.

Итак, если Compose не является приемлемым вариантом, что делать? Рассмотрим некоторые другие возможности. Во всех последующих примерах исходного кода предполагается, что используются образы, доступные в реестре Docker Hub, а не сгенерированные на сервере. Для соответствия этому предположению выгрузите собственные образы в реестр или используйте мои образы из Docker Hub (`amouat/identidock:1.0`, `amouat/dnmonster:1.0`, `amouat/proxy:1.0`).

Скрипты командной оболочки

Простейший способ запуска без использования Compose – написать небольшой скрипт командной оболочки, который выполняет команды Docker для инициализации контейнеров. Это будет работать достаточно успешно для большинства простых вариантов использования, а если добавить некоторые функции контроля, то с большой вероятностью вы будете знать обо всех возникающих проблемах, требующих особого внимания. Но при длительном сроке эксплуатации такое решение далеко от совершенства – обычно все заканчивается сопровождением запутанного и неструктурированного скрипта, в который постоянно добавляются все новые и новые функциональные возможности, появляющиеся в других вариантах решений.

Для контейнеров, преждевременно завершивших свою работу, можно обеспечить автоматический повторный запуск с помощью аргумента `--restart` в команде `docker run`. Этот аргумент определяет стратегию повторного запуска посредством

[1] В действительности система не вполне готова. Необходимо тщательно продумать способы защиты приложения, прежде чем приглашать для ознакомления с ним всех желающих. Более подробно о защите см. главу 13.

[2] Кроме того, необходимо решить, как будут поддерживаться контроль системы и фиксация событий в системных журналах. Об этом никогда не следует забывать. См. главу 10.

значений no (никогда), on-failure (после критического сбоя) или always (всегда). По умолчанию определено значение no, то есть контейнеры не перезапускаются автоматически. При выборе стратегии on-failure контейнеры будут перезапускаться только после выхода с ненулевым кодом возврата, кроме того, можно определить максимальное количество попыток перезапуска (например, команда docker run --restart on-failure:5 будет пытаться перезапустить контейнер пять раз, прежде чем откажется от дальнейших попыток).

Следующий скрипт, размещенный в файле *deploy.sh*, инициализирует и запускает наш сервис identidock:

```
#!/bin/bash
set -e

echo "Starting identidock system"

docker run -d --restart=always --name redis redis:3
docker run -d --restart=always --name dnmonster amouat/dnmonster:1.0
docker run -d --restart=always --link dnmonster:dnmonster --link redis:redis \
  -e ENV=PROD --name identidock amouat/identidock:1.0
docker run -d --restart=always --name proxy --link identidock:identidock -p 80:80 \
  -e NGINX_HOST=45.55.251.164 -e NGINX_PROXY=http://identidock:9090 \
  amouat/proxy:1.0

echo "Started"
```

Обратите внимание на то, что в действительности здесь выполнено преобразование содержимого файла *docker-compose.yml* в соответствующие команды оболочки. Но, в отличие от механизма Compose, здесь отсутствуют команды освобождения ресурсов и очистки после критических сбоев и команды, проверяющие наличие уже запущенных контейнеров.

Для работы с облачным провайдером Digital Ocean можно использовать команды ssh и scp для запуска сервиса identidock с помощью только что созданного скрипта командной оболочки:

```
$ docker-machine scp deploy.sh identihost-do:~/deploy.sh
deploy.sh                                     100%   575   0.6KB/s   00:00
$ docker-machine ssh identihost-do
...
$ chmod +x deploy.sh
$ ./deploy.sh
Starting identidock system
3b390441b16eaece94df7e0e07d1edcb4c11ce7232108849d691d153330c6dfb
57459e4c0c2a75d2fbcef978aca9344d445693d2ad6d9efe70fe87bf5721a8f4
5da04a34302b400ec08e9a1d59c3baeec14e3e65473533c165203c189ad58364
d1839d8de1952fca5c41e0825ebb27384f35114574c20dd57f8ce718ed67e3f5
Started
```

Разумеется, все эти команды можно было бы выполнить и вручную в командной оболочке. Главная причина, по которой предпочтение отдано скрипту, – возможность документирования и обеспечение переносимости, то есть если нужно

запустить identidock на новом хосте, то в файле скрипта можно найти конкретные инструкции для формирования аналогичной версии приложения.

При необходимости обновления образов или внесения каких-либо изменений можно воспользоваться механизмом Docker Machine для установления соединения локального клиента с удаленным Docker-сервером или зарегистрироваться непосредственно на удаленном сервере и там использовать клиента. Для выполнения обновления контейнера без остановки его работы потребуется наличие балансировщика нагрузки или обратного прокси-сервера перед контейнером, при этом необходимы следующие действия:

1. Инициализация нового контейнера с обновленным образом (лучше всего отказаться от попыток обновления самих работающих образов).
2. Изменить настройки балансировщика нагрузки таким образом, чтобы направить в новый образ весь трафик или его часть.
3. Протестировать работоспособность нового контейнера.
4. Отключить старый контейнер.

Также см. раздел «Тестирование в процессе эксплуатации» главы 8, в котором описаны различные методики развертывания обновлений без остановки работы сервисов.

Некорректность соединений при перезапуске
В более старых версиях Docker возникали проблемы, когда соединения становились некорректными при перезапуске контейнеров. Если такая проблема появилась, проверьте, используете ли вы достаточно свежую версию Docker. Во время написания книги я пользовался Docker версии 1.8, которая работала правильно, и все изменения IP-адресов контейнеров автоматически передавались в связанные соединениями контейнеры. Также следует отметить, что обновляется только файл */etc/hosts*, а переменные среды не корректируются при изменениях в связанных контейнерах.

В конце этого раздела мы рассмотрим возможности управления запуском и развертыванием контейнеров с использованием существующей методики, с которой вы, возможно, уже знакомы. В главе 12 будет описано более новое, специализированное для Docker инструментальное средство, выполняющее ту же задачу.

Использование диспетчера процессов или systemd для глобального управления

Вместо скрипта командной оболочки и функциональных возможностей Docker для перезапуска контейнеров можно воспользоваться диспетчером процессов (задач) или системой инициализации, такой как systemd или upstart, для запуска контейнеров. Это особенно удобно, если на хосте имеются сервисы, не размещенные в контейнерах, но зависящие от одного или нескольких контейнеров. При этом необходимо учитывать следующие аспекты:

- нужна полная уверенность в том, что функции автоматического перезапуска контейнеров Docker не используются, то есть нельзя применять аргумент `--restart=always` в командах `docker run`;

- обычно диспетчер процессов в конечном итоге контролирует процесс `docker client`, а не процессы внутри контейнера. В большинстве случаев этого достаточно, но если разрывается сетевое соединение или возникает какая-либо другая ошибка, то Docker-клиент завершает работу, но контейнер продолжает работать, и это становится источником проблем. Хотелось бы, чтобы диспетчер процессов управлял основным процессом внутри контейнера. Возможно, в будущем ситуация изменится к лучшему, но пока рекомендую ознакомиться с проектом systemd-docker (https://github.com/ibuildthecloud/systemd-docker), ориентированным на обеспечение управления контейнерами на основе механизма cgroup. (Более подробную информацию по этому вопросу см. на соответствующей странице GitHub https://github.com/docker/docker/issues/6791.)

Чтобы продемонстрировать на практике возможности управления контейнерами с помощью systemd, можно воспользоваться одним из файловых сервисов для запуска identidock на хосте, использующим systemd. Для этого примера я использовал дистрибутив CentOS 7, но подойдут и другие дистрибутивы, в которых применяется systemd. Пример с использованием upstart (система инициализации Ubuntu) не рассматривается, так как почти все основные дистрибутивы Linux постепенно переходят на systemd. Все файлы примера должны размещаться в каталоге */etc/systemd/system/*.

Начнем с описания файлового сервиса для контейнера Redis – *identidock.redis.servise*, который не зависит от каких-либо других контейнеров:

```
[Unit]
Description=Redis Container for Identidock
After=docker.service
Requires=docker.service  ❶

[Service]
TimeoutStartSec=0  ❷
Restart=always
ExecStartPre=-/usr/bin/docker stop redis  ❸
ExecStartPre=-/usr/bin/docker rm redis
ExecStartPre=/usr/bin/docker pull redis  ❹
ExecStart=/usr/bin/docker run --rm --name redis redis

[Install]
WantedBy=multi-user.target
```

❶ Нужна полная уверенность в том, что Docker инициализирован до запуска контейнера.
❷ Поскольку для выполнения команд Docker требуется некоторое время, проще всего совсем отключить тайм-аут.
❸ Перед запуском контейнера сначала нужно удалить все старые контейнеры с тем же именем, а это значит, что кэш Redis будет удален при перезапуске. Но для identidock это не является проблемой. Использование символа дефиса `'-'` перед командой сообщает systemd о том, что не следует останавливать инициализацию данного сервиса, если эта команда возвращает ненулевой код.
❹ Явная загрузка образа гарантирует запуск самой новой версии.

Сервис *identidock.identidock.service* похож на предыдущий, но зависит от других сервисов:

```
[Unit]
Description=identidock Container for Identidock
After=docker.service
Requires=docker.service
After=identidock.redis.service  ❶
Requires=identidock.redis.service
After=identidock.dnmonster.service
Requires=identidock.dnmonster.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop identidock
ExecStartPre=-/usr/bin/docker rm identidock
ExecStartPre=/usr/bin/docker pull amouat/identidock
ExecStart=/usr/bin/docker run --name identidock --link dnmonster:dnmonster \
  --link redis:redis -e ENV=PROD amouat/identidock

[Install]
WantedBy=multi-user.target
```

❶ Кроме самого механизма Docker, необходимо объявить о зависимости от других контейнеров, используемых в identidock, в данном случае это контейнеры Redis и dnmonster. Необходимы оба параметра `After` и `Requires`, чтобы избежать состояний гонки при инициализации.

Для сервиса proxy (полное имя файла *identidock.proxy.service*) описание выглядит следующим образом:

```
[Unit]
Description=Proxy Container for Identidock
After=docker.service
Requires=docker.service
Requires=identidock.identidock.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop proxy
ExecStartPre=-/usr/bin/docker rm proxy
ExecStartPre=/usr/bin/docker pull amouat/proxy
ExecStart=/usr/bin/docker run --name proxy --link identidock:identidock -p 80:80 \
  -e NGINX_HOST=0.0.0.0 -e NGINX_PROXY=http://identidock:9090 amouat/proxy

[Install]
WantedBy=multi-user.target
```

И последний требуемый сервис (файл с именем *identidock.dnmonster.service*):

```
[Unit]
Description=dnmonster Container for Identidock
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop dnmonster
ExecStartPre=-/usr/bin/docker rm dnmonster
ExecStartPre=/usr/bin/docker pull amouat/dnmonster
ExecStart=/usr/bin/docker run --name dnmonster amouat/dnmonster

[Install]
WantedBy=multi-user.target
```

Теперь можно запустить сервис identidock командой `systemctl start identidock.*`. Главное различие между этой системой и функцией перезапуска Docker состоит в том, что в systemd перезапуск любого остановленного контейнера инициализирует цепочку перезапусков всех связанных с ним контейнеров: если остановилась работа контейнера Redis и нужен его перезапуск, то будут также перезапущены контейнеры identidock и proxy. В Docker принят совершенно другой подход, поскольку сам механизм знает, как полностью обновить соединения без перезапуска контейнера.

Несмотря на упомянутые выше проблемы, следует отметить, что и CoreOS, и сервис Giant Swarm PaaS используют именно systemd для управления контейнерами. Здесь также уместно сказать, что существует неразрешенный конфликт между Docker и systemd – обе системы стремятся к абсолютному управлению жизненным циклом сервисов, работающих на хосте.

Использование инструментальных средств управления конфигурацией

Если ваша организация отвечает за сопровождение достаточно большого количества хостов, то, скорее всего, для этого используется одно из *инструментальных средств управления конфигурацией* (*configuration management – CM*) (если не используется, то следует основательно подумать об этом). Во всех проектах должна учитываться возможность своевременного обновления операционной системы на Docker-хосте, особенно если обновления повышают степень защиты. С другой стороны, нужна полная уверенность в том, что все работающие образы Docker представлены новейшими версиями и нет необходимости составлять запутанные комбинации из различных версий используемого ПО. Решения по управлению конфигурацией, такие как Puppet, Chef, Ansible и Salt, специально предназначены для решения проблем, возникающих при управлении многочисленными вариантами и версиями ПО.

Можно предложить два способа использования инструментов конфигурации для контейнеров:

1. Считать контейнеры виртуальными машинами и использовать инструментальное ПО для управления и обновления ПО, работающего внутри контейнеров.
2. Использовать инструментальное ПО для управления Docker-хостом и с его помощью предоставлять для запуска контейнеров корректные версии образов, при этом рассматривать сами контейнеры как черные ящики, которые можно заменять, но нельзя модифицировать.

Первый подход вполне осуществим, но это не соответствует принципам Docker. Предпочтительнее работать с файлами Dockerfile в соответствии с философией «небольшой контейнер с единственным процессом», на которой основана вся система Docker. В оставшейся части данного раздела мы сосредоточимся на втором варианте, в большей степени соответствующем философии Docker и технологии микросервисов.

В соответствии с этой методикой контейнеры считаются подобными *«золотым образам»* (*golden images*), как их называют на специфическом языке виртуальных машин, и не должны изменяться после запуска. При необходимости обновления заменяется весь работающий контейнер в целом на новый образ без попыток изменить что-либо внутри активного образа. Основное преимущество такого подхода: возможность точно узнать, что именно запускается в контейнере, просто взглянув на тег образа (при условии, что применяется правильная система назначения тегов и теги не используются повторно).

Рассмотрим пример практической реализации описанной методики.

Ansible

Для этого примера мы используем Ansible (http://www.ansible.com/), потому что это широко известное и распространенное инструментальное средство управления конфигурацией, его легко освоить, к тому же это продукт с открытым исходным кодом. Но это вовсе не означает, что другие инструменты лучше или хуже.

В отличие от других средств управления конфигурацией, Ansible не требует специальной установки агентов на хосты. Вместо этого используется механизм SSH для конфигурирования хостов.

Существует Docker-модуль Ansible с функциональностью, обеспечивающей создание контейнеров и организацию их работы. Возможно использование Ansible в файлах Dockerfile для установки и конфигурирования ПО, но здесь мы будем рассматривать только вариант применения Ansible для настройки виртуальной машины с образом identidock. Поскольку вся работа выполняется на одном хосте, возможности Ansible не задействованы в полной мере, но пример демонстрирует, насколько успешно можно организовать совместное использование Ansible и Docker.

Вместо установки Ansible-клиента есть возможность воспользоваться его образом из реестра Docker Hub. Официального образа не существует, но образ `generik/ansible` вполне пригоден для тестирования.

Начнем с создания файла *hosts*, содержащего список всех серверов, которыми должен будет управлять Ansible. В этот файл записывается IP-адрес конкретного удаленного хоста или виртуальной машины.

```
$ cat hosts
[identidock]
46.101.162.242
```

Теперь необходимо создать «сценарий» установки identidock. Создайте файл *identidock.yml* со следующим содержимым, заменяя имена образов на реально используемые вами:

```
---
- hosts: identidock
  sudo: yes
  tasks:
  - name: easy-install
    apt: pkg=python-setuptools
  - name: pip
    easy-install: name=pip
  - name: docker-py
    pip: name=docker-py
  - name: redis container
    docker:
      name: redis
      image: redis:3
      pull: always
      state: reloaded
      restart_policy: always
  - name: dnmonster container
    docker:
      name: dnmonster
      image: amouat/dnmonster:1.0
      pull: always
      state: reloaded
      restart_policy: always
  - name: identidock container
    docker:
      name: identidock
      image: amouat/identidock:1.0
      pull: always
      state: reloaded
      links:
        - "dnmonster:dnmonster"
        - "redis:redis"
      env:
        ENV: PROD
      restart_policy: always
  - name: proxy container
    docker:
      name: proxy
      image: amouat/proxy:1.0
      pull: always
```

```
    state: reloaded
    links:
      - "identidock:identidock"
    ports:
      - "80:80"
    env:
      NGINX_HOST: www.identidock.com
      NGINX_PROXY: http://identidock:9090
    restart_policy: always
```

Бóльшая часть файла конфигурации очень похожа на данные для Docker Compose, но необходимо обратить внимание на следующее:

- обязательность установки на хост `docker-py` для обеспечения возможности использования модуля Docker Ansible. Это, в свою очередь, потребует установки некоторых модулей языка Python для удовлетворения зависимостей;
- переменная `pull` определяет, когда проверять образы Docker на наличие обновлений. Установка значения `always` гарантирует, что Ansible будет проверять наличие новых версий образов при каждом выполнении данной задачи;
- переменная `state` определяет, в каком состоянии должен находиться контейнер. Установка значения `reloaded` приводит к перезапуску контейнера после любого изменения конфигурации.

В действительности доступно гораздо большее количество параметров конфигурации, но приведенный выше конфигурационный файл дает нам схему, очень похожую на другие варианты конфигурации, описанные в данной главе.

Осталось только выполнить созданный сценарий:

```
$ docker run -it \
    -v ${HOME}/.ssh:/root/.ssh:ro  ❶
    -v $(pwd)/identidock.yml:/ansible/identidock.yml \
    -v $(pwd)/hosts:/etc/ansible/hosts \
    --rm=true generik/ansible ansible-playbook identidock.yml

PLAY [identidock] *************************************************************

GATHERING FACTS ***************************************************************
(СБОР ДАННЫХ)
The authenticity of host '46.101.41.99 (46.101.41.99)' can't be established.
(Аутентичность хоста '46.101.41.99 (46.101.41.99)' невозможно установить.)
ECDSA key fingerprint is SHA256:R0LfM7Kf3OgRmQmgxINko7SonsGAC0VJb27LTotGEds.
Are you sure you want to continue connecting (yes/no)? yes
(Вы действительно хотите продолжить установление соединения (да/нет)? да)
Enter passphrase for key '/root/.ssh/id_rsa':
(Введите пароль для ключа '/root/.ssh/id_rsa':)
ok: [46.101.41.99]

TASK: [easy_install] **********************************************************
changed: [46.101.41.99]

TASK: [pip] *******************************************************************
changed: [46.101.41.99]
```

```
TASK: [docker-py] *************************************************************
changed: [46.101.41.99]

TASK: [redis container] *******************************************************
changed: [46.101.41.99]

TASK: [dnmonster container] ***************************************************
changed: [46.101.41.99]

TASK: [identidock container] **************************************************
changed: [46.101.41.99]

TASK: [proxy container] *******************************************************
changed: [46.101.41.99]

PLAY RECAP ********************************************************************
(ИТОГ ВЫПОЛНЕНИЯ СЦЕНАРИЯ)
46.101.41.99               : ok=8    changed=7    unreachable=0    failed=0
$ curl 46.101.41.99
<html><head><title>Hello...
```

❶ Эта команда необходима, чтобы установить соответствие для пары SSH-ключей, используемых для доступа к удаленному серверу.

Вся процедура займет некоторое время, так как Ansible нужно будет загрузить указанные образы. После завершения этой процедуры приложение identidock будет полностью готово к работе.

В этом примере продемонстрирована лишь малая часть функциональных возможностей Ansible. На самом деле с помощью этого инструмента можно сделать гораздо больше, особенно в плане определения процессов, выполняющих непрерывные обновления контейнеров без нарушения зависимостей и почти без остановок работы.

Конфигурация хоста

До сих пор в данной главе предполагалось, что контейнеры работают на основе дроплета Docker (*дроплет* – *droplet* – это специальный термин облачной среды Digital Ocean, обозначающий предварительно сконфигурированную виртуальную машину), предоставленного облачным сервисом Digital Ocean (который во время написания данной книги использовал Ubuntu 14.04). Но существует множество других вариантов выбора операционной системы хоста и его инфраструктуры со своими достоинствами и недостатками. Если вы отвечаете за работу локального ресурса организации, то изучению вариантов выбора следует уделить особое внимание.

Несмотря на то что для организации работы Docker-хостов вполне пригодны компьютеры без каких-либо специализированных программных средств (как в локальной, так и в облачной среде), в настоящее время наиболее распространено использование виртуальных машин. Большинство организаций уже имеет в своем распоряжении те или иные типы сервиса виртуальных машин, которые можно ис-

пользовать при создании хостов для контейнеров; при этом обеспечивается высокая степень изоляции и защиты пользователей.

Выбор операционной системы

В настоящее время в рассматриваемой области предлагается несколько вариантов с различными достоинствами и недостатками. Если нужна поддержка работы небольшого или среднего приложения, возможно, проще всего выбрать то, что уже хорошо вам известно: если есть опыт работы с Ubuntu или Fedora, то используйте эти дистрибутивы (но при этом следует помнить о проблемах с драйверами файловых систем, о чем речь пойдет немного ниже). С другой стороны, если необходима поддержка особо крупного приложения или кластера (сотен или тысяч контейнеров, распределенных по множеству хостов), то лучше обратить внимание на специализированные решения, такие как CoreOS, Project Atomic или RancherOS, а также на решения по оркестрации контейнеров, которые будут рассматриваться в главе 12.

Для большинства облачных хостов уже имеются в наличии Docker-образы, готовые к практическому применению, и при необходимости их можно использовать для тестирования и организации работы в соответствующей инфраструктуре.

Выбор драйвера файловой системы

В настоящее время Docker поддерживает несколько драйверов файловых систем, кроме того, планируется расширение списка поддерживаемых файловых систем. Выбор правильного драйвера файловой системы крайне важен для обеспечения надежности и эффективности в процессе эксплуатации. Какой драйвер считать самым лучшим, зависит от конкретного варианта использования и от вашего практического опыта. На сегодня доступны следующие драйверы файловых систем:

- *AUFS* – самый первый драйвер файловой системы для Docker. На настоящий момент с большой вероятностью можно утверждать, что это наиболее тщательно проверенный и широко используемый драйвер. Вместе с Overlay обладает главным преимуществом – поддержкой совместного использования страниц памяти несколькими контейнерами – если два контейнера загружают библиотеки или данные из одного и того же нижележащего уровня, то ОС имеет возможность предоставить единую страницу памяти для обоих контейнеров. Основной недостаток AUFS – отсутствие поддержки в ядре, хотя уже в течение некоторого времени в Debian и Ubuntu эта файловая система используется. Кроме того, AUFS работает на уровне файлов, поэтому даже при незначительном изменении в большом файле весь этот файл будет копироваться в уровень чтения/записи контейнера. Для сравнения отметим, что BTRFS и Device mapper работают на уровне дисковых блоков, поэтому более эффективно используют дисковое пространство при обработке больших файлов. Если в текущий момент вы используете хост под управлением Debian или Ubuntu, то определенно имеет смысл выбор драйвера AUFS;

- *Overlay* – драйвер очень похож на AUFS, его поддержка появилась в ядре Linux, начиная с версии 3.18. Overlay с большой вероятностью станет в будущем основным драйвером файловой системы, поэтому должен обеспечивать несколько более высокую производительность, по сравнению с AUFS. В настоящий момент главными его недостатками являются необходимость использования самых новых версий ядер (для которых в большинстве дистрибутивов потребуется внесение специфических изменений (patching)), а также то, что он недостаточно тщательно протестирован, по сравнению с AUFS и некоторыми другими вариантами;
- *BTRFS* – файловая система с поддержкой механизма копирования при записи (copy-on-write – COW)[1], основное внимание в которой сосредоточено на обеспечении устойчивости при критических сбоях и на поддержке весьма больших размеров файлов и томов. Так как для BTRFS характерно применение специфических уловок и трюков (это особенно касается работы с фрагментами (chunks)), рекомендовать ее можно только тем организациям, которые уже имеют опыт использования BTRFS или которым необходимы особенные возможности BTRFS, не поддерживаемые другими драйверами. Возможно, это станет удачным выбором, если контейнеры считывают и записывают очень большие файлы и требуется поддержка на уровне блоков;
- *ZFS* – эта весьма популярная файловая система была разработана компанией Sun Microsystems. Во многих отношениях похожа на BTRFS, но по некоторым спорным утверждениям обладает более высокой производительностью и надежностью. Запуск ZFS в Linux является непростой процедурой, поскольку включение ее в ядро невозможно из-за проблем с лицензированием. Поэтому рекомендовать ее можно только для использования в организациях, обладающих значительным практическим опытом применения ZFS;
- *Device mapper* – применяется по умолчанию в системах Red Hat. Device mapper – это драйвер ядра, используемый в качестве основы для некоторых других технологий, в том числе для RAID, для шифрования устройств и для организации снимков (snapshots). Docker использует функцию *«динамического выделения»* (*thin provisioning*) Device mapper[2], ориентированную на выполнение операции копирования при записи на уровне блоков, а не файлов. «Пул динамического резервирования» создается на основе разреженного файла, по умолчанию имеющего размер 100 Гб. Контейнерам выделяется

---

[1] Я не знаю, как правильно произносить BTRFS: некоторые говорят «ButterFS», некоторые – «BetterFS». Я говорю «FSCK» (fsck – Unix-утилита для проверки и восстановления файловых систем).

[2] Вместо немедленного предоставления всех требуемых ресурсов клиенту методика тонкого резервирования предлагает выделение ресурсов только по запросу клиента. Это полная противоположность «предварительному резервированию», при котором клиент сразу получает в свое распоряжение все необходимые ресурсы, даже если он фактически использует только их часть.

файловая система, поддерживаемая этим пулом и при создании по умолчанию не превышающая лимит в 100 Гб (в Docker 1.8). Так как файлы разреженные, в действительности они занимают намного меньше дискового пространства, но контейнер не может превысить предел 100 Гб без изменения значений по умолчанию. Некоторые считают, что Device mapper является самым сложным в использовании драйвером файловых систем в Docker, представляет собой постоянный источник проблем и требует специфической поддержки. Я бы рекомендовал по возможности отказаться от Device mapper и воспользоваться одним из альтернативных вариантов. Но если уж вы окончательно решили применять именно этот драйвер, помните об огромном количестве параметров и характеристик, которые можно настроить для обеспечения максимальной производительности (в частности, удачным решением будет перемещение файловой системы (хранилища данных) с принятого по умолчанию устройства «обратной петли» (loopback) на реально существующее устройство);

- *VFS* – принятая по умолчанию в ОС Linux виртуальная файловая система (Virtual File System). В ней не реализован механизм копирования при записи, поэтому требуется создание полной копии образа при инициализации контейнера. Это существенно замедляет запуск контейнеров и значительно увеличивает требуемый объем дискового пространства, занимаемого копиями образов. Преимущество состоит в том, что эта файловая система проста и не нуждается в поддержке какими-либо особенными функциями ядра. VFS может стать разумным выбором при возникновении проблем с другими драйверами и при отсутствии необходимости обеспечения сверхвысокой производительности (например, если вы используете небольшое количество контейнеров с длинным жизненным циклом).

Если у вас нет особенных причин для выбора альтернативных вариантов, то я бы предложил остановиться на использовании AUFS или Overlay, несмотря на то что для них требуется модификация ядра ОС.

Смена драйвера файловой системы

Сменить драйвер файловой системы достаточно просто, если у вас установлены все необходимые для этого зависимости. Просто перезапустите демон Docker, при этом передавая ему соответствующее значение аргумента `--storage-driver` (или `-s` в кратком варианте). Например, команда `docker daemon -s overlay` запускает демон с драйвером Overlay, если текущее ядро поддерживает его. Следует обратить внимание на важный аргумент `--graph` или `-g`, который устанавливает корневой каталог для среды времени выполнения Docker, – иногда необходимо явно указать для этого раздел с соответствующей файловой системой (например, `docker daemon -s btrfs -g /mnt/btrfs_partition` для драйвера BTRFS).

Чтобы сделать изменение постоянным, нужно отредактировать скрипт запуска или файл конфигурации сервиса Docker. Для Ubuntu 14.04 необходимо изменить значение переменной `DOCKER_OPTS` в файле */etc/default/docker*.

Использование образов с различными драйверами файловой системы
При смене драйвера файловой системы вы теряете доступ ко всем старым контейнерам и образам. Возврат к предыдущему драйверу восстанавливает возможность доступа. Для переключения образа на новый драйвер файловой системы просто сохраните нужный образ в архивный tar-файл, затем распакуйте этот архив в новой файловой системе. Например:

```
$ docker save -o /tmp/debian.tar debian:wheezy
$ sudo stop docker
$ docker daemon -s vfs
...
```

В новом терминале выполните следующие команды:

```
$ docker images
REPOSITORY   TAG      IMAGE ID        CREATED      VIRTUAL SIZE
$ docker load -i /tmp/debian.tar
$ docker images
REPOSITORY   TAG      IMAGE ID        CREATED      VIRTUAL SIZE
debian       wheezy   b3d362b223ec1   2 days ago   84.96 MB
```

Специализированные варианты хостинга

Существует несколько специально подготовленных вариантов хостинга контейнеров, которые не требуют прямого вмешательства в управление хостами.

Triton

Triton от компании Joyent (https://www.joyent.com/) представляет собой, возможно, самый интересный вариант, поскольку не использует внутренних виртуальных машин. Это дает значительное преимущество в производительности, по сравнению с решениями на основе виртуальных машин, и позволяет распределять ресурсы непосредственно по контейнерам.

Triton не использует механизма Docker, но его собственный контейнерный механизм работает на гипервизоре SmartOS (истоки которого следует искать в ОС Solaris) на основе системы виртуализации Linux Virtualization. Благодаря реализации удаленного API Docker Triton полностью совместим с обычным Docker-клиентом, который используется в качестве стандартного интерфейса к Triton. Образы из реестра Docker Hub работают, как обычно.

Triton является продуктом с открытым исходным кодом и доступен в двух версиях: хост-версия, работающая в облачной среде Joyent, и версия для локального применения. Наше приложение identidock можно легко и быстро подготовить к работе в общедоступной открытой облачной среде Joyent. После создания учетной записи в Triton и установления связи Docker-клиента с Triton попробуйте выполнить команду выдачи информации:

```
$ docker info
Containers: 0
```

```
Images: 0
Storage Driver: sdc
 SDCAccount: amouat
Execution Driver: sdc-0.3.0
Logging Driver: json-file
Kernel Version: 3.12.0-1-amd64
Operating System: SmartDataCenter
CPUs: 0
Total Memory: 0 B
Name: us-east-1
ID: 92b0cf3a-82c8-4bf2-8b74-836d1dd61003
Username: amouat
Registry: https://index.docker.io/v1/
```

Обратите внимание на значения, определяющие операционную систему и драйвер выполнения, которые указывают, что сейчас мы не работаем с обычным механизмом Docker. Для запуска identidock можно воспользоваться инструментом Docker Compose и приведенным ниже файлом *triton.yml*, поскольку Triton поддерживает большинство функциональных возможностей прикладного программного интерфейса механизма Docker:

```
proxy:
  image: amouat/proxy:1.0
  links:
    - identidock
  ports:
    - "80:80"
  environment:
    - NGINX_HOST=www.identidock.com
    - NGINX_PROXY=http://identidock:9090
  mem_limit: "128M"
identidock:
  image: amouat/identidock:1.0
  links:
    - dnmonster
    - redis
  environment:
    ENV: PROD
  mem_limit: "128M"
dnmonster:
  image: amouat/dnmonster:1.0
  mem_limit: "128M"
redis:
  image: redis
  mem_limit: "128M"
```

Содержимое этого файла почти полностью совпадает с содержимым файла *prod.yml*, приводимым ранее, с добавлением параметров ограничения памяти, сообщающих механизму Triton размеры запускаемых контейнеров. Здесь также

используются готовые образы из реестра, а не созданные локально (в настоящее время Triton не поддерживает команду `docker build`).

Запуск приложения:

```
$ docker-compose -f triton.yml up -d
...
Creating triton_proxy_1...
$ docker inspect -f {{.NetworkSettings.IPAddress}} triton_proxy_1
165.225.128.41
$ curl 165.225.128.41
<html><head><title>Hello...
```

Triton автоматически использует общедоступный IP-адрес, если видит открытый для доступа без ограничений порт.

После завершения работы контейнеров на Triton не забывайте останавливать и удалять их, так как Triton берет плату за остановленные, но не удаленные контейнеры.

Использование собственных средств Docker для взаимодействия с Triton является большим преимуществом, но присутствуют и некоторые недостатки: поддерживаются не все вызовы функций прикладного программного интерфейса, есть определенные проблемы при обеспечении работы Compose с томами, но эти неудобства должны быть устранены в ближайшее время.

До тех пор, пока основные облачные провайдеры не обеспечат условия изоляции ядра Linux, достаточно строгие для того, чтобы появилась возможность работы контейнеров без дополнительных средств защиты, Triton будет являться одним из наиболее привлекательных решений для организации функционирования систем контейнеризации.

Google Container Engine

Google Container Engine (GKE) (https://cloud.google.com/container-engine/) применяет собственную независимую методику контейнеризации, основанную на системе управления Kubernetes.

Kubernetes – это проект корпорации Google с открытым исходным кодом, в котором учтен опыт, полученный при эксплуатации контейнеров в среде корпоративного менеджера кластеров Borg[1].

Развертывание приложения в среде GKE требует понимания основ работы Kubernetes, а также создания нескольких специализированных для Kubernetes файлов конфигурации, которые будут рассматриваться в разделе «Kubernetes» главы 12.

Затратив некоторые дополнительные усилия на конфигурирование своего приложения, вы получаете набор сервисов, таких как автоматическая репликация и ба-

[1] См. документ «Large-scale cluster management at Google with Borg» (https://research.google.com/pubs/pub43438.html), обстоятельно описывающий подробности организации работы кластера, выполняющего сотни тысяч задач.

лансировка нагрузки. Может показаться, что такие сервисы необходимы только для крупных сетевых приложений с высоким трафиком и множеством распределенных компонентов, но эти функциональные возможности быстро становятся важными для любого сервиса, для которого предполагается интенсивное развитие в будущем.

Настоятельно рекомендую Kubernetes и особенно GKE для развертывания систем контейнеров, но при этом помните, что принятие этого решения привязывает вас к модели Kubernetes, затрудняя распространение системы для других провайдеров.

Amazon EC2 Container Service

Amazon EC2 Container Service (ECS) (https://aws.amazon.com/ecs/) помогает организовать работу контейнеров в инфраструктуре Amazon EC2. ECS предоставляет веб-интерфейс и прикладной программный интерфейс для запуска контейнеров и управления расположенным уровнем ниже кластером EC2.

На каждом узле кластера ECS запускает контейнер-агент, который обменивается данными с ECS-сервисом и отвечает за запуск, остановку и контроль контейнеров.

Запуск identidock в ECS выполняется относительно быстро, хотя и требует описания типового AWS-интерфейса со множеством параметров конфигурации. После регистрации в системе ECS и создания кластера необходимо выгрузить «Определение задачи» (Task Definition) для identidock. Приведенный ниже JSON-фрагмент можно использовать в качестве такого определения:

```
{
  "family": "identidock",
  "containerDefinitions": [
    {
      "name": "proxy",
      "image": "amouat/proxy:1.0",
      "cpu": 100,
      "memory": 100,
      "environment": [
        {
          "name": "NGINX_HOST",
          "value": "www.identidock.com"
        },
        {
          "name": "NGINS_PROXY",
          "value": "http://identidock:9090"
        }
      ],
      "portMappings": [
        {
          "hostPort": 80,
          "containerPort": 80,
          "protocol": "tcp"
```

```
        }
      ],
      "links": [
        "identidock"
      ],
      "essential": true
    },
    {
      "name": "identidock",
      "image": "amouat/identidock:1.0",
      "cpu": 100,
      "memory": 100,
      "environment": [
        {
          "name": "ENV",
          "value": "PROD"
        }
      ],
      "links": [
        "dnmonster",
        "redis"
      ],
      "essential": true
    },
    {
      "name": "dnmonster",
      "image": "amouat/dnmonster:1.0",
      "cpu": 100,
      "memory": 100,
      "essential": true
    },
    {
      "name": "redis",
      "image": "redis:3",
      "cpu": 100,
      "memory": 100,
      "essential": false
    }
  ]
}
```

Для каждого контейнера необходимо определить объем памяти (в мегабайтах) и количество процессоров. Ключ `essential` определяет, должна ли быть остановлена данная задача при возникновении критического сбоя контейнера. В нашем случае контейнер Redis можно считать некритичным, так как приложение способно продолжать работу и без него. Смысл значений других полей очевиден.

После успешного создания задачи необходимо запустить ее в кластере. Приложение identidock должно быть запущено как *сервис*, а не как одноразовая задача.

Запуск в качестве сервиса означает, что ECS будет контролировать контейнеры, обеспечивая их доступность, и предоставлять возможность установления соединения с механизмом балансировки нагрузки Amazon Elastic Load Balancer для правильного распределения трафика между экземплярами. При создании сервиса ECS запрашивает его имя и требуемое количество экземпляров задач. После создания сервиса и небольшой паузы при его запуске предоставляется доступ к identidock по IP-адресу экземпляра EC2. Адрес можно найти на странице подробного описания экземпляра задачи, в разделе информации о контейнере proxy.

Остановка сервиса и освобождение связанных с ним ресурсов выполняются в несколько этапов. Сначала нужно обновить состояние сервиса, изменив количество задач на 0, чтобы механизм ECS не пытался перезапускать задачи при остановке сервиса. После этого сервис можно удалить. Перед удалением кластера нужно обязательно отменить регистрацию экземпляров контейнера. Кроме того, не забывайте об остановке и освобождении всех ранее выделенных ресурсов, таких как Elastic Load Balancer или хранилища данных EBS.

Внутри механизма ECS выполняется множество скрытых технологических операций, которые позволяют без затруднений запустить сотни и даже тысячи контейнеров парой щелчков мышью и предоставляют мощные функциональные возможности для масштабирования приложений. Распределение контейнеров по хостам конфигурируется в мельчайших подробностях, позволяя пользователям производить оптимизацию в соответствии с потребностями, достигая максимальной эффективности или максимальной надежности. Пользователи могут заменить принятый по умолчанию планировщик ECS на собственный или воспользоваться независимым решением, таким как Marathon (см. раздел «Mesos и Marathon» главы 12).

Кроме того, ECS объединен с существующими функциональными возможностями Amazon, такими как балансировщик Elastic Load Balancer, предназначенный для распределения нагрузки между несколькими экземплярами сервиса, и хранилище Elastic Block Store, служащее для постоянного хранения данных.

Giant Swarm

Giant Swarm позиционируется как «частное решение для архитектур микросервисов», что в действительности означает быстрый и простой способ запуска систем на основе Docker с использованием специализированного формата конфигурации. Giant Swarm предлагает хост-версию на совместно используемом кластере, а также специальное обслуживание (Giant Swarm предоставляет выделенные хосты без лишнего ПО и их сопровождение для клиента) и локальное решение на площадях клиента. На момент написания книги вариант совместно используемого кластера находился на стадии альфа-тестирования, но вариант спецобслуживания выделенных хостов был вполне готов к эксплуатации.

Giant Swarm представляет тот редкий случай, когда виртуальные машины используются по минимуму или вообще не используются. Пользователи с жесткими требованиями по обеспечению безопасности получают в свое распоряжение от-

дельные хосты без лишнего ПО, но в совместно используемом кластере бок о бок работают контейнеры от различных пользователей.

Рассмотрим подробнее процедуру запуска identidock в совместно используемом кластере Giant Swarm. Предположим, что есть доступ к Giant Swarm и установлен соответствующий интерфейс командной строки swarm[1]. Начнем с создания показанного ниже файла конфигурации с именем *swarm.json*:

```
{
    "name": "identidock_svc",
    "components": {
        "proxy": {
            "image": "amouat/proxy:1.0",
            "ports": [80],
            "env": {
                "NGINX_HOST": "$domain",
                "NGINX_PROXY": "http://identidock:9090"
            },
            "links": [ {
                "component": "identidock",
                "target_port": 9090
            }],
            "domains": { "80": "$domain" }
        },
        "identidock": {
            "image": "amouat/identidock:1.0",
            "ports": [9090],
            "links": [
            {
                "component": "dnmonster",
                "target_port": 8080
            },
            {
                "component": "redis",
                "target_port": 6379
            }
            ]
        },
        "redis": {
            "image": "redis:3",
            "ports": [6379]
        },
        "dnmonster": {
            "image": "amouat/dnmonster:1.0",
            "ports": [8080]
        }
    }
}
```

[1] Это не имеет никакого отношения к кластерному решению Docker, которое тоже называется Swarm.

Теперь можно запустить identidock:

```
$ swarm up --var=domain=identidock-$(swarm user).gigantic.io
Starting service identidock_svc...
(Запуск сервиса identidock_svc...)
Service identidock_svc is up.
(Сервис identidock_svc запущен.)
You can see all components using this command:
(Все компоненты можно увидеть, выполнив следующую команду:)

    swarm status identidock_svc

$ swarm status identidock_svc
Service identidock_svc is up
(Сервис identidock_svc запущен)

component   image                   instanceid     created               status
dnmonster   amouat/dnmonster:1.0    m6eyoilfiei1   2015-09-04 09:50:40   up
identidock  amouat/identidock:1.0   r22ut7h0vx39   2015-09-04 09:50:40   up
proxy       amouat/proxy:1.0        6dr38cmrg3nx   2015-09-04 09:50:40   up
redis       redis:3                 jvcf15d6lpz4   2015-09-04 09:50:40   up
$ curl identidock-amouat.gigantic.io
<html><head><title>Hello...
```

Здесь показана одна из функциональных возможностей, которая отличает файлы конфигурации Giant Swarm от файлов Docker Compose – возможность использования шаблонных переменных. В нашем случае передается имя хоста, заданное в командной строке, а swarm продолжает обработку и подставляет это значение вместо шаблона `$domain` в файле *swarm.json*. Среди других возможностей, предоставляемых конфигурацией swarm.json, можно выделить возможность определять *pods* – *группы контейнеров* с совместным планированием их работы, а также возможность определения количества запускаемых контейнеров.

В дополнение к интерфейсу командной строки swarm предлагается графический пользовательский веб-интерфейс для контроля сервисов и просмотра журналов регистрации событий, а также прикладной программный интерфейс REST для автоматизации взаимодействия с Giant Swarm.

Контейнеры для постоянно хранимых данных и для промышленной эксплуатации

Пока нет единого мнения о степени изменения методики хранения данных при использовании Docker, по крайней мере для больших объемов данных. Если вы работаете с собственными базами данных, то можете выбрать контейнеры Docker, или виртуальные машины, или чисто аппаратное решение. В любом случае, при больших объемах данных все сводится к тому, что виртуальная машина или контейнер фактически привязывается к хост-компьютеру из-за трудностей перемещения данных. Таким образом, преимущества переносимости, обычно присущие

контейнерам, здесь не действуют, тем не менее использование контейнеров позволяет сохранить преимущества изоляции и устойчивости платформы. Если особое внимание уделяется производительности, то аргументы `--net=host` и `--privileged` обеспечат эффективность работы контейнера на уровне управляющей виртуальной машины или хост-компьютера, но при этом не следует забывать о проблемах безопасности. Даже если вы не работаете с собственными базами данных, а пользуетесь управляющим сервисом, таким как Amazon RDS, описанные выше проблемы в основном сохраняются.

В приложениях мелкого масштаба, где контейнеры сопровождаются файлами конфигурации и работают со средними объемами данных, может проявиться проблема с ограничением использования томов, так как тома фактически привязывают вас к конкретному хост-компьютеру, серьезно затрудняя масштабирование и перемещение контейнеров. В этом случае можно рассмотреть вариант перемещения данных в отдельные хранилища типа «ключ-значение» или в СУБД, которые также можно запустить в контейнере. Заслуживает внимания и другой подход, использующий Flocker (https://github.com/ClusterHQ/flocker) для управления томами данных. Flocker пользуется возможностями файловой системы ZFS для поддержки перемещения данных вместе с контейнерами. При попытках применения технологии микросервисов вы обнаружите, что дело существенно упрощается, если обеспечить поддержку контейнеров без сохранения состояния во всех возможных случаях.

Совместное использование закрытых данных

Вероятнее всего, вам придется работать с некоторыми закрытыми важными данными, такими как пароли и API-ключи, для которых необходимо обеспечить безопасное совместное использование всеми работающими контейнерами. В следующих разделах описываются различные методики обеспечения безопасности, а также их достоинства и недостатки[1].

Сохранение закрытых данных в образе

Никогда не делайте этого. Это плохая идея.™

Возможно, это самое простое решение, но при этом секретные данные становятся доступными всем, кому разрешен доступ к образу. Такие данные нельзя удалить, потому что они сохранятся на предыдущих уровнях. Даже если вы не пользуетесь частным закрытым реестром или вообще не используете реестр, существует большая вероятность случайного предоставления совместного свободного доступа к образу, после чего любой пользователь, получивший доступ к этому образу, без труда узнает секретные данные. Кроме того, такое решение привязывает образ к специфической процедуре развертывания.

---

[1] Если для развертывания контейнеров вы используете специализированное ПО для управления конфигурацией, например Ansible, то, скорее всего, в комплект включены средства решения этой проблемы, или такие средства можно приобрести дополнительно.

Можно было бы хранить секретные данные в образах в зашифрованном виде, но при этом остается необходимость передачи ключа шифрования каким-либо способом, то есть злоумышленники получают дополнительный шанс для перехвата ключа и взлома данных.

Лучше отказаться от этой идеи. Я включил этот раздел в книгу только для того, чтобы объяснить, насколько неудачен выбор этого способа хранения закрытых данных.

Передача закрытых данных в переменных среды

Использование переменных среды для передачи закрытых данных – вполне очевидное решение, которое значительно лучше варианта с хранением закрытых данных в образе. Его реализация проста: закрытые данные передаются как аргументы в команду `docker run`. Например:

```
$ docker run -d -e API_TOKEN=my_secret_token myimage
```

Многие приложения и файлы конфигурации напрямую поддерживают использование переменных среды. Для остальных потребуется написание скрипта, подобного показанному в разделе «Использование прокси-сервера» главы 9.

Такой подход рекомендуется в соответствии с широко известной и распространенной методикой The Twelve-Factor App (http://12factor.net/) для создания приложений типа *«ПО-как-сервис»* (*Software-as-a-Service – SaaS*)[1]. Настоятельно рекомендую изучить этот документ и следовать на практике большинству советов, но при этом не забывать о присущих способу хранения закрытых данных в переменных среды серьезных недостатках:

- переменные среды доступны всем дочерним процессам, команде `docker inspect` и всем контейнерам, с которыми установлено соединение, хотя для этого нет никаких разумных оснований;
- набор переменных среды часто сохраняется для контроля и отладки. Это создает большую опасность появления закрытых данных в журналах событий и отладочных протоколах;
- данные в переменных среды невозможно удалить. В идеальном случае желательно перезаписать или удалить закрытые данные после использования, но в контейнерах Docker это невозможно.

По указанным выше причинам эту методику нельзя рекомендовать для реального применения.

Передача закрытых данных в томах

Немного улучшенное решение (но все еще далекое от совершенства) – применение томов для совместного использования закрытых данных. Например:

```
$ docker run -d -v $(pwd):/secret-file:/secret-file:ro myimage
```

---

[1] Здесь следует отметить, что методика Twelve-Factor появилась раньше контейнеров Docker, поэтому некоторые ее положения необходимо привести в соответствие с технологией контейнеризации.

Если вы не включаете полностью все закрытые данные в конфигурационные файлы, то, вероятно, потребуется специальный скрипт для обработки данных, передаваемых этим способом. Если вы уверены в своих действиях, то можно создать временный файл с закрытыми данными, а после считывания всех требуемых данных удалить его (при этом будьте внимательны – не удалите случайно оригинал).

Для файлов конфигурации, использующих переменные среды, также можно написать скрипт, который присваивает значения переменным среды. Выполнение этого скрипта можно инициализировать перед запуском соответствующего приложения, например:

```
$ cat /secret/env.sh
export DB_PASSWORD=s3cr3t
$ source /secret/env.sh >> run_my_app.sh
```

Такой способ дает важное преимущество: передаваемые переменные среды недоступны команде `docker inspect` и контейнерам, с которыми установлены соединения.

Главный недостаток этой методики – требуется хранение закрытых данных в файлах, к содержимому которых можно легко получить доступ в системе управления версиями. Здесь также можно предложить более тщательно проработанное решение, которое обычно требует написания скриптов.

Использование хранилища типа «ключ-значение»

С некоторой долей уверенности можно утверждать, что наилучшим решением для хранения закрытых данных и извлечения их из контейнера во время выполнения является использование хранилища типа «ключ-значение». Такое решение обеспечивает уровень управления закрытыми данными, недоступный в описанных выше методиках, но при этом требуются дополнительные трудозатраты для настройки и обеспечения безопасности хранилища.

В этой области можно предложить следующие конкретные решения:

- *KeyWhiz* (https://square.github.io/keywhiz/) – хранит зашифрованные закрытые данные в памяти и обеспечивает доступ к ним через прикладной программный интерфейс REST и через интерфейс командной строки. Решение разработано компанией Square (сфера обработки платежей) и активно ею используется;
- *Vault* (https://hashicorp.com/blog/vault.html) – предоставляет возможность хранения зашифрованных закрытых данных в разнообразных внутренних (backend) хранилищах, включая обычные файлы и Consul. Также предлагает прикладной программный интерфейс и интерфейс командной строки. Обладает некоторыми функциональными возможностями, в текущий момент отсутствующими в KeyWhiz, но считается менее проработанным вариантом. Решение разработано компанией Hashicorp, также осуществляющей поддержку инструмента обнаружения сервисов Consul и инструмента создания конфигурации инфраструктуры Terraform;

- *Crypt* (https://xordataexchange.github.io/crypt/) – хранит зашифрованные значения в БД типа «ключ-значение» etcd или Consul. Главное преимущество заключается в том, что эта методика предоставляет степень управления закрытыми данными, недоступную ранее. Позволяет с легкостью изменять и удалять данные, применяет практику предоставления закрытых данных «во временное пользование» на ограниченный срок, а также средства полной блокировки доступа к закрытым данным в случаях возникновения угрозы их безопасности.

Тем не менее и здесь остается нерешенной проблема: каким образом контейнер будет выполнять процедуру аутентификации для доступа к конкретному хранилищу? Как правило, здесь сохраняется необходимость передачи в контейнер либо закрытого ключа с использованием тома, либо маркера доступа через переменную среды. Ранее перечисленные недостатки использования переменной среды могут быть частично устранены при создании *одноразового маркера доступа* (*токена*), немедленно удаляемого после применения. Другое решение, в текущий момент находящееся в разработке, заключается в использовании динамически подключаемого тома для хранилища, которое монтирует закрытые данные из исходного хранилища как файл внутри контейнера. Такая методика, применяемая в отношении к хранилищу KeyWhiz, более подробно описана в соответствующем документе на GitHub (https://github.com/calavera/docker-volume-keywhiz).

Этот тип решения будет реализован в будущем. Предоставляемый им уровень управления важными данными достаточно сложен в реализации, но сложность должна снижаться по мере усовершенствования инструментальных средств. В любом случае можно подождать с принятием окончательного решения и понаблюдать за развитием этой отрасли контейнеризации. А пока применяйте тома для совместного использования своих закрытых данных, но не обрабатывайте закрытых данных в системах управления конфигурацией.

Сетевая среда

Сетевая среда во всех подробностях обсуждается в главе 11. Но следует отметить, что если вы используете сетевую среду Docker как основу для эксплуатации, то получаете значительное снижение производительности – объединение интерфейсов в сетевой мост Docker и использование veth[1] приводит к выполнению большинства операций маршрутизации в пространстве пользователя, что существенно медленнее, чем аппаратная маршрутизация или маршрутизация средствами ядра.

Реестр для промышленной эксплуатации

При работе с identidock для загрузки образов мы использовали только реестр Docker Hub. Большинство эксплуатационных структур включает реестр (или не-

[1] Virtual Ethernet или veth – виртуальное сетевое устройство с собственным MAC-адресом, специально предназначенное для использования в виртуальных машинах.

сколько реестров) для предоставления быстрого доступа к образам, а также чтобы устранить зависимость от сторонних провайдеров для особо важных инфраструктур (в некоторых организациях беспокойство вызывает хранение их кода сторонним провайдером, вне зависимости от того, хранится код в закрытом репозитории или нет). Более подробно о настройке реестра см. раздел «Организация собственного реестра» главы 7.

Важное значение имеют поддержка в реестре образов в актуальном состоянии и возможность внесения изменений – вряд ли вас устроят хосты, загружающие старые и потенциально уязвимые образы. Поэтому лучше регулярно проводить контрольные проверки реестров по методике, описанной в разделе «Проведение контрольных проверок» главы 13. Но при этом следует помнить, что каждый Docker-хост поддерживает собственный кэш образов, для которого также необходимы контрольные проверки.

Создание зеркал и подобные варианты использования для формирования масштабируемых топологий и для обеспечения мгновенного доступа к образам в настоящее время разрабатываются в рамках проекта Docker distribution project (https://github.com/docker/distribution/).

Непрерывное развертывание/доставка

Непрерывная доставка (Continuous Delivery) является расширением методики непрерывной интеграции в режиме эксплуатации, в соответствии с которой инженеры по эксплуатации должны иметь возможность внесения изменений в процессе разработки, с обязательными процедурами тестирования и дальнейшим обеспечением развертывания измененных версий «одним нажатием кнопки». Методика непрерывного развертывания (Continuous Deployment) идет дальше, автоматически внедряя изменения, прошедшие тестирование перед развертыванием.

В главе 8 мы наблюдали процедуру установки системы непрерывной интеграции, использующую сервер Jenkins. Эту процедуру можно развить до стадии непрерывного развертывания, выгружая образы в эксплуатационный реестр и переключая работающие контейнеры на новые образы. Переключение образов без остановки требует создания новых контейнеров и изменения маршрута трафика перед остановкой старых контейнеров. Это можно сделать несколькими способами с соблюдением мер безопасности, например по методикам blue/green развертывания и постепенного развертывания, описанным в разделе «Тестирование в процессе эксплуатации» главы 8. Реализация этих методик часто выполняется собственными инструментальными средствами, несмотря на то что предлагаются готовые встроенные решения, такие как Kubernetes, и я хотел бы видеть на рынке больше специализированных инструментов для этой области контейнеризации.

Резюме

В этой главе был предложен обширный материал, и мы рассмотрели множество разнообразных аспектов процесса развертывания контейнеров для промышленной эксплуатации, даже в таких простых случаях, как identidock.

Несмотря на то что контейнеризация является молодой отраслью, в ней уже сформировались возможности промышленного уровня для хостинга контейнеров. Выбор наилучшего варианта зависит от размера и сложности конкретной системы, а также от того, сколько усилий и денежных средств вы готовы потратить на развертывание и сопровождение. Малые комплекты развертывания могут без труда управляться механизмом Docker Engine внутри виртуальной машины в облачной среде, но при больших комплектах развертывания трудозатраты на сопровождение резко увеличиваются. Несколько улучшить положение можно с помощью таких систем, как Kubernetes и Mesos, которые будут подробно рассматриваться в главе 12, или используя специализированный сервис хостинга, например Giant Swarm, Triton или ECS.

В этой главе рассматривались проблемы, часто возникающие при эксплуатации: от сравнительно простых вопросов, связанных с запуском контейнеров, до более сложных задач, таких как передача закрытых данных, обработка томов данных и непрерывное развертывание. Некоторые из этих задач требуют новых подходов к организации систем контейнеризации, особенно если системы сформированы из динамических микросервисов. В настоящее время разрабатываются новые шаблоны и практические методики для решения перечисленных задач, и это непременно приведет к созданию новых инструментальных средств и программных сред. Контейнеры уже сейчас готовы к надежной промышленной эксплуатации, но их будущее еще более оптимистично.

Глава 10

Ведение журналов событий и контроль

Обеспечение эффективного контроля и фиксации событий, происходящих в работающих контейнерах, является чрезвычайно важной задачей, если необходима поддержка сложной системы в рабочем состоянии, а также обеспечение правильных процедур отладки. При использовании архитектуры микросервисов ведение журналов событий и обеспечение контроля приобретают особую важность в связи с увеличением количества рабочих компьютеров. Из-за недолговечности жизненного цикла контейнеров любой контейнер может прекратить свое существование даже в процессе отладки при возникновении какой-либо проблемы, и поэтому централизованное ведение журналов событий является незаменимым средством.

В последнее время количество доступных решений для ведения журналов и контроля увеличивалось с впечатляющей скоростью. Существующие поставщики инструментов ведения журналов и контроля начали предлагать специализированные решения для контейнеров и интегрированные решения. В этой главе сделана попытка общего обзора разнообразных возможностей и методик, при этом основное внимание сосредоточено на бесплатных вариантах и на ПО с открытым исходным кодом. Мы подробно рассмотрим, как снабдить приложение identidock средствами контроля и ведения журналов событий, которые можно без труда масштабировать для более крупных приложений.

Исходные коды для этой главы доступны на странице книги в репозитории GitHub. Как и в главе 9, в примерах используются образы из реестра Docker Hub, но вы можете заменить их на собственный контейнер identidock, если хотите.
Для получения начальной версии кода для данной главы выполните команду с использованием тега v0:

```
$ git clone -b v0 https://github.com/using-docker/logging/
...
```

Следующие теги соответствуют версиям кода, получаемым в процессе дальнейшей разработки учебного примера в данной главе.
Кроме того, можно скачать код, соответствующий любому тегу, со страницы Releases проекта в репозитории GitHub (https://github.com/using-docker/logging/releases).

Ведение журналов событий

Сначала рассмотрим, как работает по умолчанию подсистема ведения журналов событий в Docker, затем добавим полноценное решение ведения журналов в приложение identidock, после чего перейдем к изучению альтернативных решений и способов устранения проблем при эксплуатации.

Принятая по умолчанию подсистема ведения журналов событий в Docker

Начнем с описания самого простого решения, которое включено в базовый комплект Docker. Если не заданы соответствующие аргументы и не установлено специальное ПО для ведения журналов, то Docker будет фиксировать все события, отправляя сообщения о них в стандартный поток вывода `STDOUT` и в стандартный поток ошибок `STDERR`. В дальнейшем эти записи можно извлечь командой `docker logs`, например:

```
$ docker run --name logtest debian sh -c 'echo "stdout"; echo "stderr" >&2'
stderr
stdout
$ docker logs logtest
stderr
stdout
```

Также можно получить отметки времени записей, используя для этого аргумент `-t`:

```
$ docker logs -t logtest
2015-04-27T10:30:54.002057314Z stderr
2015-04-27T10:30:54.005335068Z stdout
```

Кроме того, можно считывать журнальные записи о событиях из работающего контейнера с помощью аргумента `-f`:

```
$ docker run -d --name streamtest debian \
            sh -c 'while true; do echo "tick"; sleep 1; done;'
13aa6ee6406a998350781f994b23ce69ed6c38daa69c2c83263c863337a38ef9
$ docker logs -f streamtest
tick
tick
tick
tick
tick
tick
...
```

Это также можно сделать с применением интерфейса прикладного программирования Docker Remote API[1], который предоставляет возможности программного

[1] Для получения более подробной информации о Remote API и его применении см. официальную документацию https://docs.docker.com/engine/reference/api/docker_remote_api_v1.25/#docker-remote-api-v1-25.

управления и обработки журналов событий. При использовании Docker Machine описанная в предыдущем примере операция выполняется следующим образом:

```
$ curl -i --cacert ~/.docker/machine/certs/ca.pem \
          --cert ~/.docker/machine/certs/ca.pem \
          --key ~/.docker/machine/certs/key.pem \
          "https://$(docker-machine ip default):2376/containers/\
          $(docker ps -lq)/logs?stderr=1&stdout=1"
tick
tick
tick
...
```

Если вы работаете в MacOS, то обратите внимание на то, что `curl` будет работать немного по-другому, и вам придется создать единый сертификат, содержащий характеристики и *ca.pem*, и *key.pem*. Более подробно об этом можно прочесть в Open Solitude blog (https://opensolitude.com/2015/07/12/curl-docker-remote-api-os-x.html).

У подсистемы ведения журналов событий, принятой по умолчанию в Docker, есть некоторые недостатки. Она может работать только со стандартными потоками `STDOUT` и `STDERR`, что создает трудности, если ваше приложение фиксирует события только в файле. Кроме того, отсутствует ротация журналов, и если вы попытаетесь использовать приложения, подобные `yes` (которое просто периодически записывает слово «yes» в стандартный поток вывода `STDOUT`), для подтверждения работы контейнера, то очень скоро на диске не останется свободного места[1]. Например:

```
$ docker run -d debian yes
ba054389b7266da0aa4e42300d46e9ce529e05fc4146fea2dff92cf6027ed0c7
```

Существуют некоторые другие методы ведения журналов, которые можно применить с помощью аргумента `--log-driver` в команде `docker run`. Механизм ведения журналов, принятый по умолчанию, заменяется на переданный через аргумент `--log-driver` при инициализации демона Docker. Возможные варианты значений, определяющих механизмы ведения журналов:

- `json-file` – механизм, принятый по умолчанию, который описан выше;
- `syslog` – драйвер syslog, который будет рассматриваться ниже;
- `journald` – драйвер для ведения журналов под управлением systemd;
- `gelf` – драйвер Graylog Extend Log Format (GELF);
- `fluentd` – перенаправляет журнальные записи в fluentd (http://www.fluentd.org/);
- `none` – отключает ведение журналов событий.

Отключение ведения журналов событий может оказаться полезным в ситуациях, подобных описанному выше примеру с использованием утилиты `yes`.

Объединение журналов

Вне зависимости от того, какой драйвер для ведения журналов вы используете, будет доступно лишь частичное решение, особенно в системах со множеством хос-

[1] Да, я был настолько неосмотрителен, что практически проверил это на собственном опыте.

тов. Необходимо объединить все журналы (в том числе и с разных хостов) в единой точке, чтобы появилась возможность обрабатывать их аналитическими и контролирующими инструментальными средствами.

Для решения этой задачи предлагаются две основные методики:

1) запуск внутри всех контейнеров второго процесса, который работает как посредник, пересылающий все журнальные записи в объединяющий сервис;
2) сбор всех журнальных записей на конкретном хосте или в отдельном независимом контейнере и перенаправление их в объединяющий сервис.

Первая методика вполне работоспособна и даже иногда применяется, но она приводит к росту размеров образов и к не вполне оправданному увеличению количества активных процессов, поэтому мы будем рассматривать только вторую методику.

Доступ с хоста к журналам контейнера можно организовать несколькими способами:

1) использование интерфейса прикладного программирования Docker API. Несомненным преимуществом является официальная поддержка API, а накладные расходы, связанные с применением HTTP-соединения, невелики. В следующем разделе мы рассмотрим пример использования Logspout для решения этой задачи;
2) при использовании драйвера syslog можно воспользоваться его функциональностью для автоматического перенаправления журнальных записей, как показано в разделе «Перенаправление журнальных записей с помощью rsyslog» текущей главы;
3) можно организовать прямой доступ к файлам журналов из каталога Docker. Этот способ описан в разделе «Извлечение журнальных записей из файла» текущей главы.

Если для ведения журналов событий приложение использует только записи в файл без возможности вывода в стандартные потоки `STDOUT` и `STDERR`, то обратитесь к примечанию «Работа с приложениями, ведущими журналы в файле», чтобы получить несколько советов по обходу этого ограничения.

Ведение журналов с использованием ELK

Чтобы добавить механизм ведения журналов событий в наше приложение identidock, воспользуемся так называемым стеком ELK. ELK – это Elasticsearch, Logstash и Kibana:

- *Elasticsearch* (https://github.com/elastic/elasticsearch) – механизм текстового поиска, работающий почти в реальном времени. Создан как легко масштабирующийся при наличии множества сетевых узлов для обеспечения обработки больших объемов данных, поэтому наилучшим образом подходит для поиска в многочисленных журнальных записях;
- *Logstash* (https://github.com/elastic/logstash) – инструмент для чтения необработанных журнальных записей с последующим их синтаксическим разбором и фильтрацией перед отправкой в другой сервис, например для индек-

сирования или хранения (в нашем случае Logstash будет перенаправлять обработанные журнальные записи в Elasticsearch). Поддерживает широкий спектр типов входных и выходных данных, а также существующие синтаксические анализаторы (парсеры) для обработки журналов разнообразных приложений;

- *Kibana* (https://github.com/elastic/kibana) – основанный на JavaScript графический интерфейс для Elasticsearch. Может использоваться для формирования и запуска запросов Elasticsearch и для визуального представления результатов поиска с помощью разнообразных графиков и схем. Есть возможность настройки панелей управления для представления оперативных обзоров текущего состояния системы.

Этот стек мы будем использовать локально, на основе файла *prod.yml* из предыдущей главы[1]. В идеальном случае следовало бы переместить ELK-контейнеры на отдельный хост для обеспечения явной разделенности компонентов системы. Вариант с полным разделением будет рассмотрен в главе 11, но сейчас для упрощения мы оставим все компоненты на одном хосте.

В первую очередь необходимо определить, как отправлять журналы Docker в Logstash. Для этого воспользуемся Logspout (https://github.com/gliderlabs/logspout), специально предназначенным для Docker инструментом, который использует Docker API для перенаправления журнальных записей из работающих контейнеров в заданный конечный пункт (это в некоторой степени похоже на rsyslog для Docker). Поскольку в качестве конечного пункта мы намерены использовать Logstash, нужо также установить адаптер logspout-logstash (https://github.com/looplab/logspout-logstash), который форматирует журнальные записи Docker таким образом, чтобы их проще было читать в Logstash. Logspout тщательно проработан, поэтому имеет минимальный размер и обладает максимально возможной эффективностью, что позволяет ему работать на каждом Docker-хосте с весьма скромными требованиями к ресурсам. Причина такого минимализма объясняется тем, что Logspout написан на языке Go и основан на чрезвычайно компактном образе Alpine Linux. Поскольку по умолчанию контейнер для Logspout не содержит адаптера logstash, в последующих примерах я буду использовать собственный, специально подготовленный образ.

Общая схема нашей целевой системы должна выглядеть приблизительно так, как показано на рис. 10.1. Журнальные записи контейнеров в левой части схемы объединяются с помощью Logspout, затем выполняются синтаксический разбор и фильтрация в Logstash, потом передача обработанных записей в Elasticsearch. На последней стадии используется Kibana для визуального представления и исследования данных в контейнере Elasticsearch.

---

[1] При желании вы можете попробовать работу стека в облачной среде, но при этом может потребоваться обновление вашего сервера для обеспечения возможности запуска всех необходимых сервисов.

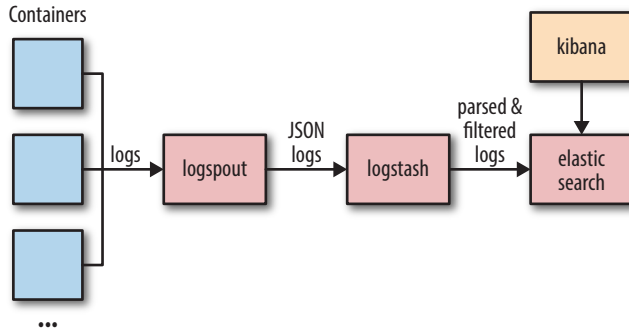

Рис. 10.1. Обработка журнальных записей контейнеров с помощью Logspout и ELK

Начнем с создания нового файла *prod-with-logging.yml* со следующим содержимым:

```
proxy:
  image: amouat/proxy:1.0
  links:
    - identidock
  ports:
    - "80:80"
  environment:
    - NGINX_HOST=45.55.251.164  ❶
    - NGINX_PROXY=http://identidock:9090
identidock:
  image: amouat/identidock:1.0  ❷
  links:
    - dnmonster
    - redis
  environment:
    ENV: PROD
dnmonster:
  image: amouat/dnmonster
redis:
  image: redis

logspout:
  image: amouat/logspout-logstash
  volumes:
    - /var/run/docker.sock:/tmp/docker.sock  ❸
  ports:
    - "8000:90"  ❹
```

❶ Замените этот IP-адрес на IP-адрес вашего хоста.
❷ Я использовал свой образ identidock из реестра Docker Hub, но вы можете заменить его своей версией.

❸ Монтирование Docker-сокета, для того чтобы Logspout мог установить соединение с Docker API.
❹ Открывается общий доступ к HTTP-интерфейсу Logspout для просмотра журналов. В реально эксплуатируемой системе не следует держать этот интерфейс открытым.

После перезапуска приложения должна появиться возможность установления соединения с Logspout через HTTP-интерфейс:

```
$ docker-compose -f prod-with-logging.yml up -d
...
$ curl localhost:8000/logs
```

Если открыть identidock в браузере, то в терминале сразу должны появиться журнальные записи:

```
logging_proxy_1|192.168.99.1 - - [24/Sep/2015:11:36:53 +0000] "GET / HTTP/1....
logging_identidock_1|[pid: 6|app: 0|req: 1/1] 172.17.0.14 () {40 vars in 660...
logging_identidock_1|Cache miss
     logging_proxy_1|192.168.99.1 - - [24/Sep/2015:11:36:53 +0000] "GET /mon...
logging_identidock_1|[pid: 6|app: 0|req: 2/2] 172.17.0.14 () {42 vars in 788...
logging_identidock_1|[pid: 6|app: 0|req: 3/3] 172.17.0.14 () {42 vars in 649...
```

Эта часть задачи выполнена, и, вероятно, в дальнейшем созданный интерфейс может оказаться полезным. Далее необходимо передать выводимую информацию для обработки, в нашем случае – в контейнер Logstash. Следует отметить, что в многохостовых системах потребуется запуск отдельного контейнера Logstash на каждом хосте с организацией маршрутизации до центрального экземпляра Logstash. Теперь надо подключить Logstash. Начнем с редактирования Compose-файла:

```
...
logspout:
  image: amouat/logspout-logstash
  volumes:
    - /var/run/docker.sock:/tmp/docker.sock
  ports:
    - "8000:80"
  links:
    - logstash  ❶
  command: logstash://logstash:5000  ❷
logstash:
  image: logstash
  volumes:
    - ./logstash.conf:/etc/logstash.conf  ❸
  environment:
    LOGSPOUT: ignore  ❹
  command: -f /etc/logstash.conf
```

❶ Добавлено соединение с контейнером Logstash.
❷ Здесь используется префикс "logstash", который сообщает Logspout о необходимости использования модуля Logstash для вывода данных.

❸ Определяется соответствующий файл конфигурации для Logstash.
❹ Logspout не будет собирать журнальные записи из тех контейнеров, в которых установлена переменная среды LOGSPOUT. В данном случае не нужен сбор журнальных записей из самого контейнера Logstash, так как при этом возникает опасность бесконечного зацикливания при получении некорректной записи, вызывающей ошибку в Logstash, которая фиксируется в журнале и снова возвращается в Logstash, возникает новая ошибка, и так до бесконечности...

Файл конфигурации с именем *logstash.conf* должен содержать следующие данные:

```
input {
  tcp {
    port => 5000
    codec => json  ❶
  }
  udp {
    port => 5000
    codec => json  ❶
  }
}

output {
  stdout { codec => rubydebug }  ❷
}
```

❶ Для обработки выходных данных Logspout необходимо использовать кодек json.
❷ Для проведения тестирования вывод журнальных записей будет выполняться в стандартный поток STDOUT.

Снова запустим приложение и понаблюдаем за результатом:

```
$ docker-compose -f prod-with-logging.yml up -d
...
$ curl -s localhost > /dev/null
$ docker-compose -f prod-with-logging.yml logs logstash
...
logstash_1 | {
logstash_1 |             "message" => "2015/09/24 12:50:25 logstash: write u...
logstash_1 |         "docker.name" => "/logging_logspout_1",
logstash_1 |           "docker.id" => "d8f69d05123c43c9da7470951547b23ab32d4...
logstash_1 |        "docker.image" => "amouat/logspout-logstash",
logstash_1 |     "docker.hostname" => "d8f69d05123c",
logstash_1 |            "@version" => "1",
logstash_1 |          "@timestamp" => "2015-09-24T12:50:25.708Z",
logstash_1 |               "host" => "172.17.0.11"
logstash_1 | }
...
```

Вы должны увидеть несколько записей в формате Ruby, похожие на приведенные в примере выше. Отметим, что в вывод включены такие поля, как имя кон-

тейнера и его идентификатор, которые добавлены программой Logspout. Logstash принял данные в формате JSON, обработал их и вывел в отладочном формате Ruby. Но с помощью Logstash можно сделать гораздо больше: можно фильтровать и редактировать журнальные записи, если это необходимо. Например, может потребоваться удаление данных, идентифицирующих личность, или секретной информации перед передачей журнальных записей в другой сервис для дальнейшей обработки или хранения. В нашем случае неплохо было бы разделить сообщение из журнала nginx на более удобные фрагменты. Это можно сделать, добавив раздел `filter` в файл конфигурации Logstash:

```
input {
  tcp {
    port => 5000
    codec => json
  }
  udp {
    port => 5000
    codec => json
  }
}

filter {
  if [docker.image] =~ /^amouat\/proxy.*/ {
    mutate { replace => { type => "nginx" } }
    grok {
      match => { "message" => "%{COMBINEDAPACHELOG}" }
    }
  }
}

output {
  stdout { codec => rubydebug }  (2)
}
```

Этот фильтр проверяет сообщения, выбирая среди них те, которые пришли от образа с именем `amouat/proxy`. Для выбранных сообщений выполняется синтаксический разбор с использованием существующего Logstash-фильтра `COMBINEDAPACHELOG`, который добавляет к выводимой информации несколько дополнительных полей. Если после добавления указанного фильтра перезапустить приложение, то вывод журнальных записей должен выглядеть приблизительно так:

```
logstash_1 | {
logstash_1 |             "message" => "87.246.78.46 - - [24/Sep/2015:13:02:...
logstash_1 |         "docker.name" => "/logging_proxy_1",
logstash_1 |           "docker.id" => "5bffa4f4a9106e7381b22673569094be20e8...
logstash_1 |        "docker.image" => "amouat/proxy:1.0",
logstash_1 |     "docker.hostname" => "5bffa4f4a910",
logstash_1 |            "@version" => "1",
```

```
logstash_1 |          "@timestamp" => "2015-09-24T13:02:59.751Z",
logstash_1 |               "host" => "172.17.0.23",
logstash_1 |               "type" => "nginx",
logstash_1 |           "clientip" => "87.246.78.46",
logstash_1 |              "ident" => "-",
logstash_1 |               "auth" => "-",
logstash_1 |          "timestamp" => "24/Sep/2015:13:02:59 +0000",
logstash_1 |               "verb" => "GET",
logstash_1 |            "request" => "/",
logstash_1 |        "httpversion" => "1.1",
logstash_1 |           "response" => "200",
logstash_1 |              "bytes" => "266",
logstash_1 |           "referrer" => "\"-\"",
logstash_1 |              "agent" => "\"curl/7.37.1\""
logstash_1 | }
```

Отметим, что применение фильтра позволило получить много дополнительной информации, такой как, например, код ответа, тип запроса, URL и т. п. Используя подобную методику, можно установить фильтры для обработки журнальных записей из разнообразных образов.

Следующий шаг – установление соединения контейнера Logstash с контейнером Elasticsearch. Одновременно с этой операцией мы также добавим контейнер Kibana, чтобы обеспечить удобный пользовательский интерфейс для Elasticsearch.

Еще раз отредактируем Compose-файл:

```
...
logspout:
  image: amouat/logspout-logstash
  volumes:
    - /var/run/docker.sock:/tmp/docker.sock
  ports:
    - "8000:80"
  links:
    - logstash
  command: logstash://logstash:5000

logstash:
  image: logstash:1.5
  volumes:
    - ./logstash.conf:/etc/logstash.conf
  environment:
    LOGSPOUT: ignore
  links:
    - elasticsearch  ❶
  command: -f /etc/logstash.conf

elasticsearch:  ❷
  image: elasticsearch:1.7
```

```
  environment:
    LOGSPOUT: ignore

kibana:  ❸
  image: kibana:4.0
  environment:
    LOGSPOUT: ignore
    ELASTICSEARCH_URL: http://elasticsearch:9200
  links:
    - elasticsearch
  ports:
    - "5601:5601"
```

❶ Добавление соединения с контейнером Elasticsearch.
❷ Создание контейнера Elasticsearch на основе официального образа.
❸ Создание контейнера Kibana 4. Отметим, что здесь добавляется соединение с контейнером Elasticsearch и объявляется открытым порт 5601 для интерфейса.

Также необходимо изменить раздел `output` в файле *logstash.conf*, чтобы направить вывод данных в контейнер Elasticsearch:

```
...
output {
  elasticsearch { host => "elasticsearch" }  ❶
  stdout { codec => rubydebug }  ❷
}
```

❶ Вывод данных в формате, который может прочитать Elasticsearch, направленный на удаленный хост с именем elasticsearch.
❷ Эту строку можно удалить, так как она нужна только для отладки и в текущем состоянии приводит к дублированию выводимых журнальных записей.

Еще раз перезапустим приложение и получим следующий результат:

```
$ docker-compose -f prod-with-logging.yml up -d
Recreating logging_dnmonster_1...
Recreating logging_redis_1...
Recreating logging_elasticsearch_1...
Recreating logging_kibana_1...
Recreating logging_identidock_1...
Recreating logging_proxy_1...
Recreating logging_logstash_1...
Recreating logging_logspout_1...
```

Затем в браузере нужно перейти по адресу `localhost` и выполнить несколько действий в приложении identidock, чтобы в журнале появились записи для анализа. После этого в браузере откройте URL `localhost:5601` для запуска приложения Kibana. Результат должен быть похож на рис. 10.2.

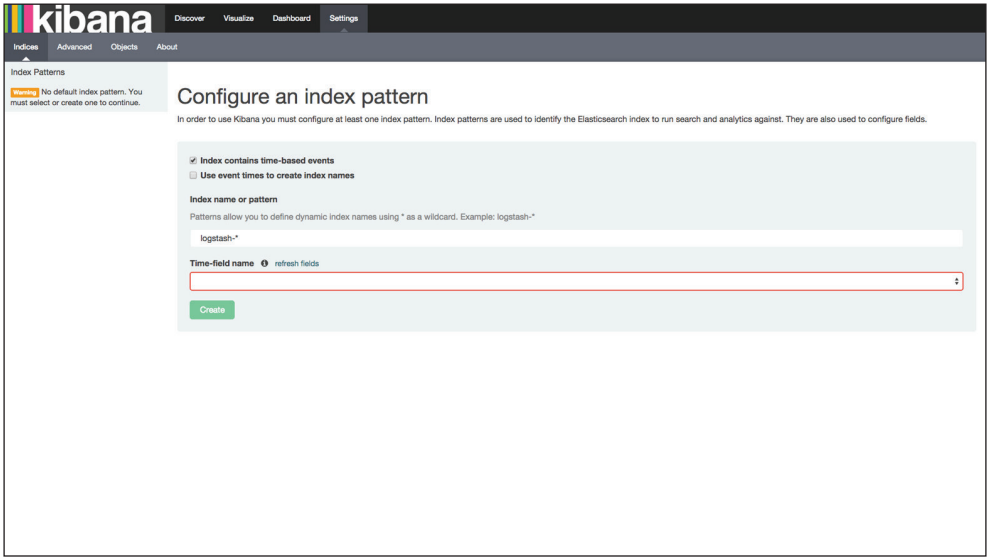

Рис. 10.2. Страница конфигурации Kibana

В качестве имени поля временной метки введите `@timestamp` и щелкните по кнопке **Create** (Создать). Затем будет выведена страница, содержащая все поля, найденные сервисом Elasticsearch, в том числе поля журналов nginx и Docker.

Затем щелкните по кнопке **Discover** (Анализ), чтобы получить страницу с гистограммой, отображающей объемы журнальных записей, под которой размещен список последних полученных записей, как показано на рис. 10.3.

Временной интервал получения данных можно с легкостью изменить, щелкнув по значку часов в верхнем правом углу. Можно выполнять фильтрацию журнальных записей, определив поиск заданных терминов в конкретных полях. Например, можно определить поиск фразы «Cache miss» в поле `message` и получить гистограмму количества промахов кэша за некоторый интервал времени. Более сложные графики и визуальные представления можно сгенерировать, если воспользоваться вкладкой **Visualize** (Визуализация), где предлагаются линейные и секторные диаграммы, а также таблицы данных и настраиваемые метрики.

Kibana – версия 3 и версия 4

Если используется версия Kibana, более ранняя, чем 4.0, то необходимо убедиться в том, что браузер обеспечивает возможность доступа к контейнеру Elasticsearch при перенаправлении порта на хост. Причина этого требования заключается в том, что Kibana является приложением, основанным на JavaScript, которое работает на стороне клиента. Начиная с версии 4, обеспечивается механизм прокси для соединений через Kibana, и вышеуказанное требование отменяется.

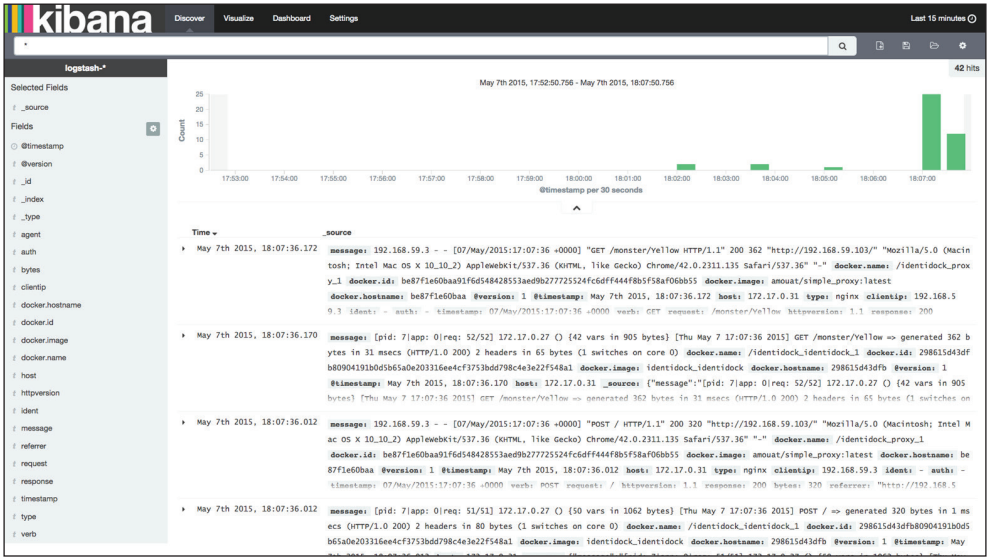

Рис. 10.3. Страница анализа Kibana

В этой главе не ставится задача полного описания всестороннего анализа журнальных записей, поэтому здесь мы завершим рассмотрение решения с использованием Kibana. Достаточно добавить, что Kibana и аналогичные решения предлагают мощные методики визуализации для исследования приложений и данных.

Хранение и ротация журналов

После выбора драйвера журналов и средств анализа журнальных записей необходимо решить, каким образом организовать хранение журналов, и определить срок их хранения. Если эти вопросы оставить без внимания, то, вероятнее всего, контейнеры будут использовать механизм ведения журналов по умолчанию и постепенно «съедят» все пространство на жестком диске, что приведет к критическому сбою хоста.

Утилиту logrotate ОС Linux можно использовать для управления размерами файлов журналов. Обычно используется несколько так называемых поколений (версий) файлов журналов, и через регулярные интервалы времени файлы журналов перемещаются по линии поколений. Например, в дополнение к текущей версии журнала могут существовать поколения «отец», «дед» и «прадед» как отдельные версии журналов. Версии «дед» и «прадед» подвергаются процедуре сжатия для более компактного хранения. Каждый день текущая версия журнала перемещается в поколение «отец», в свою очередь, бывший «отец» сжимается и перемещается в поколение «дед», бывший «дед» становится «прадедом», а бывший «прадед» удаляется.

Для работы по описанной схеме можно использовать приведенную ниже конфигурацию logrotate, которая должна быть сохранена в новом файле в подкаталоге */etc/logrotate.d/* (например, */etc/logrotate.d/docker*) или добавлена в файл */etc/logrotate.conf*:

```
/var/lib/docker/containers/*/*.log {
  daily ❶
  rotate 3 ❷
  compress ❸
  delaycompress ❸
  missingok ❹
  copytruncate ❺
}
```

❶ Выполнять ротацию журналов каждый день.
❷ Хранить три поколения файлов журналов.
❸ Использовать функцию сжатия, но исключить одно (самое младшее) поколение.
❹ Не останавливать работу logrotate при возникновении ошибки, если файлы журналов не найдены.
❺ Вместо перемещения текущего файла журнала копировать его, затем выполнять операцию усечения (до нулевого размера). Это необходимо для предотвращения сбоев в работе Docker при внезапном исчезновении требуемого файла. Но при этом существует вероятность потери данных, если приложение вносит запись в журнал между операциями копирования и усечения.

Вероятно, вы заметите, что по умолчанию logrotate выполняется как задание системного сервиса `cron` один раз в день. Если необходимо более часто производить очистку файлов журналов, то нужно внести соответствующие изменения в файл crontab. Для организации более основательного и надежного хранилища журналов перенаправляйте записи в проверенную базу данных, такую как PostgreSQL. Можно без затруднений добавить эту возможность как второй поток вывода из Logstash или аналогичного инструментального средства. При сохранении журнальных записей не следует полностью полагаться на инструменты индексирования, подобные Elasticsearch, так как они не способны обеспечить устойчивость к сбоям, присущую «настоящим» СУБД типа PostgreSQL. Отметим возможность использования фильтров Logstash для исключения некоторых данных, например персональных данных, если это необходимо.

Работа с приложениями, ведущими журналы в файле

Если приложение фиксирует события только в файле, а не выводит сообщения в поток `STDOUT`/`STDERR`, то и здесь сохраняются некоторые возможности их обработки. Если вы уже используете интерфейс Docker API для ведения журналов (например, с помощью контейнера Logspout), то самым простым решением будет запуск процесса (обычно это `tail -F`), который просто выводит содержимое файла журнала в стандартный поток `STDOUT`. Более изящный способ сделать это, сохраняя основной принцип «один процесс – один контейнер», – создать второй контейнер, который монтирует файлы журналов с помощью аргумента `--volumes-from`. Например, при наличии контейнера tolog, который объявляет том в каталоге */var/log*, можно воспользоваться следующей командой:

```
$ docker run -d --name tolog-logger --volumes-from tolog \
            debian tail -F
/dev/log/*
```

Если такая методика не подходит, то можно смонтировать файлы журналов в известный каталог на хосте и запустить инструмент сбора журнальных записей, например fluentd (http://www.fluentd.org/), для обработки этих файлов.

Ведение журналов Docker с использованием syslog

Если на Docker-хосте обеспечена поддержка сервиса syslog, то вы можете воспользоваться драйвером syslog, который будет пересылать журнальные записи контейнера в syslog на хосте. Возможно, этот подход будет более понятным на конкретном примере:

```
$ ID=$(docker run -d --log-driver=syslog debian \
      sh -c 'i=0; while true; do i=$((i+1)); echo "docker $i"; sleep 1; done;')
$ docker logs $ID
"logs" command is supported only for "json-file" logging driver (got: syslog)  ❶
(Команда logs поддерживается только для драйвера json-file (здесь: syslog))
$ tail /var/log/syslog  ❷
Sep 24 10:17:45 reginald docker/181b6d654000[3594]: docker 48
Sep 24 10:17:46 reginald docker/181b6d654000[3594]: docker 49
Sep 24 10:17:47 reginald docker/181b6d654000[3594]: docker 50
Sep 24 10:17:48 reginald docker/181b6d654000[3594]: docker 51
Sep 24 10:17:49 reginald docker/181b6d654000[3594]: docker 52
Sep 24 10:17:50 reginald docker/181b6d654000[3594]: docker 53
Sep 24 10:17:51 reginald docker/181b6d654000[3594]: docker 54
Sep 24 10:17:52 reginald docker/181b6d654000[3594]: docker 55
Sep 24 10:17:53 reginald docker/181b6d654000[3594]: docker 56
Sep 24 10:17:54 reginald docker/181b6d654000[3594]: docker 57
```

❶ Во время написания книги команда `docker logs` работала только для подсистемы ведения журналов, принятой по умолчанию.

❷ На моем Ubuntu-хосте журнальные записи пересылаются в */var/log/syslog*. В различных дистрибутивах Linux целевой файл может быть другим.

Вероятно, в файл журнала syslog вместе с сообщениями из контейнера также будут включены журнальные записи из разнообразных сервисов и других контейнеров. Поскольку сообщения, фиксируемые в журнале, содержат идентификатор контейнера (в краткой форме), можно воспользоваться утилитой `grep`, чтобы найти все сообщения, принадлежащие интересующему нас контейнеру:

```
$ grep ${ID:0:12} /var/log/syslog  ❶
Sep 24 10:16:58 reginald docker/181b6d654000[3594]: docker 1
Sep 24 10:16:59 reginald docker/181b6d654000[3594]: docker 2
Sep 24 10:17:00 reginald docker/181b6d654000[3594]: docker 3
Sep 24 10:17:01 reginald docker/181b6d654000[3594]: docker 4
Sep 24 10:17:02 reginald docker/181b6d654000[3594]: docker 5
Sep 24 10:17:03 reginald docker/181b6d654000[3594]: docker 6
Sep 24 10:17:04 reginald docker/181b6d654000[3594]: docker 7
...
```

❶ В журналах используется краткая форма идентификатора, поэтому при поиске следует ограничить длину идентификатора 12 символами.

Интерфейс прикладного программирования (API) обработки событий Docker

В дополнение к журналам контейнеров и журналам демона Docker существует еще один набор данных, который может потребовать контроля и обработки, – события Docker. События фиксируются для большинства этапов жизненного цикла контейнера Docker. Список возможных событий: `create`, `destroy`, `die`, `export`, `kill`, `pause`, `attach`, `restart`, `start`, `stop`, `unpause`. Смысл почти всех событий понятен по их именам, но следует отметить, что событие `die` возникает при завершении работы (выходе) контейнера, а событие `destroy` – при его удалении (то есть при выполнении команды `docker rm`). На рис. 10.4 жизненный цикл контейнера и соответствующие события показаны в виде схемы.

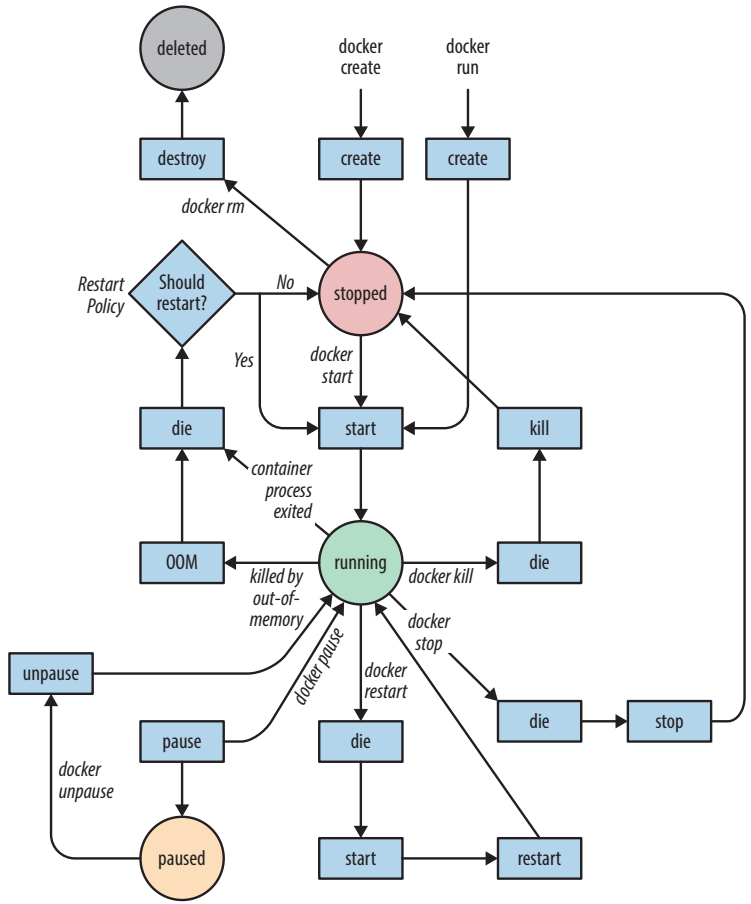

Рис. 10.4. Полный жизненный цикл контейнера Docker (из официальной документации Docker; автор схемы Мэт Гуд (Mat Good), компания Glider labs, изображение защищено лицензией CC-BY-SA **4.0** (https://creativecommons.org/licenses/by-sa/4.0/))

События `untag` и `delete` фиксируются для образов, событие `untag` возникает при удалении тега, то есть после успешного выполнения команды `docker rmi`. Событие `delete` возникает при удалении образа нижележащего уровня (это событие не всегда сопутствует выполнению команды `docker rmi`, так как у образа может быть несколько тегов). Отметки времени выводятся в формате, определенном в документе RFC 3339 (https://www.ietf.org/rfc/rfc3339.txt).
События можно просматривать с помощью команды `docker events`:

```
$ docker events
2015-09-24T15:23:28.000000000+01:00 44fe57bab...: (from debian) create
2015-09-24T15:23:28.000000000+01:00 44fe57bab...: (from debian) attach
2015-09-24T15:23:28.000000000+01:00 44fe57bab...: (from debian) start
2015-09-24T15:23:28.000000000+01:00 44fe57bab...: (from debian) die
```

Поскольку команда `docker events` выводит непрерывный поток данных о событиях, необходимо выполнить некоторые команды Docker с другого терминала, чтобы увидеть какие-либо результаты. Кроме того, команда `docker events` принимает аргументы для фильтрации результатов и управления интервалами времени при выводе результатов. События могут быть отфильтрованы по контейнерам, по образам и по типам событий. Отметки времени, используемые в аргументах, должны быть отформатированы в соответствии с требованиями документа RFC 3339 (пример форматирования: `"2006-01-02T15:04:05.000000000Z07:00"`) или заданы в секундах, отсчитываемых от начала Unix-эпохи (пример: `"1378216169"`).
Прикладной программный интерфейс обработки событий Docker может оказаться очень полезным, если необходима автоматическая реакция на события контейнеров. Например, утилита Logspout использует этот интерфейс для оповещений о запуске контейнеров и начале процесса фиксации записей в журнале. Контейнер nginx-proxy Джейсона Уайлдера, обсуждаемый в примечании «Усовершенствованная генерация файла конфигурации» в главе 9, использует прикладной программный интерфейс обработки событий для автоматической балансировки при увеличении нагрузки на контейнеры. Кроме того, может потребоваться фиксация в журналах всех данных, необходимых для анализа жизненного цикла работающих контейнеров.

Положение можно немного улучшить, если настроить syslog таким образом, чтобы он помещал сообщения от Docker в отдельный журнальный файл. Docker подтверждает возможность ведения журналов демона в syslog, поэтому можно без труда настроить syslog для записи сообщений от всех демонов в заданный файл, но фильтрацию сообщений только от Docker организовать немного сложнее[1]. Если вы используете syslog версии 7 или (что более вероятно) более поздней версии, то можете применить следующее правило:

```
:syslogtag,startswith, "docker/" /var/log/containers.log
```

По этому правилу все сообщения от контейнеров Docker будут помещаться в файл журнала */var/log/container.log*. Добавьте это правило в файл конфигурации

[1] Подобных сложностей не должно быть, но похоже, что эти проблемы syslog остались нерешенными еще с 1980-х гг.

syslog. В Ubuntu, например, можно создать новый файл */etc/rsyslog.d/30-docker.conf* и поместить правило в этот файл. Затем перезапустить syslog, после чего в новом файле журнала появятся записи:

```
$ sudo service rsyslog restart
rsyslog stop/waiting
rsyslog start/running, process 15863
$ docker run -d --log-driver=syslog debian \
  sh -c 'i=0; while true; do i=$((i+1)); echo "docker $i"; sleep 1; docne;'
$ cat /var/log/containers.log
Sep 24 10:30:46 reginald docker/1a1a57b885f3[3594]: docker 1
Sep 24 10:30:47 reginald docker/1a1a57b885f3[3594]: docker 2
Sep 24 10:30:48 reginald docker/1a1a57b885f3[3594]: docker 3
Sep 24 10:30:49 reginald docker/1a1a57b885f3[3594]: docker 4
Sep 24 10:30:50 reginald docker/1a1a57b885f3[3594]: docker 5
Sep 24 10:30:51 reginald docker/1a1a57b885f3[3594]: docker 6
Sep 24 10:30:52 reginald docker/1a1a57b885f3[3594]: docker 7
```

Но при этом сообщения будут фиксироваться и в других файлах (например, в */var/log/syslog*). Чтобы отменить это действие, добавьте строку с командой `&stop` непосредственно после строки с новым правилом, например:

```
:syslogtag,startswith, "docker/" /var/log/containers.log
&stop
```

syslog и виртуальные машины Docker Machine

Во время написания книги syslog по умолчанию не запускался в виртуальных машинах boot2docker, предоставляемых механизмом Docker Machine. Для тестирования можно организовать ведение журналов внутри виртуальной машины, запустив демон syslogd, например:

```
$ docker-machine ssh default
...
docker@default:~$ syslogd
```

Это изменение можно сделать постоянным, вызывая syslogd из файла */var/lib/boot2docker/bootsync.sh* внутри виртуальной машины boot2docker. Этот скрипт виртуальная машина выполнит перед запуском Docker:

```
$ docker-machine ssh default
...
docker@default:~$ cat /var/lib/boot2docker/bootsync.sh
#!/bin/sh
syslogd
```

Следует отметить, что виртуальная машина boot2docker по умолчанию использует реализацию syslog из busybox, которая не столь гибка, как rsyslogd.

Можно определить syslog как механизм журналирования по умолчанию для контейнеров Docker, добавив аргумент `--log-driver=syslog` в команду инициализации демона (обычно для этого вносятся изменения в файл конфигурации для

сервиса Docker, например указанный аргумент добавляется в строку `DOCKER_OPTS` в файле */etc/default/docker* в Ubuntu).

Перенаправление журнальных записей с помощью syslog

Можно также настроить rsyslog таким образом, чтобы он перенаправлял записи журналов на другой сервер, а не хранил их локально. Такой подход применяется для сбора журналов в централизованном сервисе, таком как Logstash, или на другом сервере syslog без дополнительных накладных расходов, связанных с использованием утилит типа Logspout.

Для замены Logspout на syslog в нашем примере identidock необходимо отредактировать файл конфигурации Logstash, чтобы создать возможность приема ожидаемых данных в syslog, перенаправить порт на хосте в Logstash для обмена данными с rsyslog и указать rsyslog на необходимость передачи журналов по сети вместо записи их в локальный файл.

Можно начать с изменения конфигурации Logstash. Отредактируйте файл конфигурации следующим образом:

```
input {
  syslog {
    type => syslog
      port => 5544
  }
}

filter {
  if [type] == "syslog" {
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd HH:mm:ss" ]
    }
  }
}

output {
  elasticsearch { host => "elasticsearch" }
  stdout { codec => rubydebug }
}
```

Теперь займемся конфигурацией rsyslog. Файл конфигурации почти тот же, что приводился выше, только вместо указания локального файла */var/log/containers.log* необходимо использовать синтаксическую конструкцию `@@localhost:5544`, например:

```
:syslogtag,startswith, "docker/" @@localhost:5544
&stop
```

Это сообщает rsyslog о необходимости передачи журналов с использованием протокола TCP в порт 5544 на хосте localhost. Для передачи по протоколу UDP используйте один символ `@`[1].

---

[1] Я полагаю, это очевидно, не так ли?

Заключительным этапом изменения конфигурации является редактирование Compose-файла. Но до этого лучше остановить все работающие экземпляры identidock:

```
$ docker-compose -f prod-with-logging.yml stop
...
```

Теперь можно без последствий удалить Logspout из Compose-файла и объявить открытым порт для rsyslog на хосте, чтобы обмениваться данными с Logstash:

```
...
logstash:
  image: logstash:1.5
  volumes:
    - ./logstash.conf:/etc/logstash.conf
  environment:
    LOGSPOUT: ignore
  links:
    - elasticsearch
    - "127.0.0.1:5544:5544"  ❶
  command: -f /etc/logstash.conf

elasticsearch:
  image: elasticsearch:1.7
  environment:
    LOGSPOUT: ignore

kibana:
  image: kibana:4.1
  environment:
    LOGSPOUT: ignore
    ELASTICSEARCH_URL: http://elasticsearch:9200
  links:
    - elasticsearch
  ports:
   - "5601:5601"
```

❶ Объявляется открытым порт 5544. Здесь выполняется привязка только к интерфейсу 127.0.0.1, так что хост может установить соединение с этим портом, но другим компьютерам в сети это запрещено.

После этого нужно перезапустить rsyslog, затем identidock. Теперь мы можем видеть, как журналы передаются в Logstash с помощью rsyslog, а не с использованием более медленного способа Logspout. Потребуется еще некоторая дополнительная работа по формированию конфигурационных фильтров, чтобы получить в Logstash всю необходимую информацию, которую ранее предоставлял Logspout, тем не менее использование rsyslog для перенаправления файлов журналов представляет собой весьма эффективное и надежное решение.

Обеспечение надежного ведения журналов

При проектировании инфраструктуры ведения журналов вам придется найти компромисс между эффективностью (производительностью) и абсолютной точностью и надежностью, вне зависимости от ваших намерений. Если журнальные записи нужны только для отладки и контроля, то, вероятнее всего, вы предпочтете наиболее простое решение. Но если для определенных сообщений в журналах требуются немедленное уведомление и реакция на них, или необходима полная проверка журналов на соответствие заданным стратегиям, то гораздо важнее рассмотреть все функциональные характеристики и обеспечить все возможные связи в инфраструктуре ведения журналов.

Вот некоторые ключевые пункты, требующие особого внимания:

- какой транспортный протокол используется для передачи журналов? UDP быстрее, но обладает меньшей надежностью, по сравнению с TCP (в свою очередь, протокол TCP не гарантирует абсолютной надежности);
- что происходит, если сеть неработоспособна? Отметим, что многие инструментальные средства, в том числе rsyslog, могут быть сконфигурированы для создания буферов, хранящих сообщения до тех пор, пока не станет доступным удаленный сервер;
- как организовано хранение и резервное копирование сообщений? СУБД предлагают более надежные и устойчивые к критическим сбоям возможности хранения данных, чем файловые системы.

С этими вопросами тесно связана проблема обеспечения безопасности журналов. В большинстве случаев журналы содержат секретную информацию, поэтому важно иметь средства управления доступа к ним. Необходима уверенность в том, что проходящие по открытой части Интернета журналы полностью зашифрованы, и доступ к хранящимся журналам может получить только строго определенный круг людей.

Извлечение журнальных записей из файла

Еще одним эффективным способом перенаправления журналов является организация доступа к исходным журнальным записям в файловой системе.

Если вы пользуетесь механизмом ведения журналов, принятым по умолчанию, то Docker в настоящее время хранит файлы сообщений от контейнеров по следующей схеме: */var/lib/docker/containers/<container_id>/<container_id>-json.log*.

Получение журнальных записей непосредственно из файлов является эффективным решением, но это решение основано на внутренних аспектах реализации Docker, а не на внешнем прикладном программном интерфейсе. Поэтому существует вероятность того, что решения, основанные на этой методике, потеряют работоспособность при изменениях во внутреннем механизме Docker.

Контроль и система оповещения

В системе, использующей микросервисы, существуют десятки, возможно, сотни или даже тысячи работающих контейнеров. В такой ситуации полезными будут все средства, позволяющие контролировать состояние работающих контейне-

ров и всей системы в целом. Правильная методика контроля должна обеспечивать оперативное отображение состояния системы и выдачу предварительных предупреждений о потенциальных опасностях, связанных с исчерпанием ресурсов (дискового пространства, процессоров, оперативной памяти). Также необходимы уведомления обо всех нежелательных событиях (например, начало обработки запроса задерживается на длительное время).

Контроль с помощью Docker Tools

В дистрибутивном комплекте Docker имеется простой инструмент командной строки `docker stats`, который возвращает в реальном времени характеристики использования ресурсов. Команда принимает имя одного или нескольких контейнеров и выводит разнообразную статистическую информацию по ним почти так же, как известная Unix-утилита `top`. Например:

```
$ docker stats logging_logspout_1
CONTAINER            CPU %   MEM USAGE/LIMIT     MEM %   NET I/O
logging_logspout_1   0.13%   1.696 MB/2.099 GB   0.08%   4.06 kB/9.479 kB
```

Статистика показывает степень использования процессоров и оперативной памяти, а также сетевые характеристики. Отметим, что если для контейнера не установлен лимит оперативной памяти, то предельным значением для него будет общий объем оперативной памяти на хосте, а не какой-либо ограниченный объем памяти, доступной для данного контейнера.

Получение статистических данных по всем работающим контейнерам
В большинстве случаев необходимо получать статистические данные по всем контейнерам, работающим на хосте (на мой взгляд, это должно быть определено по умолчанию). Такой результат можно получить, применив немного умения программировать на языке командной оболочки:

```
$ docker stats $(docker inspect -f {{.Name}} $(docker ps -q))
CONTAINER                  CPU %   MEM USAGE/LIMIT    ...
/logging_dnmonster_1       0.00%   57.51 MB/2.099 GB
/logging_elasticsearch_1   0.60%   337.8 MB/2.099 GB
/logging_identidock_1      0.01%   29.03 MB/2.099 GB
/logging_kibana_1          0.00%   61.61 MB/2.099 GB
/logging_logspout_1        0.14%   1.7 MB/2.099 GB
/logging_logstash_1        0.57%   263.8 MB/2.099 GB
/logging_proxy_1           0.00%   1.438 MB/2.099 GB
/logging_redis_1           0.14%   7.283 MB/2.099 GB
```

Команда `docker ps -q` выдает идентификаторы всех работающих контейнеров, которые используются как входные данные для команды `docker inspect -f {{.Name}}`, которая выполняет преобразование идентификаторов в имена, передаваемые далее в команду `docker stats`.

Получаемая информация полезна даже в таком виде, а кроме того, она дает понять, что с помощью Docker API можно получать статистические данные программно. Такая возможность действительно существует: можно обратиться к ко-

нечному пункту `/containers/<id>/stats` для получения потока разнообразных статистических данных по заданному контейнеру, более подробных, чем результаты, выводимые командой `docker stats`. В некоторой степени программный интерфейс менее гибок – можно обновлять поток всех выводимых данных каждую секунду или только однократно получить всю статистическую информацию, но при этом нет возможности управлять частотой вывода данных или фильтровать их. Поэтому для постоянного контроля интерфейс вывода статистики не вполне пригоден, так как связан с довольно большими издержками, но он остается полезным для оперативных запросов и исследований.

Большинство метрик, предъявляемых механизмом Docker, также доступно непосредственно из ядра Linux через функции CGroups и пространства имен, доступ к которым предоставляется с помощью различных библиотек и инструментальных средств, в том числе и посредством библиотеки runc из набора инструментов Docker (https://github.com/opencontainers/runc). Если нужен контроль какой-то особой метрики, то можно создать эффективное решение с помощью runc или напрямую работать с вызовами ядра. Для этого потребуется использование языка программирования, позволяющего выполнять низкоуровневые вызовы функций ядра, например Go или C. Кроме того, необходимо помнить о нескольких важных аспектах низкоуровневого программирования, например как избежать порождения новых процессов при обновлении метрик. Статья Runtime Metrics на сайте Docker (https://docs.docker.com/articles/runmetrics) описывает этот подход и подробно рассматривает разнообразные метрики, которые можно получить из ядра. После определения набора значений, требующих контроля, можно обратить внимание на инструментальные средства, такие как statsd (https://github.com/etsy/statsd), предназначенные для сбора и вычисления метрик, InfluxDB (http://influxdb.com/) и OpenTSBD (http://opentsbd.net) для хранения данных, и Graphite (http://graphite.readthedocs.org) и Grafana (https://github.com/grafana/grafana) для вывода результатов.

Контроль и оповещение с использованием Logstash

Logstash в основном представляет собой инструмент ведения журналов, тем не менее заслуживает внимания тот факт, что даже с его помощью можно обеспечить определенный уровень контроля и что сами по себе журнальные данные являются важными метриками для контроля.

Например, можно проверять коды состояния nginx и автоматически отправлять сообщения (электронной почтой или другим способом) о слишком большом количестве отказов сервера (код 500 и выше). Кроме того, в Logstash имеются модули вывода для многих известных решений, обеспечивающих контроль, включая Nagios, Ganglia и Graphite.

В большинстве случаев потребуется применение существующего инструментального средства для сбора и обработки метрик, а также для оформления их визуального представления. Для этой цели предлагается множество коммерческих решений, но мы будем рассматривать только самые удачные решения с открытым исходным кодом, специализированные для работы с контейнерами.

cAdvisor

cAdvisor (сокращение от Container Advisor) от компании Google является наиболее широко применяемым инструментальным средством контроля для контейнеров Docker. Этот инструмент обеспечивает графическое представление использования ресурсов и метрик производительности контейнеров, работающих на заданном хосте.

Поскольку cAdvisor сам доступен в виде контейнера, можно начать его использование сразу после загрузки. Просто запустите контейнер cAdvisor со следующими аргументами:

```
$ docker run -d --name cadvisor \
  -v /:/rootfs:ro \
  -v /var/run:/var/run:rw \
  -v /sys:/sys:ro \
  -v /var/lib/docker/:/var/lib/docker:ro \
  -p 8080:8080 \
  google/cadvisor:latest
```

Если вы работаете на хосте под управлением Red Hat (или CentOS), то потребуется также монтирование каталога *cgroups* с помощью аргумента `--volume=/cgroup:/cgroup`.

После запуска контейнера в браузере задайте адрес *http://localhost:8080*. Вы должны увидеть страницу с набором схем, подобную изображенной на рис. 10.5. Здесь можно перейти к отдельным контейнерам, сначала щелкнув по ссылке Docker Containers, затем щелкая по именам интересующих вас контейнеров.

Рис. 10.5. График использования процессора, предоставленный инструментом cAdvisor

cAdvisor объединяет и обрабатывает разнообразные статистические данные, а также обеспечивает через REST API их доступность для дальнейшей обработки и хранения. Данные можно также экспортировать в InfluxDB, СУБД, предназначенную для хранения и выдачи по запросам последовательностей данных, собранных за определенные интервалы времени, включая метрики и аналитические данные. В дорожную карту cAdvisor включены такие функциональные возможности, как советы по улучшению и тонкой настройке производительности контейнеров и вспомогательная информация по организации кластеров и использованию средств планирования.

Кластерные решения

cAdvisor – хороший инструмент, но он предназначен для использования на одном хосте. При работе с большой системой необходимо получать статистические данные о контейнерах на всех хостах, а также о самих хостах. Нужна информация о формировании групп контейнеров, представляющая как отдельные подсистемы, так и определенные уровни функциональности, обеспечиваемые различными экземплярами. Например, может потребоваться сводка использования оперативной памяти всеми контейнерами nginx или график использования процессоров группой контейнеров, выполняющих задачу анализа данных. Так как требуемые метрики почти всегда зависят от особенностей конкретного приложения и соответствующей решаемой задачи, качественное решение должно предоставить пользователю язык запросов, на котором можно создавать новые метрики и визуальные представления.

Компания Google разработала решение для контроля кластера на основе cAdvisor и назвала его Heapster, но во время написания книги это решение поддерживало только Kubernetes и CoreOS, поэтому здесь мы не будем его рассматривать.

Вместо этого обратим свое внимание на Prometheus, решение для контроля кластера с открытым исходным кодом от компании SoundCloud, которое может принимать входные данные из самых разнообразных источников, включая cAdvisor. Решение предназначено для поддержки крупномасштабных архитектур микросервисов и используется компаниями SoundCloud и Docker Inc.

Prometheus

Решение Prometheus (http://prometheus.io) примечательно тем, что основано на *модели компонентов* (*pull-based model*). Предполагается, что приложения сами предъявляют свои метрики, которые «вытаскивает» (pull) сервер Prometheus вместо обычной отправки метрик непосредственно в механизм Prometheus. Пользовательский интерфейс Prometheus можно применять для формирования запросов и схем данных в интерактивном режиме, а отдельный модуль PromDash предназначен для сохранения схем, графиков и измерительных инструментов на панели управления. Также в Prometheus имеется компонент Alertmanager, способный объединять и блокировать предупреждающие сообщения и перенаправлять их в сервисы уведомления, например уведомления по электронной почте и специализированные сервисы, такие как PagerDuty (http://www.pagerduty.com) и Pushover (https://pushover.net).

Рассмотрим практическое применение Prometheus для приложения identidock. Мы не будем добавлять каких-либо особенных метрик, но сделать это было бы просто с помощью библиотеки client языка Python для оформления вызовов в нашем исходном коде.

Вместо этого мы установим соединение Prometheus с контейнером cAdvisor. Для этого можно было бы воспользоваться проектом container-exporter (https://github.com/docker-infra/container_exporter), который также использует библиотеку libcontainer от Docker. Если контейнер cAdvisor уже работает, то вы можете наблюдать метрики, предъявляемые им в Prometheus в точке входа `/metrics`:

```
$ curl localhost:8080/metrics
# HELP container_cpu_system_seconds_total Cumulative system cpu time consume...
# TYPE container_cpu_system_seconds_total counter
container_cpu_system_seconds_total{id="/",name="/"} 97.89
container_cpu_system_seconds_total{id="/docker",name="/docker"} 40.66
container_cpu_system_seconds_total{id="/docker/071c24557187c14fb7a2504612d4c...
container_cpu_system_seconds_total{id="/docker/1a1a57b885f33d2e16e85cee7f138...
...
```

Процедура запуска контейнера Prometheus проста и очевидна, но все же требует создания файла конфигурации. Сохраните следующие инструкции в файле *prometheus.conf*:

```
global:
  scrape_interval: 1m  ❶
  scrape_timeout: 10s
  evaluation_interval: 1m

scrape_configs:

- job_name: prometheus
  scheme: http  ❷
  target_groups:
  - targets:
    - 'cadvisor:8080'
    - 'localhost:9090'  ❸
```

❶ Сообщает Prometheus о необходимости извлекать статистические данные каждые пять секунд, но установке этого значения следует уделить особое внимание. В реальной среде эксплуатации нужно выбирать интервал с учетом накладных расходов на извлечение данных, сравнивая их с издержками, связанными с устареванием метрик.
❷ Сообщает Prometheus URL для сбора данных от cAdvisor (мы будем использовать ссылку для установки имени хоста).
❸ Также собирать собственные метрики Prometheus в конечном пункте на порте 9090.

Запустите контейнер Prometheus со следующими аргументами:

```
$ docker run -d --name prometheus -p 9090:9090 \
             -v $(pwd)/prometheus.conf:/prometheus.conf \
             --link cadvisor:cadvisor \
             prom/prometheus -config.file=/prometheus.conf
```

После этого должна появиться возможность открыть приложение Prometheus по адресу *http://localhost:9090*. Домашняя страница выдает некоторую информацию о конфигурации Prometheus и состояние точки входа, в которой должны извлекаться данные. На вкладке **Graph** (Схема) можно начать исследование данных внутри приложения Prometheus. Prometheus предлагает собственный язык запросов, поддерживающий фильтры, регулярные выражения и разнообразные операторы. В качестве простого примера попробуйте ввести следующее выражение:

```
sum(container_cpu_usage_seconds_total {name=~"logging*"}) by (name)
```

Эта команда должна сформировать график использования процессора в течение некоторого интервала времени для каждого контейнера из приложения identidock. Выражение `{name=~"logging*"}` позволяет исключить те контейнеры, которые не являются частью Compose-приложения (например, сами инструменты cAdvisor и Prometheus). При необходимости замените образец `"logging"` на имя своего проекта Compose или соответствующего каталога. Функция `sum` обязательна, так как по умолчанию отслеживается использование каждого отдельного процессора. Результат должен быть похож на изображение на рис. 10.6.

Можно сделать больше и настроить панель управления с помощью контейнера PromDash. Эта процедура очень проста, поэтому ее выполнение предлагается читателю в качестве самостоятельного задания. На рис. 10.7 показана панель управления с описанной выше метрикой процессора и графиком использования оперативной памяти. Кроме того, PromDash поддерживает вывод графиков по данным нескольких распределенных экземпляров Prometheus, что может оказаться полезным для получения графиков по отдельным местам географического расположения или по отделам предприятия.

Разумеется, это самый простой пример применения инструмента Prometheus. В реальной эксплуатационной среде сбор данных будет выполняться с большого количества точек входа, распределенных по разным хостам, а в конфигурацию будут включены настройки разнообразных панелей управления и подробных графических представлений с использованием PromDash, а также настройка системы уведомлений с помощью Alertmanager.

Коммерческие решения, обеспечивающие контроль и ведение журналов

В этой главе я намеренно рассматривал во всех подробностях только внедряемые локально решения с открытым исходным кодом, но следует отметить, что существует множество глубоко проработанных коммерческих решений с технической поддержкой производителя. Здесь конкуренция весьма высока и продолжает постоянно возрастать. Я не буду перечислять специализированные ре-

шения в этой быстро развивающейся отрасли, скажу лишь, что они несомненно заслуживают внимания, особенно если вы ищете зрелое и/или решение с удаленным хостингом.

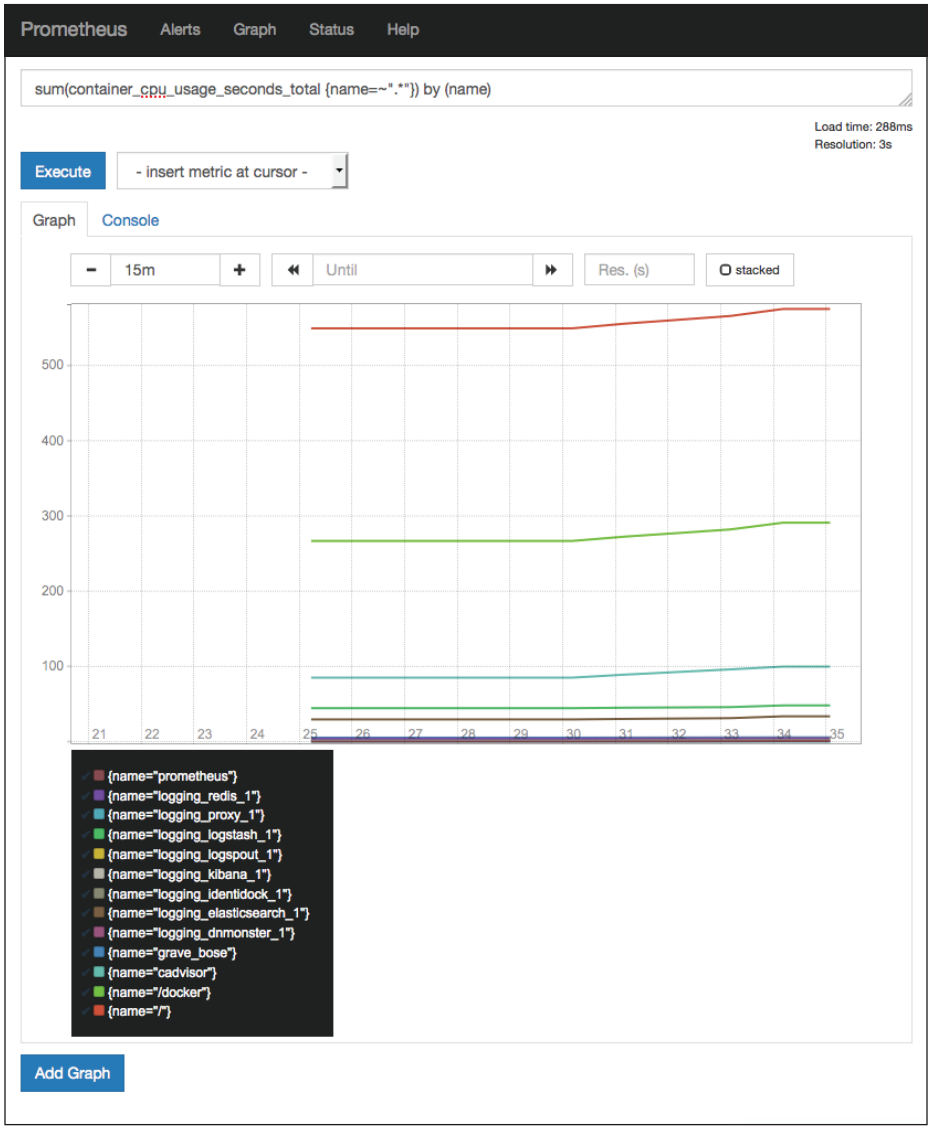

Рис. 10.6. График использования процессора, созданный инструментом Prometheus

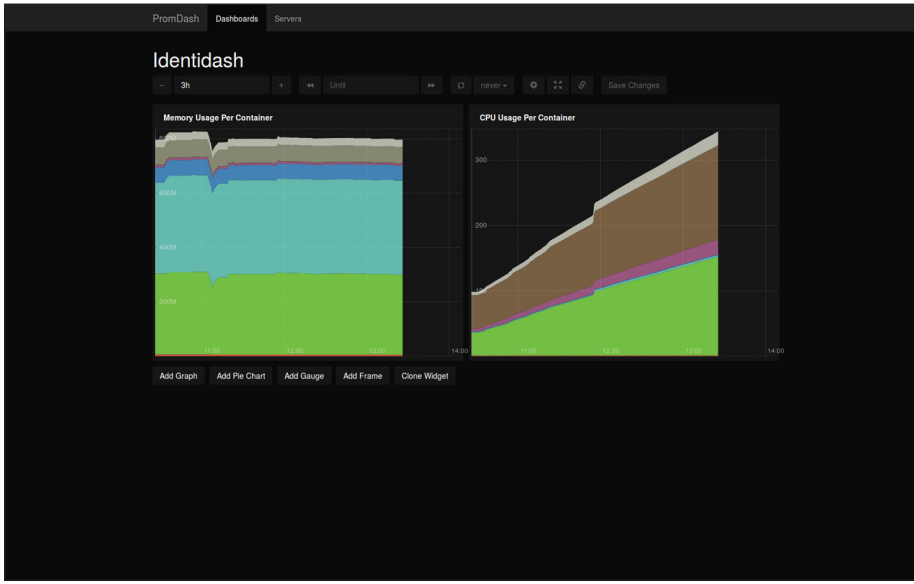

Рис. 10.7. Панель управления для identidock, созданная при помощи PromDash

Резюме

Правильное ведение журналов и надежный контроль крайне важны для работы приложения, основанного на микросервисах. В этой главе продемонстрирована возможность эффективной организации ведения журналов и контроля приложения identidock с использованием ELK-стека, а также инструментальных средств cAdvisor и Prometheus. Несмотря на то что это решение выглядит гораздо более «тяжеловесным», чем само приложение (в том смысле, что сами процессы ведения журналов и контроля являются преобладающими метриками), мы на примере увидели, как быстро и просто можно получить эффективное решение с помощью этих средств.

В будущем мы можем рассчитывать на расширенные возможности поддержки ведения журналов средствами самой среды Docker. Коммерческие продукты, обеспечивающие ведение журналов, контроль и уведомления, занимают прочные позиции, поэтому ожидается, что все производители в конечном итоге придут к решениям на основе Docker и микросервисов.

Часть III

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА И МЕТОДИКИ

В части III подробнейшим образом рассматриваются инструментальные средства и методики, необходимые для поддержки безопасной и надежной работы кластеров, состоящих из контейнеров Docker.

Мы начнем с описания сетевой среды и механизма обнаружения сервисов, поскольку эта задача сразу становится чрезвычайно важной для обеспечения работы контейнеров, расположенных на нескольких хостах. Можно поставить вопрос подругому: как контейнеры находят друг друга и как устанавливается соединение между ними?

Затем мы рассмотрим программные решения, предназначенные для организации кластеров, состоящих из контейнеров. Эти инструменты помогают разработчику решать такие вопросы, как балансировка нагрузки, масштабирование и обеспечение устойчивости к критическим сбоям, а также поддерживают организацию планирования операций контейнеров и способствуют максимальному использованию ресурсов. Любое приложение с длительным жизненным циклом рано или поздно (скорее, рано) столкнется с этими проблемами, поэтому предварительное изучение проблем и потенциальных решений дает значительное преимущество.

В последней главе рассматривается обеспечение безопасности контейнеров и развертываемых микросервисов. Контейнеры породили новые проблемы в области безопасности, но при этом также предложили новые инструментальные средства и методики. Несмотря на расположение в самом конце книги, эта тема является весьма важной для всех читателей, чья деятельность так или иначе связана с контейнерами.

Глава 11

Сетевая среда и обнаружение сервисов

В контексте использования контейнеров линия, разделяющая механизм обнаружения сервисов и сетевую среду, неожиданно может стать неразличимой. Обнаружение сервисов (service discovery) – это процесс автоматического предоставления клиентам[1] сервиса с информацией, необходимой для установления соединения (обычно это IP-адрес и номер порта) с подходящим[2] экземпляром данного сервиса. Задача решается просто в статической системе на одном хосте, где существуют лишь единственные экземпляры всех сервисов, но намного усложняется в распределенной системе с многочисленными экземплярами сервисов, постоянно создаваемыми, изменяющимися и удаляемыми. Один из способов обнаружения сервисов состоит в том, что клиент просто запрашивает сервис по имени (например, `db` или `api`), и некоторые скрытые операции во внутренних компонентах в итоге должны указать ему требуемую локацию. Подобная «скрытность» может проявляться в форме простых контейнеров-посредников, решения обнаружения сервисов, такого как Consul, общего сетевого решения, такого как Weave (включающего возможности обнаружения сервисов) или какой-либо комбинации перечисленных выше решений.

Для наших целей решения сетевой среды могут рассматриваться как процесс установления соединений, связывающий контейнеры. Это не подразумевает обязательное подключение физических кабелей Ethernet, хотя часто используются программные аналоги, такие как veth. Организация сетевой среды контейнеров начинается с предположения о том, что существуют доступные маршруты между хостами, причем не важно, проходят ли эти маршруты через общедоступную сеть Интернета или всего лишь обеспечиваются простым локальным коммутатором.

Таким образом, механизм обнаружения сервисов позволяет клиентам находить нужные экземпляры, а сетевая среда принимает на себя все обязанности по установле-

---

[1] Здесь я использую термин «клиент» в широком смысле; в первую очередь имеются в виду приложения и прочие сервисы, работающие как внутренние компоненты систем, а также равноправные компоненты (в смысле кластера, состоящего из совместно работающих экземпляров) и клиенты конечного пользователя, такие как браузеры.

[2] Здесь смысл слова «подходящий» сильно зависит от контекста, оно может означать «любой», «самый быстрый», «расположенный ближе прочих к запрашиваемым данным» и т. д.

нию соответствующих соединений. Организация сетевой среды и решения по обнаружению сервисов в настоящее время все больше объединяются по функциональности, поскольку решения по обнаружению сервисов предлагаются для разнородных сетевых сред, а решения по организации сетевой среды часто включают функции обнаружения сервисов (например, Weave). Решение, ориентированное исключительно на обнаружение сервисов, подобное Consul, вероятнее всего, предложит более богатую функциональность в отношении проверки работоспособности, устойчивости к критическим сбоям и балансировки нагрузки. Решения по организации сетевой среды предлагают разнообразные возможности для установления соединений и планирования маршрутизации между контейнерами[1] в дополнение к основным функциям, таким как шифрование трафика и изоляция групп контейнеров.

Достаточно часто возникает необходимость в использовании и решения по обнаружению сервисов, и средств организации сетевой среды (определенная работа по организации сетевой среды необходима всегда). Что именно требуется, будет зависеть от конкретной ситуации, но наилучшие практические методики продолжают развиваться. Организация сетевой среды, вероятнее всего, будет распределяться между средами разработки, тестирования и эксплуатации, но механизм обнаружения сервисов обычно подразумевает решения на уровне приложений и остается неизменным во всех средах.

В этой главе мы попытаемся рассмотреть в целом всю задачу организации сетевой среды и обнаружения сервисов со всеми ее сложностями, поэтому начнем с самого простого решения для связи нескольких хостов – контейнеров-посредников, потом перейдем к решениям по обнаружению сервисов, в том числе etcd и Consul, затем рассмотрим подробности организации сетевой среды Docker и такие комплексные решения, как Weave, Flannel и Project Calico.

Посредники

Использование *посредников* (*ambassadors*) представляет собой один из способов установления соединений между контейнерами, расположенными на разных хостах. Это промежуточные контейнеры (прокси-контейнеры), которые замещают реальный контейнер (или сервис) и перенаправляют трафик в действительный сервис. Посредники обеспечивают разделение обязанностей, что делает их полезными во многих ситуациях, а не только при установлении соединений между сервисами, расположенными на разных хостах.

Главное преимущество контейнеров-посредников состоит в том, что они позволяют отделить эксплуатационную сетевую архитектуру от архитектуры этапа разработки без внесения каких-либо изменений в код. Разработчики могут использовать локальные версии баз данных и прочих ресурсов, точно зная, что посредники

---

[1] Необходимость применения «отдельного», независимого механизма обнаружения сервисов сохраняется при использовании в некоторых сетевых средах, вне зависимости от того, является ли среда принятой по умолчанию в Docker с использованием моста сетью с портами, объявленными на хосте, или самой сетью хоста, — в обоих случаях требуется управление портами.

могут переориентировать приложение на использование указанных кластерных сервисов или удаленных ресурсов без каких-либо изменений в исходном коде. Кроме того, посредники способны оперативно, без паузы в работе переключаться на другой внутренний сервис, в то время как использование прямого соединения с конкретным сервисом потребует перезапуска контейнера-клиента.

Недостаток контейнеров-посредников заключается в том, что для них требуется дополнительная настройка, соответственно, и дополнительные издержки, а кроме того, посредники представляют собой потенциальные точки отказа. Посредники могут быстро становиться чрезмерно сложными, а при большом количестве соединений существенно усложняется и управление ими.

На рис. 11.1 показан типичный пример разработки, когда разработчик устанавливает прямое соединение приложения с контейнером базы данных, используя механизм соединений Docker, причем все работает на локальном компьютере. Это удобно для быстрого внесения изменений, и в процессе разработки можно отказываться от каких-то решений и начинать реализацию альтернатив.

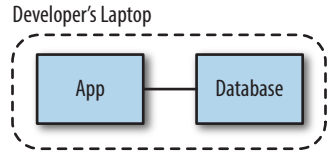

Рис. 11.1. Пример разработки без посредников

На рис. 11.2 мы видим обобщенный пример эксплуатации, в котором используется посредник для установления соединения между приложением и необходимым для работы сервисом, работающим на отдельном сервере. Здесь необходимо сконфигурировать конкретный экземпляр посредника для передачи трафика в сервис и использовать соответствующий механизм для установления соединения приложения с посредником. В коде продолжается использование тех же имен хостов и номеров портов, что и прежде, а необходимые преобразования выполняются внутри посредника.

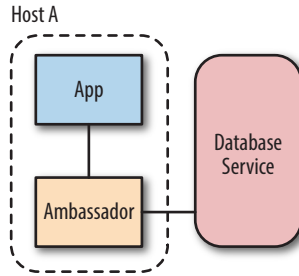

Рис. 11.2. Использование посредника для установления соединения с эксплуатационным сервисом

На рис. 11.3 показано, как приложение обменивается информацией с контейнером на удаленном хосте, причем сам этот контейнер действует через посредника. Такая конфигурация позволяет удаленному хосту направлять трафик в новый контейнер с новым адресом, просто изменяя параметры конфигурации посредника. И в этом случае для перехода к новой конфигурации не требуются никакие изменения в исходном коде.

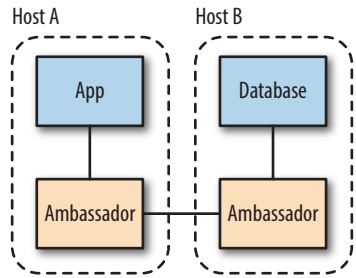

Рис. 11.3. Использование посредников
для установления соединения с удаленным контейнером

Сам по себе посредник может быть очень простым контейнером, ведь все, что от него требуется, – это установление соединения между приложением и необходимым сервисом. Официальных образов для создания посредников не существует, поэтому вам придется сформировать собственный или выбрать подходящий из пользовательских образов на Docker Hub.

Образ amouat/ambassador

В этой главе используется образ с именем `amouat/ambassador`. Этот образ представляет собой простой порт посредника Свена Доуидейта (Sven Dowideit) (https://hub.docker.com/r/svendowideit/ambassador/), адаптированного для использования базового образа `alpine` и запускаемого в режиме автоматической сборки на Docker Hub. Этот образ использует утилиту socat (http://www.dest-unreach.org/socat/) для настройки передачи данных между посредником и заданной целью. Переменные среды в той же форме, в которой они были созданы механизмом соединений Docker (например, `REDIS_PORT_6379_TCP=tcp://172.17.0.1:6379`), определяют целевой пункт соединения. Это означает, что для передачи данных в контейнер по локальному соединению (например, на хосте B на рис. 11.3) потребуется крайне мало усилий по настройке.
Так как данный образ основан на минималистичном дистрибутиве Alpine Linux, размер его не превышает 7 Мб. Такой маленький образ быстро загружается, и накладные расходы при его использовании в системе невелики.

Рассмотрим подробнее, как можно воспользоваться посредниками для установления соединения нашего приложения identidock с контейнером Redis, работающим на отдельном хосте, как на рис. 11.3. Для этого создадим две виртуальные машины VirtualBox с помощью Docker Machine (см. раздел «Предоставление ре-

сурсов с помощью Docker Machine» главы 9), но вы можете просто запустить этот пример, используя различные Docker-хосты в облачной среде. Подготовка хостов:

```
$ docker-machine create -d virtualbox redis-host
...
$ docker-machine create -d virtualbox identidock-host
...
```

Теперь настроим контейнер Redis (с именем `real-redis`) и посредник (с именем `real-redis-ambassador`) на хосте `redis-host`:

```
$ eval $(docker-machine env redis-host)
$ docker run -d --name real-redis redis:3
Unable to find image 'redis:3' locally
3: Pulling from redis
...
60bb8d255b950b1b34443c04b6a9e5feec5047709e4e44e58a43285123e5c26b
$ docker run -d --name real-redis-ambassador \
             -p 6379:6379 \  ❶
             --link real-redis:real-redis \  ❷
             amouat/ambassador
be613f5d1b49173b6b78b889290fd1d39dbb0fda4fbd74ee0ed26ab95ed7832c
```

❶ Необходимо сделать общедоступным порт 6379 на этом хосте, чтобы обеспечить возможность установления соединения с удаленным хостом.

❷ Посредник использует переменные среды из связанного с ним контейнера `real-redis`, чтобы настроить передачу данных с перенаправлением потока запросов, приходящих в порт 6379, в контейнер real-redis.

Далее необходимо настроить посредник на хосте `identidock-host`:

```
$ eval $(docker-machine env identidock-host)
$ docker run -d --name redis_ambassador --expose 6379 \
         -a REDIS_PORT_6379_TCP=tcp://$(docker-machine ip redis-host):6379 \  ❶
         amouat/ambassador
Unable to find image 'amouat/ambassador:latest' locally
latest: Pulling from amouat/ambassador
31f630c65071: Pull complete
cb9fe39636e8: Pull complete
3931d220729b: Pull complete
154bc6b29ef7: Already exists
Digest: sha256:647c29203b9c9aba8e304fabfd194429a4138cfd3d306d2becc1a10e646fcc23
Status: Downloaded newer image for amouat/ambassador:latest
26d74433d44f5b63c173ea7d1cfebd6428b7227272bd52252f2820cdd513f164
```

❶ Здесь необходимо вручную установить значение переменной среды, чтобы посредник мог установить соединение с удаленным хостом. IP-адрес удаленного хоста извлекается с помощью команды `docker-machine ip`.

Теперь запускаем identidock и dnmonster и устанавливаем соединение identidock со своим посредником:

```
$ docker run -d --name dnmonster amouat/dnmonster:1.0
Unable to find image 'amouat/dnmonster:1.0' locally
1.0: Pulling from amouat/dnmonster
...
c7619143087f6d80b103a0b26e4034bc173c64b5fd0448ab704206b4ccd63fa
$ docker run -d --link dnmonster:dnmonster --link redis_ambassador:redis \
            -p 80:9090 amouat/identidock:1.0
Unable to find image 'amouat/identidock:1.0' locally
1.0: Pulling from amouat/identidock
...
5e53476ee3c0c982754f9e2c42c82681aa567cdfb0b55b48ebc7eea2d586eeac
```

Проверяем работу всей системы:

```
$ curl $(docker-machine ip identidock-host)
<html><head>...
```

Превосходно. Только что мы распределили работу приложения identidock по различным хостам без каких-либо изменений в исходном коде, просто используя для этого два небольших контейнера-посредника. Реализация этого подхода требует немного больше усилий, появляются два дополнительных контейнера, но все же он очень прост и гибок. Легко представить себе ситуации, в которых действуют более сложно организованные посредники, например:

- шифрование трафика в ненадежном соединении;
- автоматическое установление соединения между контейнерами при их запуске в потоке, контролируемом механизмом обработки событий Docker;
- организация прокси-сервиса для запросов на чтение на сервере, защищенном от записи, и перенаправления запросов на запись на другой сервер.

Во всех случаях клиенту нет необходимости знать, что именно выполняется в посреднике.

Несмотря на очевидную полезность контейнеров-посредников, в большинстве случаев проще и эффективнее, с точки зрения масштабируемости, использовать свойства сетевой среды и/или решения по обнаружению сервисов для поиска и установления соединения с удаленными сервисами и контейнерами.

Обнаружение сервисов

В начале этой главы было приведено определение обнаружения сервисов как процесса автоматического предоставления клиенту требуемого сервиса с соответствующей информацией, необходимой для установления соединения с наиболее подходящим экземпляром этого сервиса.

Для приложения-клиента это означает, что он должен запросить или получить каким-то образом адрес требуемого сервиса. Мы рассмотрим решения, в которых клиент должен явно обращаться к функциям интерфейса прикладного программирования для получения адреса сервиса, а также решения на основе DNS, которые легко встраиваются в существующие приложения.

В этом разделе описаны основные решения по обнаружению сервисов, используемые в среде Docker на текущий момент. Подробно рассматриваются etcd, Consul и SkyDNS, затем дается краткий обзор некоторых других решений, заслуживающих внимания. Ни одно из этих решений не было создано специально для контейнеров, но все они предназначены для крупных распределенных систем.

etcd

etcd – это распределенное хранилище данных в формате «ключ-значение». Это реализация *алгоритма решения задачи консенсуса Raft* (https://raftconsensus.github.io/) на языке Go, выполненная с учетом обеспечения эффективности и отказоустойчивости. *Консенсус* (*consensus*) представляет собой процесс согласования несколькими членами кластера процедуры записи значений данных. Такой процесс быстро усложняется из-за ошибок и критических сбоев. Алгоритм Raft гарантирует логическую целостность значений и допускает добавление новых записей только при получении большинства голосов членов кластера.

Каждый член в кластере etcd запускает экземпляр бинарного файла etcd, который обменивается информацией с другими членами. Клиенты получают доступ к etcd через интерфейс REST, предоставляемый всеми членами кластера.

Рекомендуемый минимальный размер кластера etcd равен 3 для обеспечения надежности и устойчивости в случае критического сбоя. Но в следующем примере мы будем использовать только два члена кластера, чтобы более наглядно показать, как работает etcd.

Оптимальный размер кластера

И для etcd, и для Consul рекомендуется установить размер кластера равным 3, 5 или 7 для сохранения баланса между устойчивостью к критическим сбоям и производительностью.

В случае с одним членом данные будут потеряны при любом критическом сбое. Если имеются два члена и один из них становится неработоспособным, то оставшийся член не сможет обеспечить «кворум»[1], и все последующие попытки записи данных будут неудачными до тех пор, пока второй член не вернется в рабочее состояние.

Таблица 11.1. Обоснование размера кластера

Сервер	No. required for majority	Failure tolerance
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

[1] Проще говоря, обычное большинство голосов.

Из табл. 11.1 можно понять, что добавление членов кластера улучшает устойчивость к критическим сбоям. Но увеличение числа членов также означает необходимость согласования каждой процедуры записи и обмен данными между бóльшим количеством узлов, что замедляет работу системы в целом. Когда количество членов кластера превышает 7, вероятность отказа такого количества узлов, которое способно «сломать» всю систему, становится настолько низкой, что дальнейшее наращивание числа узлов не имеет смысла по соображениям сохранения достаточно высокой производительности. Также следует отметить, что четное количество членов, вообще говоря, не рекомендуется, поскольку при этом увеличивается размер кластера (следовательно, снижается производительность), а устойчивость к критическим сбоям не улучшается.

Разумеется, многие распределенные системы с гораздо бóльшим количеством хостов. В таких случаях 5 или 7 хостов используются для формирования кластера, а на остальных узлах запускаются клиенты, которые могут выполнять запросы к системе, но не принимают участия в согласовании репликации данных. Такой подход реализован с помощью механизма прокси в etcd и с использованием режима клиента (client mode) в Consul.

Начнем с создания новых хостов с помощью Docker Machine:

```
$ docker-machine create -d virtualbox etcd-1
...
$ docker-machine create -d virtualbox etcd-2
...
```

Теперь можно запустить контейнеры etcd. Так как заранее известно количество членов кластера etcd, мы просто перечислим их явно при запуске контейнеров. Также возможно использование поискового механизма на основе URL или DNS для кластеров, в которых адреса членов заранее неизвестны. При запуске etcd необходимо установить большое количество флагов, поэтому я воспользовался переменными среды для хранения IP-адресов виртуальных машин, что сделало команду запуска немного проще:

```
$ HOSTA=$(docker-machine ip etcd-1)
$ HOSTB=$(docker-machine ip etcd-2)
$ eval $(docker-machine env etcd-1)
$ docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
  --name etcd quay.io/coreos/etcd \  ❶
  -name etcd-1 -initial-advertise-peer-urls http://${HOSTA}:2380 \  ❷
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  -advertise-client-urls http://${HOSTA}:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster etcd-1=http://${HOSTA}:2380,etcd-2=http://${HOSTB}:2380 \  ❸
  -initial-cluster-state new
...
d4c12bbb16042b11252c5512ab595403fefcb2f46abb6441b0981103eb596eed
```

❶ Получение официального образа etcd из реестра quay.io.
❷ Настройка различных URL для доступа к etcd. Необходимо обеспечить для etcd прослушивание IP-адресов по шаблону `0.0.0.0` для удаленных и локальных соединений, но указать другим клиентам и «коллегам», что следует устанавливать соединение по IP-адресу данного хоста. В реальных условиях узлы кластера etcd должны обмениваться данными в пределах внутренней сети, не предоставляя прямого доступа из внешнего мира (то есть их адреса не должны устанавливаться командой `docker-machine ip`).
❸ Здесь явно перечисляются все узлы кластера, включая и сам запускаемый узел. Эту часть команды можно заменить на другие методы обнаружения.

Для второй виртуальной машины параметры настройки почти такие же, за исключением того, что нужно объявить IP-адрес `etcd-2` всем внешним клиентам:

```
$ eval $(docker-machine env etcd-2)
$ docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
  --name etcd quay.io/coreos/etcd \
  -name etcd-2 -initial-advertise-peer-urls http://${HOSTB}:2380 \
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  -advertise-client-urls http://${HOSTB}:2379 \
  -initial-cluster-token etcd-cluster-1
  -initial-cluster etcd-1=http://{HOSTA}:2380,etcd-2=http://${HOSTB}:2380 \
  -initial-cluster-state new
...
2aa2d8fee10aec4284b9b85a579d96ae92ba0f1e210fb36da2249f31e556a65e
```

Теперь кластер etcd работает. Можно получить список членов кластера etcd с помощью простого запроса `curl` с использованием HTTP API:

```
$ curl -s http://$HOSTA:2379/v2/members | jq '.'
{
  "members": [
  {
    "clientURLs": [
      "http://192.168.99.100:2379"
    ],
    "peerURLs": [
      "http://192.168.99.100:2380"
    ],
    "name": "etcd-1",
    "id": "30650851266557bc"
  },
  {
    "clientURLs": [
      "http://192.168.99.101:2379"
    ],
    "peerURLs": [
      "http://192.168.99.101:2380"
    ],
```

```
    "name": "etcd-2",
    "id": "9636be876f777946"
  }
  ]
}
```

Здесь использована утилита `jq` для аккуратного форматирования при выводе результатов. Можно динамически добавлять и удалять членов кластера, посылая HTTP-запросы `POST` и `DELETE` в ту же самую точку входа.

Следующий этап – добавление некоторых хранимых данных и проверка их корректности с помощью операции чтения с обоих хостов. Данные хранятся в каталогах внутри контейнера etcd и возвращаются в формате JSON. В следующем примере значение `service_address` сохраняется в каталоге *service_name* с помощью HTTP-запроса `PUT`:

```
$ curl -s http://$HOSTA:2379/v2/keys/service_name \
          -XPUT -d value="service_address" | jq '.'
{
  "node": {
    "createdIndex": 17,
    "modifiedIndex": 17,
    "value": "service_address",
    "key": "/service_name"
  },
  "action": "set"
}
```

Для получения сохраненных данных достаточно выполнить запрос `GET` в соответствующий каталог:

```
$ curl -s http://$HOSTB:2379/v2/keys/service_name | jq '.'
{
  "node": {
    "createdIndex": 17,
    "modifiedIndex": 17,
    "value": "service_address",
    "key": "/service_name"
  },
  "action": "get"
}
```

По умолчанию etcd возвращает некоторые метаданные о ключе и о соответствующем ему значении. Обратите внимание на то, что мы записали данные в `etcd-1`, а считали их из `etcd-2`. Поскольку оба узла являются членами одного кластера, не имеет значения, какой хост используется для операций, – оба дают одинаковый ответ.

Существует также клиент с интерфейсом командной строки `etcdctl`, который можно применять для обмена данными с кластером etcd. Чтобы не устанавливать эту программу, воспользуемся соответствующим контейнером:

```
$ docker run binocarlos/etcdctl -C ${HOSTB}:2379 get service_name
service_address
```

Можно лучше понять работу etcd, изучая журналы его контейнеров, в которых показано, каким образом члены кластера выбирают лидера, а также многие другие подробности. Полное описание и спецификацию внутреннего алгоритма Raft можно найти на сайте https://raftconsensus.github.io/.

Теперь можно понять, как написать приложение, которое напрямую использует etcd для обнаружения сервисов. Для нашего приложения identidock можно было бы сформировать простой HTTP-запрос в коде Python для поиска адресов сервисов Redis и dnmonster. Кроме того, можно было бы изменить контейнеры dnmonster и Redis таким образом, чтобы они регистрировали свои адреса в etcd при запуске, полностью автоматизируя систему.

Но мы не будем менять код identidock, вместо этого в следующем разделе, посвященном SkyDNS, мы рассмотрим, как можно на основе etcd создать решение по обнаружению сервисов, не требующее никаких изменений в исходном коде.

SkyDNS

Утилита SkyDNS (https://github.com/skynetservices/skydns) предлагает основанное на DNS решение по обнаружению сервисов, используя для этого etcd. Наиболее примечательно то, что эта утилита используется механизмом контейнеризации Google Container Engine для реализации обнаружения сервисов в предлагаемом Google продукте Kubernetes (см. раздел «Kubernetes» главы 12).

Мы можем воспользоваться SkyDNS для получения полноценного решения на основе etcd и получить приложение identidock, работающее на двух хостах без каких-либо изменений в его исходном коде. Если вы успешно выполнили самый последний пример из предыдущего раздела, то у вас есть два сервера, работающих в кластере etcd: `etcd-1` с IP-адресом `$HOSTA` и `etcd-2` с IP-адресом `$HOSTB`. После завершения выполнения текущего примера мы получим систему, изображенную на рис. 11.4, где контейнер identidock использует SkyDNS для поиска контейнеров dnmonster и Redis.

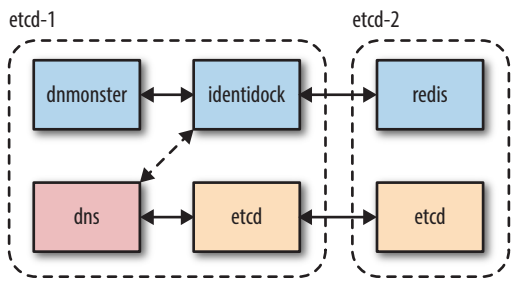

Рис. 11.4. Приложение identidock,
распределенное между хостами и использующее SkyDNS и etcd

В первую очередь нужно добавить некоторые параметры настройки SkyDNS в конфигурацию etcd, чтобы стало возможным использование SkyDNS:

```
$ curl -XPUT http://${HOSTA}:2379/v2/keys/skydns/config \
       -d value='{"dns_addr":0.0.0.0:53", "domain":"identidock.local."}' | jq .
{
  "action": "set",
  "node": {
    "key": "/skydns/config",
    "value": "{\"dns_addr\":0.0.0.0:53\", \"domain\":\"identidock.local.\"}",
    "modifiedIndex": 6,
    "createdIndex": 6
  }
}
```

При таких параметрах SkyDNS будет прослушивать все интерфейсы на порте 53, тем самым обеспечивая авторизацию для домена *identidock.local*.

В нашем примере имеет смысл воспользоваться контейнером SkyDNS, но можно также запустить эту программу как процесс хоста. Возьмем образ `skynetservices/skydns`[1], созданный разработчиками SkyDNS, и запустим его на узле `etcd-1`:

```
$ eval $(docker-machine env etcd-1)
$ docker run -d -e ETCD_MACHINES="http://${HOSTA}:2379,http://${HOSTB}:2379" \
             --name dns skynetservices/skydns:2.5.2a
...
f95a871247163dfa69cf0a974be6703fe1dbf6d07daad3d2fa49e6678fa17bd9
```

Раньше приходилось добавлять дополнительные параметры конфигурации для определения области поиска внутреннего механизма etcd, но сейчас у нас есть работающий DNS-сервер. Правда, мы пока еще не сообщили ему ни об одном сервисе. Начнем с размещения сервера Redis на узле `etcd-2` и добавления его в таблицу SkyDNS:

```
$ eval $(docker-machine env etcd-2)
$ docker run -d -p 6379:6379 --name redis redis:3
...
d9c72d30c6cbf1e48d3a69bc6b0464d16232e45f32ec00dcebf5a7c6969b6aad
$ curl -XPUT http://${HOSTA}:2379/v2/keys/skydns/local/identidock/redis \
       -d value='{"host":"'$HOSTB'","port":6379}' | jq .
{
  "action": "set",
  "node": {
    "key": "/skydns/local/identidock/redis",
```

[1] Во время написания книги для образа SkyDNS не обеспечивалась его автоматическая сборка, что делало его в определенной мере «черным ящиком». Возможно, вы предпочтете создать собственный контейнер SkyDNS для полной уверенности в его содержимом. Для этой цели существует Dockerfile, доступный в проекте SkyDNS на GitHub (https://github.com/skynetservices/skydns).

```
    "value": "{\"host\":\"192.168.99.101\",\"port\":6379}",
    "modifiedIndex": 7,
    "createdIndex": 7
  }
}
```

Здесь в запросе через `curl` путь, заканчивающийся на */local/identidock/redis*, который отображается в домен *redis.identidock.local*. Данные в формате JSON определяют IP-адрес и порт, необходимые для нахождения соответствующего имени. Здесь задан IP-адрес хоста, а не контейнера Redis, так как IP-адрес этого контейнера является локальным для `etcd-2`.

Сейчас уже можно проверить работу всех компонентов системы. При запуске нового контейнера используем флаг `--dns`, чтобы указать наш контейнер DNS для выполнения всех операций поиска имен:

```
$ eval $(docker-machine env etcd-1)
$ docker run --dns $(docker inspect -f {{.NetworkSettings.IPAddress}} dns) \
             -it redis:3 bash
...
root@3baff51314d6:/data# ping redis.identidock.local
PING redis.identidock.local (192.168.99.101): 48 data bytes
56 bytes from 192.168.99.101: icmp_seq=0 ttl=64 time=0.102 ms
56 bytes from 192.168.99.101: icmp_seq=1 ttl=64 time=0.090 ms
56 bytes from 192.168.99.101: icmp_seq=2 ttl=64 time=0.096 ms
^C--- redis.identidock.local ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.090/0.096/0.102/0.000 ms
root@3baff51314d6:/data# redis-cli -h redis.identidock.local ping
PONG
```

Все работает. Хотя набирать имя *redis.identidock.local* немного утомительно, гораздо лучше, если можно было бы просто ввести *redis*, но это не проходит:

```
root@3baff51314d6:/data# ping redis
ping: unknown host
```

Если запустить новый контейнер и добавить *identidock.local* как домен поиска, то ОС будет автоматически выполнять преобразование в *redis.identidock.local* после неудачной попытки поиска *redis*:

```
root@3baff51314d6:/data# exit
$ docker run --dns $(docker inspect -f {{.NetworkSettings.IPAddress}} dns) \
             --dns-search identidock.local \
             -it redis:3 redis-cli -h redis ping
PONG
```

Теперь все в порядке, но не хотелось бы каждый раз при запуске контейнера вводить длинные аргументы флагов `--dns` и `--dns-search`. Вместо этого можно было бы передавать их как дополнительные параметры в демон Docker, но при этом возникает проблема «первичности курицы или яйца» в том случае, когда сам DNS-

сервер является контейнером[1], поэтому мы рассмотрим другую возможность. Добавим указанные выше значения в файл хоста */etc/resolv.conf*[2], который управляет областью поиска ОС для доменных имен и автоматически распространяет ее на контейнеры:

```
$ docker-machine ssh etcd-1
...
docker@etcd-1:~$ echo -e "domain identidock.local \nnameserver " \
  $(docker inspect -f {{.NetworkSettings.IPAddress}} dns) > /etc/resolv.conf
docker@etcd-1:~$ cat /etc/resolv.conf
domain identidock.local
nameserver 172.17.0.3
docker@etcd-1:~$ exit
```

Проверим, работает ли этот вариант:

```
$ docker run redis:3 redis-cli -h redis ping
PONG
```

Теперь запустим dnmonster и добавим его в таблицу DNS:

```
$ docker run -d --name dnmonster amouat/dnmonster:1.0
$ DNM_IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} dnmonster)
$ curl -XPUT http://$HOSTA:2379/v2/keys/skydns/local/identidock/dnmonster \
       -d value='{"host": "'$DNM_IP'","port":8080}'
...
```

Здесь используется внутренний IP-адрес контейнера для dnmonster, поэтому он будет доступным только из `etcd-1`. Если имеется несколько серверов SkyDNS, работающих на разных хостах, вероятно, потребуется пометка этой записи как *host local*, чтобы не путать ее с записями других серверов. Это можно сделать, определив локальный домен хоста при запуске SkyDNS.

Наконец, мы запускаем identidock и убеждаемся, что он работает правильно без каких-либо явных действий по установлению соединений:

```
$ docker run -d -p 80:9090 amouat/identidock:1.0
$ curl $HOSTA
<html><head><title>...
```

Итак, теперь у нас есть интерфейс обнаружения сервисов, который не требует никаких изменений в исходном коде текущей реализации и работает на основе распределенного и устойчивого к критическим сбоям хранилища данных etcd.

---

[1] Если вы решили передавать демону параметры, то можете объявить порт 53 в контейнере DNS открытым для использования на хосте и использовать адрес Docker-шлюза для DNS-сервера.

[2] Виртуальная машина VirtualBox всегда будет заново создавать этот файл при перезагрузке, уничтожая наши изменения, поэтому приведенные здесь инструкции пригодны лишь в качестве примера. Хосты, находящиеся в реальной эксплуатации, могут использовать другие методы изменения файла resolv.conf, например утилиту `resolvconf`.

Что внутри SkyDNS

Если вы хотите более точно знать, как работает SkyDNS, то можете воспользоваться утилитой `dig`, включенной в образ SkyDNS, например:

```
$ docker exec -it dns sh
/ # dig @localhost SRV redis.identidock.local
dig @localhost SRV redis.identidock.local

; <<>> DiG 9.10.1-P2 <<>> @localhost SRV redis.identidock.local
; (2 servers found) (найдены 2 сервера)
;; global options: +cmd
;; Got answer:  (Получен ответ:)
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51805
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION  (РАЗДЕЛ ЗАПРОСОВ)
;redis.identidock.local.       IN  SRV

;; ANSWER SECTION  (РАЗДЕЛ ОТВЕТОВ)
redis.identidock.local.  3600  IN  SRV  10 100 6379 redis.identidock.local.

;; ADDITIONAL SECTION  (ДОПОЛНИТЕЛЬНЫЙ РАЗДЕЛ)
redis.identidock.local.  3600  IN  A    192.168.99.101

;; Query time: 4 msec  (Время обработки запроса: 4 мс)
;; SERVER: ::1#53(::1)
;; WHEN: Sat Jul 25 17:18:39 UTC 2015
;; MSG SIZE rcvd: 98
```

Здесь DNS-сервер вернул запись `SRV` для `redis.identidock.local`. Полученный результат содержит IP-адрес, номер порта, а также приоритет, относительный вес для записей с одинаковым приоритетом и время жизни (TTL).

SkyDNS использует SRV-записи, указывающие на местоположение сервисов, а также обычные записи типа A (для выполнения преобразований адресов IPv4). В набор полей записи SRV, помимо прочих, включается номер порта для сервиса, время жизни (TTL), приоритет и относительный вес для записей с одинаковым приоритетом. Настройка значения времени жизни позволяет автоматически удалять записи, если клиент или агент не обновляет регулярно этого значения, которое может быть использовано для реализации процедур восстановления после критического сбоя или более точной обработки ошибок, по сравнению с простым отказом по тайм-ауту.

Другие возможности включают объединение нескольких хостов в пулы адресов, которые можно использовать как одну из форм балансировки нагрузки, а также поддержку публикации метрик и статистических данных на сервисах, таких как Prometheus и Graphite.

Consul

Consul – это решение задачи обнаружения сервисов, предлагаемое компанией Hashicorps (https://consul.io). Решение распределенное, предлагающее хранили-

ще данных типа «ключ-значение», но его главной отличительной особенностью являются развитые функциональные средства оперативной проверки и контроля работоспособности, а также DNS-сервер, устанавливаемый по умолчанию.

Теорема CAP (теорема Брюера)

При подробном изучении возможностей хранилищ типа «ключ-значение» и обнаружения сервисов вы сразу встретитесь с так называемой теоремой CAP, или теоремой Брюера (Eric Brewer) (https://en.wikipedia.org/wiki/CAP_theorem; https://ru.wikipedia.org/wiki/Теорема_CAP), – эвристическим утверждением о том, что в любой реализации распределенных вычислений возможно обеспечить не более двух из трех следующих свойств: согласованность данных (Consistensy), доступность (Availability), устойчивость к разделению (Partition tolerance)[1].

В системе AP (по первым буквам свойств на английском языке) предпочтение отдается доступности, а не согласованности данных, поэтому операции чтения и записи возможны практически всегда (и обычно они выполняются очень быстро), но результат не всегда актуален (в некоторых случаях могут возвращаться устаревшие данные). В системе CP согласованность данных более приоритетна, поэтому в некоторых случаях операции записи могут завершаться неудачно, но любые возвращаемые записи всегда корректны и актуальны.

На практике все не так просто и очевидно, в частности это касается использования Consul. И etcd, и Consul основаны на алгоритме Raft, представляющем решение CP. Но в Consul возможны три различных режима – default (по умолчанию), consistent (поддержка согласованности данных) и stale (разрешены устаревшие данные), – которые позволяют найти компромисс между уровнями согласованности данных и их доступности.

На каждом хосте работает экземпляр агента Consul в режиме сервера или в режиме клиента. Агент может проверять состояние различных сервисов, а также собирать общую статистику (использование памяти и т. п.), избавляя приложение-клиент от всех сложностей. Некоторая подгруппа хостов (обычно 3, 5 или 7 – см. примечание «Оптимальный размер кластера» выше) запускает агентов в режиме сервера. Эти экземпляры агентов отвечают за запись и хранение данных, а также за организацию совместной работы с другими агентами-серверами. Агенты, работающие в режиме клиента, перенаправляют запросы на агенты-серверы.

Практическое применение Consul реализуется достаточно просто, особенно при использовании контейнеров Docker. В нашем примере воспользуемся контейнером от компании GilderLabs (http://gilderlabs.com/). И в этот раз начнем с создания двух новых виртуальных машин для решения поставленной задачи:

```
$ docker-machine create -d virtualbox consul-1
...
$ docker-machine create -d virtualbox consul-2
...
```

[1] Чтобы узнать больше о теореме CAP и получить более точные определения этих терминов, обратитесь к документу «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services» (https://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf).

Далее запускаем контейнеры Consul. Снова сохраним их IP-адреса в переменных среды, чтобы сократить объем ручного ввода:

```
$ HOSTA=$(docker-machine ip consul-1)
$ HOSTB=$(docker-machine ip consul-2)
$ eval $(docker-machine env consul-1)
$ docker run -d --name consul -h consul-1
        -p 8300:8300 -p 8301:8301 -p 8301:8301/udp \
        -p 8302:8302 -p 8400:8400 -p 8500:8500 \
        -p 172.17.42.1:53:8600/udp \
        gliderlabs/consul agent -data-dir /data -server \  ❶
                  -client 0.0.0.0 \  ❷
                  -advertise $HOSTA -bootstrap-expect 2  ❸
...
ff226b3114541298d19a37b0751ca495d11fabdb652c3f19798f49db9cfea0dc
```

❶ Запуск агента Consul в режиме сервера и сохранение данных в каталоге */data*.

❷ Определение адреса для прослушивания, необходимого для API-запросов клиентов. По умолчанию это адрес `127.0.0.1`, к которому можно обращаться только изнутри данного контейнера.

❸ Флаг `-advertise` определяет адрес(а) других хостов, с которыми сервер должен установить связь. В данном примере это IP-адрес текущего хоста. Также устанавливается флаг `-bootstrap-expect`, сообщающий Consul о том, что необходимо дождаться, пока второй сервер не присоединится к кластеру.

Здесь для установления соединений между хостами использовались общедоступные IP-адреса, возвращаемые механизмом Docker Machine. При реальной эксплуатации, вероятнее всего, потребуется использование изолированных частных адресов, доступ к которым из внешней среды Интернета запрещен.

Запустим второй контейнер, для которого воспользуемся командой `-join`, чтобы установить соединение с первым сервером:

```
$ eval $(docker-machine env consul-2)
$ docker run -d --name consul -h consul-2 \
        -p 8300:8300 -p 8301:8301 -p 8301:8301/udp \
        -p 8302:8302 -p 8400:8400 -p 8500:8500 \
        -p 172.17.42.1:53:8600/udp \
        gliderlabs/consul agent -data-dir /data -server \
               -client 0.0.0.0 \
               -advertise $HOSTB -join $HOSTA
...
```

С помощью интерфейса командной строки можно проверить, действительно ли оба сервера были добавлены в кластер:

```
$ docker exec consul consul members
Node       Address              Status   Type     Build   Protocol   DC
consul-1   192.168.99.100:8301   alive    server   0.5.2   2          dc1
consul-2   192.168.99.100:8301   alive    server   0.5.2   2          dc1
```

Записывая и считывая некоторые данные, мы можем наблюдать, как работает хранилище ключ-значение:

```
$ curl -XPUT http://$HOSTA:8500/v1/kv/foo -d bar
true
$ curl http://$HOSTA:8500/v1/kv/foo | jq .
[
  {
    "Value": "YmFy",
    "Flags": 0,
    "Key": "foo",
    "LockIndex": 0,
    "ModifyIndex": 39,
    "CreateIndex": 16
  }
]
```

Мы записали данные, затем извлекли их, но что означает `"Value":"YmFy"`? Выясняется, что механизм Consul base64 выполняет перекодировку данных «на лету». Вернуть данные в исходном виде помогут утилиты `jq` и `base64`[1]:

```
$ curl -s http://$HOSTA:8500/v1/kv/foo | jq -r '.[].Value' | base64 -d
bar
```

Немного больше ручной работы, но результат получен.

Для добавления сервисов в Consul существует отдельный прикладной программный интерфейс, который связан с механизмом обнаружения сервисов Consul и механизмом проверки работоспособности. Вообще говоря, хранилище типа ключ-значение используется только для хранения подробностей конфигурации и небольшого объема метаданных.

Рассмотрим, как можно применить сервисы Consul для распределения работы idendidock по нескольким хостам. Общая структура останется неизменной: Redis работает на хосте `consul-2`, identidock и dnmonster работают на `consul-1`. Начнем с запуска Redis:

```
$ eval $(docker-machine env consul-2)
$ docker run -d -p 6379:6379 --name redis redis:3
...
2f79ea13628c446003ebe2ec4f20c550574c626b752b6ffa3b70770ad3e1ee6c
```

Теперь сообщим Consul о наличии сервиса Redis через точку входа */service/register*:

```
$ curl -XPUT http://$HOSTA:8500/v1/agent/service/register \
       -d '{"name":"redis", "address":"'$HOSTB'", "port":6379}'
$ docker run amouat/network-utils dig @172.17.42.1 +short redis.service.consul
...
192.168.99.101
```

---

[1] Здесь используется версия GNU Linux `base64`. Если вы работаете с версией Mac OS, то вместо аргумента `-d` применяйте аргумент `-D`.

Далее необходимо создать конфигурацию хоста consul-1, обеспечивающую использование Consul для поиска имен в системе DNS. В отличие от предыдущего примера с etcd, применим другой подход и не будем вписывать демон Docker в файл хоста */etc/resolv.conf*. Вместо этого необходимо отредактировать файл */var/lib/boot2docker/profile*, включив в него определения флагов --dns и --dns-search:

```
$ docker-machine ssh consul-1
...
docker@consul-1:~$ sudo vi /var/lib/boot2docker/profile
...
docker@consul-1:~$ cat /var/lib/boot2docker/profile

EXTRA_ARGS='
--label provider=virtualbox
--dns 172.17.42.1
--dns-search service-consul  ❶
'
CACERT=/var/lib/boot2docker/ca.pem
DOCKER_HOST='-H tcp://0.0.0.0:2376'
DOCKER_STORAGE=aufs
DOCKER_TLS=auto
SERVERKEY=/var/lib/boot2docker/server-key.pem
SERVERCERT=/var/lib/boot2docker/server.pem
```

❶ Этот аргумент позволяет использовать короткие имена, например redis вместо полного имени redis.service.consul.

После этого нужно перезапустить демон Docker и вернуть Consul в рабочий режим. По моему мнению, проще всего перезапустить виртуальную машину:

```
docker@consul-1:~$ exit
$ eval $(docker-machine env consul-1)
$ docker start consul
consul
```

Оперативное тестирование работоспособности:

```
$ docker run redis:3 redis-cli -h redis ping
PONG
```

Запуск dnmonster на consul-1 и добавление соответствующего сервиса:

```
$ docker run -d --name dnmonster amouat/dnmonster:1.0
...
41c8a78989803737f65460d75f8bed1a3683ee5a25c958382a1ca87f27034338
$ DNM_IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} dnmonster)
$ curl -XPUT http://$HOSTA:8500/v1/agent/service/register \
       -d '{"name":"dnmonster", "address":"'$DNM_IP'", "port":8080}'
```

Теперь можно запускать identidock:

```
$ docker run -d -p 80:9090 amouat/identidock:1.0
...
22cfd97bfba83dc31732886a4f0aec51e637b8c7834c9763e943e80225f990ec
$ curl $HOSTA
<html><head><title>...
```

И в этом примере identidock работает без явного установления каких-либо соединений.

Одной из наиболее интересных функциональных возможностей Consul является поддержка *механизма проверки работоспособности* (*health checking*), обеспечивающего уверенность в том, что различные компоненты системы существуют и нормально работают. Можно написать тесты для самого хоста (например, для контроля доступного дискового пространства или доступной оперативной памяти) или для конкретных сервисов. В следующем коде определяется простой HTTP-тест для сервиса dnmonster:

```
$ curl -XPUT http://$HOSTA:8500/v1/agent/service/register \
      -d '{"name":"dnmonster", "address":"'$DNM_IP'", "port":8080,
           "check": {"http": "http://'$DNM_IP':8080/monster/foo", "interval":"10s"}
          }'
```

Этот тест должен подтвердить, что данный контейнер правильно отвечает на HTTP-запрос к заданному URL и возвращает код состояния 2xx. Следует отметить, что эта проверка должна выполняться на хосте `consul-1`, для того чтобы тест прошел успешно. Состояние проведенного теста можно проверить, обратившись к точке входа */health/checks/dnmonster*:

```
$ curl -s $HOSTA:8500/v1//health/checks/dnmonster | jq '.[].Status'
"passing"
```

Для создания тестов также можно использовать скрипты командной оболочки, при этом тесты будут считаться успешно выполненными, если скрипт возвращает значение 0, что позволяет создавать процедуры проверок любой сложности. Объединяя процедуры проверок работоспособности с поддержкой в Consul механизма наблюдения за обновлениями данных *watches* (https://www.consul.io/docs/agent/watches.html), можно относительно просто реализовать устойчивые к критическим сбоям решения и/или систему автоматического уведомления администраторов о возникающих проблемах.

Кроме того, заслуживают внимания такие функциональные возможности, как поддержка распределенных центров данных и шифрование сетевого трафика.

Регистрация

В предыдущих примерах вручную выполнялся заключительный этап процедуры обнаружения сервисов – *регистрация* (*registration*): приходилось писать специальный curl-запрос для регистрации сервисов Redis и dnmonster и в SkyDNS, и в Con-

sul. Вместо этого можно было бы изменить конфигурацию Redis или dnmonster так, чтобы регистрация автоматически выполнялась при запуске[1], но существует также возможность написать специализированный сервис, отслеживающий события Docker и автоматически регистрирующий контейнеры при их запуске.

Эту задачу решает Registrator (https://github.com/gliderlabs/registrator), созданный компанией GliderLabs. Registrator может использоваться вместе с Consul, etcd или SkyDNS, обеспечивая автоматическую регистрацию контейнеров. Его работа основана на отслеживании в потоке событий Docker процедур создания контейнеров и добавлении соответствующих регистрационных записей в программную среду более низкого уровня на основе метаданных контейнеров.

Достоинства и недостатки обнаружения сервисов на основе DNS

Многие из описанных выше решений предоставляют DNS-интерфейс для обнаружения сервисов. В некоторых случаях это основной или единственный интерфейс, в других случаях это удобное дополнение к прочим прикладным программным интерфейсам.

Выбор DNS для обнаружения сервисов можно обосновать следующими важными достоинствами:

- DNS обеспечивает поддержку старых, давно написанных приложений без каких-либо дополнительных усилий. При использовании других механизмов обнаружения сервисов эта проблема решается добавлением контейнеров-посредников, но требует дополнительных трудозатрат как на разработку, так и на эксплуатацию;
- от разработчиков не требуется каких-то особых действий или изучения нового API. Приложения, использующие DNS, будут работать без изменений на широком спектре платформ;
- DNS – это общеизвестный и проверенный многолетней практикой протокол с многочисленными реализациями и широкой поддержкой.

Но обнаружение сервисов на основе DNS обладает и некоторыми недостатками, из-за которых в некоторых случаях приходится рассматривать возможность перехода на другие механизмы. Мы не будем принимать во внимание критические высказывания о том, что DNS работает медленно, – в данном контексте обнаружения сервисов мы говорим о быстрых, локальных процедурах поиска в противоположность медленным, удаленным. К прочим недостаткам можно отнести следующие:

- обычные процедуры DNS-поиска не возвращают информацию о порте, поэтому приходится ограничиваться предположениями или выполнять поиск по другому каналу (записи DNS SRV возвращают информацию о порте, но приложения и программные среды практически постоянно используют только имя хоста);
- приложения (и операционные системы) также могут кэшировать ответы DNS, что приводит к задержкам обновления информации для клиентов при перемещении сервисов;
- большинство DNS-сервисов предоставляет лишь ограниченную поддержку механизмов проверки работоспособности и балансировки нагрузки. Обычно для ба-

[1] Если это нежелательно или невозможно, то измените сам сервис внутри контейнера. Это можно сделать с помощью скрипта-обертки или вспомогательного процесса.

лансировки нагрузки выбор ограничен только методом Round-robin DNS или случайным распределением, что устраивает далеко не всех пользователей. Проверку работоспособности можно организовать на основе TTL, но для более изощренных и глубоких процедур проверки обычно требуются дополнительные сервисы;
- клиенты не ограничены в реализации логики при выборе между сервисами, основанными на таких атрибутах, как доступные версии API и производительность (мощность).

Другие решения

Существуют и некоторые другие варианты выбора решения задачи обнаружения сервисов, на которые следует обратить внимание:

- *ZooKeeper* (https://zookeeper.apache.org/) – централизованное, надежное, обеспечивающее высокую степень доступности хранилище, используемое для координации работы сервисов в Mesos и Hadoop. Решение реализовано на языке Java, доступ осуществляется через Java API, хотя доступны интерфейсы к некоторым другим языкам программирования. От клиентов требуются поддержка активных соединений с серверами ZooKeeper и поддержка механизма keepalive, то есть необходимо написание значительного объема кода (отметим, что при написании кода могут оказаться полезными библиотеки, подобные Curator (https://curator.apache.org/)). ZooKeeper – зрелое, стабильное и тщательно протестированное решение. Если у вас уже есть инфраструктура, использующая ZooKeeper, то, возможно, это будет лучшим вариантом. В противном случае дополнительные трудозатраты по интеграции и созданию инфраструктуры на основе ZooKeeper, вероятнее всего, не оправдаются, особенно если вы не используете язык Java;
- *SmartStack* (http://nerds.airbnb.com/smartstack-service-discovery-cloud/) – решение по обнаружению сервисов от компании Airbnb. Состоит из двух компонентов: Nerve для проверки работоспособности и регистрации и Synapse для обнаружения сервисов. Synapse работает на каждом хосте, который пользуется сервисами, и назначает для каждого сервиса порт, выполняющий роль прокси для доступа к реальному сервису. Synapse использует HAProxy (http://www.haproxy.org/) для маршрутизации и автоматически обновляет и перезапускает HAProxy после внесения изменений. Можно определить конфигурацию для получения списка сервисов, доступных через прокси, из хранилища (например, ZooKeeper или etcd) или путем отслеживания в потоке событий Docker процедур создания контейнеров (так же как в Registrator). Для каждого сервиса существует соответствующий процесс или контейнер Nerve, контролирующий работоспособность сервиса и автоматически регистрирующий новые сервисы с предоставлением хранилища, используемого Synapse (например, ZooKeeper или etcd);
- *Eureka* (https:github.com/Netflix/eureka/wiki) – решение компании Netflix, обеспечивающее балансировку нагрузки и устойчивость к критическим сбоям в AWS. Решение среднего звена для сглаживания «эфемерности» AWS-узлов. Если вы собираетесь организовать крупный сервис на основе инфраструктуры AWS, то это решение несомненно заслуживает более глубокого изучения;

- *WeaveDNS* (http://docs.weave.works/weave/latest_release/weavedns.html) – компонент, обеспечивающий обнаружение сервисов в комплексном сетевом решении Weave. Контейнеры регистрируют свой хост или имя с помощью WeaveDNS при запуске, эта процедура полностью автоматизирована. WeaveDNS функционирует как часть маршрутизатора Weave на каждом хосте и обменивается данными с другими маршрутизаторами Weave в сети, поэтому все имена контейнеров могут быть правильно определены. Кроме того, WeaveDNS поддерживает простую форму балансировки нагрузки. Более подробную информацию можно получить в разделе «Weave» текущей главы;
- *docker-discover* (http://jasonwilder.com/blog/2014/07/15/docker-service-discovery/) – по существу, это собственная Docker-реализация решения SmartStack, использующая etcd в качестве внутреннего элемента системы. Как и SmartStack, состоит из двух компонентов: docker-register (https://github.com/jwilder/docker-register), аналог Nerve для проверки работоспособности и регистрации и docker-discover (https://github.com/jwilder/docker-discover), аналог Synapse для обнаружения сервисов. Как и SmartStack, docker-discover использует HAProxy для маршрутизации. Чрезвычайно интересный проект, но малое количество обновлений и недостаточная организационная поддержка, вероятнее всего, говорят о крайне нерегулярной разработке и сопровождении.

В заключение важно отметить, что новые сетевые функциональные возможности Docker (см. раздел «Новая сетевая среда Docker») также предоставляют ограниченную форму обнаружения сервисов через сервис-объект. Этот механизм обнаружения сервисов основан на работе с сетевым драйвером Docker Overlay или с любым другим совместимым подключаемым модулем, например Calico.

Варианты организации сетевой среды

Как мы видели ранее, и контейнеры-посредники, и решения по обнаружению сервисов могут использоваться для установления соединений между сервисами, размещенными на разных хостах, если это позволяет нижележащий уровень сетевой среды. Но при этом необходимо явно объявить порты на хосте, что требует ручного управления и снижает возможность масштабирования. Более удачным решением является организация IP-соединения между контейнерами – тема нескольких следующих разделов данной главы.

Но прежде чем мы начнем рассматривать полноценные сетевые решения с соединением многих хостов, следует более подробно изучить работу сетевой среды Docker, предоставляемой по умолчанию, и ее функциональные возможности. Доступными являются четыре основных режима: *bridge* (мост), *host* (хост), *container* (контейнер) и *none* (нет сети).

Режим bridge

Предоставляемый по умолчанию сетевой режим bridge (мост) удобен для разработки, так как предоставляет простой способ организации обмена данными между

контейнерами. В реальной эксплуатации этот режим не так хорош – вся ручная работа, необходимая «за кулисами», требует значительных трудозатрат.

На рис. 11.5 показана схема работы сетевого шлюза Docker, которому обычно присваивается имя `docker0`. Шлюз размещается по адресу `172.17.42.1`, используемому для установления соединений между контейнерами. При запуске контейнера Docker создает новый экземпляр пары veth (фактически программный аналог кабеля Ethernet), соединяющий интерфейс `eth0` в контейнере со шлюзом. Внешние соединения обслуживаются при помощи механизма перенаправления IP forwarding и правил сетевого экрана iptables, которые определяют конфигурацию механизма преобразования IP-адресов IP masquerading, одну из форм механизма NAT (Network Address Translation).

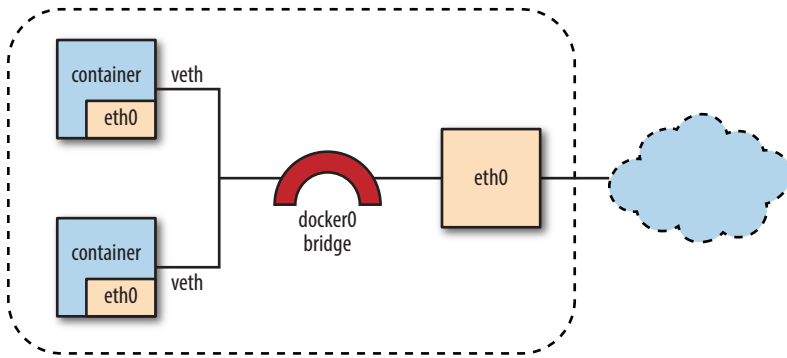

Рис. 11.5. Режим bridge сетевой среды Docker, предоставляемый по умолчанию

По умолчанию все контейнеры могут обмениваться данными друг с другом, независимо от того, установлены ли соединения между ними и объявлены ли открытые или экспортируемые порты (пример см. в разделе «Limit Container Networking» главы 13). Свободный обмен данными можно отменить, отключив эту возможность при запуске демона Docker с помощью флага `--icc=false`, который устанавливает соответствующее правило сетевого экрана iptables. Значения флагов `--icc=false` и `--iptables=true` позволяют обмениваться данными только контейнерам с установленными соединениями, и это тоже реализуется добавлением соответствующих правил iptables.

При разработке все это удобно и полезно, но проблемы производительности делают этот режим не вполне подходящим для эксплуатации.

Режим host

Контейнер, запущенный с флагом `--net=host`, использует пространство сетевых имен хоста, становясь полностью открытым для внешней сети. Это означает, что такой контейнер использует IP-адрес хоста совместно с другими контейнерами, но при этом исключается необходимость ручной работы, требуемой в режиме

bridge, таким образом, сеть работает так же быстро, как и обычная сеть, состоящая из хостов.

Поскольку IP-адрес используется совместно, контейнеры, обменивающиеся данными друг с другом, должны правильно распределить между собой порты на хосте, поэтому необходимо тщательно продумать и, возможно, изменить структуру приложений.

Также возникают проблемы обеспечения безопасности, связанные со случайным открытием доступа к портам из внешней сети. Эти проблемы могут быть устранены на уровне сетевого экрана.

Существенное улучшение производительности позволяет рассмотреть возможность реализации гибридной сетевой модели, в которой внешние контейнеры с большой сетевой нагрузкой, такие как прокси и кэши, используют сетевой режим host, а остальные контейнеры работают во внутренней сети в режиме bridge. Отметим невозможность установления прямых соединений между контейнерами, работающими в режиме bridge, с контейнерами, работающими в режиме host, но контейнеры в bridge-сети могут обмениваться данными с контейнерами host-сети, используя для этого IP-адрес шлюза `docker0`.

Режим container

В этом режиме используется пространство сетевых имен из другого контейнера. В некоторых ситуациях это очень удобно (например, при запуске контейнера с предварительно сконфигурированным сетевым стеком необходимо, чтобы все прочие контейнеры также использовали этот стек). Это позволяет создавать и многократно использовать эффективные конфигурации сетевых стеков, специализированных для конкретных вариантов использования или для архитектур центров данных. Недостаток заключается в том, что все контейнеры, совместно использующие сетевой стек, вынуждены пользоваться одним IP-адресом и т. д.

В определенных случаях этот режим вполне приемлем, и следует отметить его использование в Kubernetes (см. раздел «Kubernetes» главы 12).

Режим none

В этом режиме сетевая среда становится полностью недоступной для контейнера. Это может оказаться удобным для контейнеров, которым сеть вообще не нужна, например для контейнера-компилятора, записывающего весь свой поток вывода на том.

Режим отключения сетевой среды также может быть полезен в ситуациях, когда необходимо настроить собственную сетевую среду с нуля. В этом случае можно воспользоваться инструментальными средствами, такими как pipework (https://github.com/jpetazzo/pipework), обеспечивающими удобство при работе с сетями внутри cgroups и пространств имен.

Новая сетевая среда Docker

Во время написания книги сетевой стек Docker и соответствующий интерфейс находился на этапе генеральных контрольных испытаний перед вводом в эксплуатацию. С большой вероятностью этот этап будет завершен к моменту выхода книги из печати, и пользователям будет предоставлен широкий выбор возможностей для создания и практического применения сетей Docker (хотя все примеры кода из этой книги должны работать без каких-либо изменений). В этом разделе рассматриваются нововведения из экспериментальных предварительных версий Docker. Поскольку работа еще не завершена, возможны небольшие расхождения между представленной здесь информацией и окончательной стабильной версией Docker.

В первую очередь надо отметить самое главное изменение – появление в Docker двух новых «объектов» верхнего уровня: *network* (сеть) и *service* (сервис). Это позволяет создавать «сети»[1] и управлять ими отдельно от контейнеров. При запуске контейнеров можно присваивать им членство в определенной сети, после чего они получают возможность устанавливать прямые соединения с другими контейнерами в этой сети. Контейнеры могут объявлять открытыми сервисы, что позволяет им связываться по имени вместо необходимости установления формальных внутренних соединений (links) (которые продолжают использоваться, но становятся менее полезными).

Все сказанное выше лучше всего продемонстрировать на нескольких примерах.

Подкоманда `network ls` выводит список текущих активных сетей и их идентификаторов:

```
$ docker network ls
NETWORK ID     NAME     TYPE
d57af6043446   none     null
8fcc0afef384   host     host
30fa18d442d5   bridge   bridge
```

При запуске контейнера можно создать сервис с помощью флага `--publish-service`. Следующая команда создает сервис `db` в bridge-сети:

```
$ docker run -d --name redis1 --publish-service db.bridge redis
9567dd9eb4fbd9f588a846819ec1ea9b71dc7b6cbd73ac7e90dc0d75a00b6f65
```

Существующие сервисы можно просмотреть с помощью подкоманды `service ls`:

```
$ docker service ls
SERVICE ID     NAME   NETWORK   CONTAINER
f87430d2f240   db     bridge    9567dd9eb4fb
```

---

[1] Использование здесь термина «сеть» является не вполне правильным. В действительности «сеть» Docker во многих отношениях представляет собой пространство имен для контейнеров, которое позволяет объединять и разделять их, а также создавать каналы обмена данными.

Теперь можно создать новый контейнер в той же сети и организовать обмен данными с контейнером redis1 без необходимости установления каких-либо соединений, используя для этого сервис db:

```
$ docker run -it redis redis-cli -h db ping
PONG
```

По умолчанию контейнеры, принадлежащие другим сетям, не имеют возможности обмениваться данными с контейнерами этой сети.

Если снова воспользоваться Docker-соединениями, то можно увидеть, что в действительности они служат для настройки сервисов, но эта процедура скрыта от пользователя:

```
$ docker run -d --name redis2 redis
7fd526b2c7a6ad8a3faf4be9c1c23375dc5ae4cd17ff863a293c67df816a2b09
$ docker run --link redis2:redis2 redis redis-cli -h redis2 ping
PONG
$ docker service ls
SERVICE ID     NAME     NETWORK   CONTAINER
59b749c7fe0b   redis2   bridge    7fd526b2c7a6
f87430d2f240   db       bridge    9567dd9eb4fb
```

Еще более интересно то, что можно переназначать контейнеры, предоставляющие сервисы, с помощью подкоманд service attach и service detach:

```
$ docker run redis redis-cli -h db set foo bar  ❶
OK
$ docker run redis redis-cli -h redis2 set foo baz  ❷
OK
$ docker run redis redis-cli -h db get foo
bar
$ docker service detach redis1 db
$ docker service attach redis2 db
$ docker run redis redis-cli -h db get foo  ❸
baz
```

❶ Добавление данных в контейнер redis1. В текущий момент объявлено использование этого контейнера для сервиса db.
❷ Добавление других данных в контейнер redis2.
❸ Теперь объявлено использование контейнера redis2 для сервиса db.

Таким образом, мы видим, что новая модель предлагает намного более гибкие и эффективные средства связи между контейнерами и инструменты сопровождения систем, продолжая при этом поддерживать основы существующей сетевой среды.

Типы сетей и подключаемые модули

Вероятно, вы уже заметили, что в Docker имеются различные *типы*[1] сетей. Выше мы уже рассмотрели типы «классических» вариантов сетевой среды – host, none

[1] Их также называют *сетевыми драйверами* (*drivers*).

и bridge, – но, кроме них, имеется еще тип *overlay*, а также возможность добавления новых типов в форме *подключаемых сетевых модулей* (*networking plugins*). Тип сети, используемый по умолчанию, можно установить при запуске демона Docker. Если тип не указан явно, то по умолчанию используется bridge-сеть.

Подключаемые модули

Подключаемые модули, в том числе и сетевые драйверы[1], можно добавлять в Docker, устанавливая их в каталог */usr/share/docker/plugins*. Самый очевидный способ загрузки и установки – запуск контейнера, который монтирует этот каталог.

Подключаемые модули могут быть написаны на любом языке программирования, который способен взаимодействовать с JSON-RPC API (http://www.jsonrpc.org/), используемым в Docker.

Мы рассмотрим пример использования сетевого подключаемого модуля Project Calico в разделе «Project Calico» текущей главы. Я предполагаю появление широкого спектра подключаемых модулей, предназначенных для различных ситуаций и использующих разнообразные внутренние технологии (такие как IPVLAN (https://github.com/torvalds/linux/blob/master/Documentation/networking/ipvlan.txt) и Open vSwitch (http://openvswitch.org/)).

Комплексные сетевые решения

В этом разделе описаны различные решения, которые можно применить для организации сетевых кластеров из контейнеров, расположенных на разных хостах. Рассматривается *Overlay*, решение Docker «с включенным комплектом батареек», которое будет предоставляться вместе с новым сетевым стеком, *Weave*, решение с богатыми функциональными возможностями, в котором главное внимание сконцентрировано на простоте использования, *Flannel*, решение на основе CoreOS и *Project Calico*, решение на основе layer-3-коммутации от компании Metaswitch.

Сетевая среда Docker появилась совсем недавно и очень быстро развивается и изменяется. Она служит основой для весьма широкого поля деятельности с потенциальными возможностями поддержки множества разнообразных решений, предназначенных для различных ситуаций. Несмотря на то что представленные здесь инструменты уже достигли определенного уровня технического развития во время написания книги, эта область изменяется стремительно, и к моменту выхода книги из печати изменятся способы реализации предложенных здесь решений, появятся новые решения. Поэтому любое решение, описанное ниже, следует критически изучить перед практическим применением.

---

[1] Новые драйверы томов также можно загружать через подключаемые модули, такие как flocker (https://github.com/ClusterHQ/flocker-docker-plugin). Во время написания книги продолжалась разработка нового объекта верхнего уровня volume, но она пока еще далека от завершения.

Overlay

Overlay – это реализация решения Docker «с включенным комплектом батареек» для организации сетевой среды, распределенной по многим хостам. Решение использует туннели VXLAN для соединений хостов в их внутреннем пространстве IP-адресов и NAT для установления внешних соединений. Библиотека serf (https://serfdom.io/) используется для обмена данными между узлами.

Объединение контейнеров в сеть выполняется практически тем же способом, как и для стандартной сети со шлюзами, – устанавливается шлюз Linux для сети Overlay, затем пара виртуальных интерфейсов veth используется для установления соединения с контейнером.

Осторожно – экспериментальная версия

Виртуальные машины в приведенном ниже примере запускаются под управлением экспериментальной версии Docker, поэтому возможны отличия от вашей версии. К моменту выхода книги из печати стабильная версия Docker обеспечит поддержку сетевых подключаемых модулей, которые следует использовать вместо предлагаемых здесь вариантов.

В примере используются следующие версии Docker и Consul:

```
docker@overlay-1:~$ docker --version
Docker version 1.8.0-dev, build 5fdc102, experimental
docker@overlay-1:~$ docker run gliderlabs/consul version
Consul v0.5.2
Consul Protocol: 2 (Understands back to: 1)
```

В качестве примера используются два хоста `overlay-1` и `overlay-2` с установленной экспериментальной версией Docker и Consul в качестве хранилища данных типа ключ-значение. Приложение identidock устанавливается и настраивается так же, как раньше, контейнер Redis запускается на хосте `overlay-2`, контейнеры dnmonster и identidock запускаются на хосте `overlay-1`.

К моменту выхода книги из печати должна появиться возможность сделать то же самое или почти то же самое в стабильной версии. Здесь я напрямую обращаюсь к виртуальной машине с помощью утилиты `ssh`, чтобы удостовериться в использовании одинаковых версий клиента и демона Docker.

Начинаем с обращения к `overlay-2` через `ssh`:

```
$ docker-machine ssh overlay-2
...
```

Сначала создадим новую сеть с именем `ovn`, используя для этого драйвер `overlay`:

```
docker@overlay-2:~$ docker network create -d overlay ovn
5d2709e8fd689cb4dee6acf7a1346fb563924909b4568831892dcc67e9359de6
docker@overlay-2:~$ docker network ls
NETWORK ID     NAME     TYPE
f7ae80f9aa44   none     null
1d4c071e42b1   host     host
27c18499f9e5   bridge   bridge
5d2709e8fd68   ovn      overlay
```

Более подробную информацию о новой сети можно получить с помощью подкоманды network info:

```
docker@overlay-2:~$ docker network info ovn
Network Id: 5d2709e8fd689cb4dee6acf7a1346fb563924909b4568831892dcc67e9359de6
Name: ovn
Type: overlay
```

Теперь можно запустить Redis, используя аргумент --publish-service redis.ovn для объявления данного контейнера как сервиса с именем redis в сети ovn:

```
docker@overlay-2:~$ docker run -d --name redis-ov2 \
                            --publish-service redis.ovn redis:3
...
29a02f672a359c5a9174713418df50c72e348b2814e88d537bd2ab877150a4a5
```

Если сейчас выйти из контейнера overlay-2 и установить соединение через ssh с контейнером overlay-1, то можно видеть, что доступ предоставлен в ту же самую сеть:

```
docker@overlay-2:~$ exit
$ docker-machine ssh overlay-1
docker@overlay-2:~$ docker network ls
NETWORK ID     NAME     TYPE
7f9a4f144131   none     null
528f9267a171   host     host
dfec33441302   bridge   bridge
5d2709e8fd68   ovn      overlay
```

Контейнеры dnmonster и identidock запускаются тем же способом, с аргументом --publish-service для установления соединения с сетью ovn:

```
docker@overlay-1:~$ docker run -d --name dnmonster-ov1 \
                        --publish-service dnmonster.ovn amouat/dnmonster:1.0
...
37e7406613f3cbef0ca83320cf3d99aa4078a9b24b092f1270352ff0e1bf8f92
docker@overlay-1:~$ docker run -d --name identidock-ov1 \
                        --publish-service identidock.ovn amouat/identidock:1.0
...
41f328a59ff3644718b8ce4f171b3a246c188cf80a6d0aa96b397500be33da5e
```

Проверяем работу приложения:

```
docker@overlay-1:~$ docker exec identidock-ov1 curl -s localhost:9090
<html><head><title>Hello...
```

Работа приложения identidock с распределенными по двум хостам компонентами организована очень быстро и просто. Поскольку детали реализации и методика применения, вероятнее всего, изменятся, мы не будем углубляться в подробности внутренней работы драйвера Overlay.

Weave

Weave (http://weave.works/) представляет собой удобное для разработчиков сетевое решение, предназначенное для работы с широким диапазоном программных сред с минимальными трудозатратами. Возможно, Weave является самым полным решением из всех доступных сегодня, так как включает WeaveDNS для обнаружения сервисов и балансировки нагрузки, встроенный *механизм управления IP-адресами* (IPAM) и обеспечивает поддержку шифрования каналов обмена данными.

Рассмотрим, насколько просто организовать с помощью Weave работу приложения identidock с размещением компонентов на двух хостах. Архитектура остается той же, что и в предыдущих примерах с применением посредника и механизма обнаружения сервисов, – Redis работает на одном хосте (в нашем случае `weave-redis`), контейнеры identidock и dnmonster – на другом (`weave-identidock`). Снова воспользуемся механизмом Docker Machine для создания виртуальных машин. В примере используется Weave версии 1.1.0, в более новых версиях возможны небольшие изменения.

Начнем с создания `weave-redis`:

```
$ docker-machine create -d virtualbox weave-redis
...
```

Теперь воспользуемся утилитой `ssh` и установим Weave:

```
$ docker-machine ssh weave-redis
...
docker@weave-redis:~$ sudo curl -sL git.io/weave -o /usr/local/bin/weave
docker@weave-redis:~$ sudo chmod a+x /usr/local/bin/weave
docker@weave-redis:~$ weave launch
Setting docker0 MAC (mitigate https://github.com/docker/docker/issues/14908)
Unable to find image 'weaveworks/weaveexec:v1.1.0' locally
v1.1.0: Pulling from weaveworks/weaveexec
...
Digest: sha256:8b5e1b692b7c2cb9bff6f9ce87360eee88540fe32d0154b27584bc45acbbef0a
Status: Downloaded newer image for weaveworks/weaveexec:v1.1.0
Unable to find image 'weaveworks/weave:v1.1.0' locally
v1.1.0: Pulling from weaveworks/weave
Digest: sha256:c34b8ee7b72631e4b7ddca3e1157b67dd866cae40418c279f427589dc944fac0
Status: Downloaded newer image for weaveworks/weave:v1.1.0
```

Приведенные выше команды загружают образ Weave, затем скачивают и запускают контейнеры, составляющие инфраструктуру Weave. Немного позже мы более подробно разберемся, что делает каждый из этих контейнеров.

Следующий шаг – перенаправление клиента Docker к прокси-серверу Weave вместо демона Docker. Это дает Weave возможность настраивать разнообразные нюансы сетевой среды во время запуска контейнеров:

```
docker@weave-redis:~$ eval $(weave env)
```

Теперь можно запустить контейнер Redis, и он автоматически установит соединение с сетью Weave:

```
docker@weave-redis:~$ docker run --name redis -d redis:3
Unable to find image 'redis:3' locally
3: Pulling from redis
...
3c97d635be5107f5a79cafe3cfaf1960fa3d14eec3ed5fa80e2045249601583f
docker@weave-redis:~$ exit
```

Создадим хост для identidock и dnmonster. В этот раз не будем регистрироваться на созданном хосте, а просто выполним ssh-команды через docker-machine, так как это немного упростит процесс конфигурирования. Сначала создается виртуальная машина weave-identidock и устанавливается Weave:

```
$ docker-machine create -d virtualbox weave-identidock
...
$ docker-machine ssh weave-identidock \
   "sudo curl -sL https://git.io/weave -o /usr/local/bin/weave && \
   sudo chmod a+x /usr/local/bin/weave"
```

В этом случае при запуске weave launch необходимо передать IP-адрес хоста weave-redis:

```
$ docker-machine ssh weave-identidock "weave launch $(docker-machine ip weave-redis)"
Unable to find image 'weaveworks/weaveexec:v1.1.0' locally
v1.1.0: Pulling from weaveworks/weaveexec
...
Digest: sha256:8b5e1b692b7c2cb9bff6f9ce87360eee88540fe32d0154b27584bc45acbbef0a
Status: Downloaded newer image for weaveworks/weaveexec:v1.1.0
Unable to find image 'weaveworks/weave:v1.1.0' locally
v1.1.0: Pulling from weaveworks/weave
Digest: sha256:c34b8ee7b72631e4b7ddca3e1157b67dd866cae40418c279f427589dc944fac0
Status: Downloaded newer image for weaveworks/weave:v1.1.0
```

Необходимо проверить работоспособность созданной сетевой среды. С помощью утилиты ssh зарегистрируемся на weave-identidock и попытаемся получить доступ к контейнеру Redis, работающему на хосте weave-redis. И в этом случае необходимо настроить Weave-прокси:

```
$ docker-machine ssh weave-identidock
...
docker@weave-identidock:~$ eval $(weave env)
docker@weave-identidock:~$ docker run redis:3 redis-cli -h redis ping
...
PONG
```

Сеть работает. Для завершения выполнения примера запустим контейнеры dnmonster и identidock и убедимся, что приложение работает правильно:

```
docker@weave-identidock:~$ docker run --name dnmonster -d amouat/dnmonster:1.0
...
1bc9cdd5c3dd532d4f6a56529be8e2a068a9402c1e07df69ec33971f5c4b89b9
docker@weave-identidock:~$ docker run --name identidock -d -p 80:9090 \
    amouat/identidock:1.0
9b5e9c89a7807bcad2cff49dc0692d0e8d064494288df5405a6573d886c0208d
docker@weave-identidock:~$ exit
$ curl $(docker-machine ip weave-identidock)
<html><head>...
$ curl -s $(docker-machine ip weave-identidock)/monster/gordon | head -c 4
PNG
```

Итак, мы получили сетевую среду с распределенным по нескольким (двум) хостам приложением, поддерживающую DNS-разрешение имен контейнеров при внешнем отсутствии каких-либо явно установленных связей.

Чтобы лучше понять, как это работает, следует более внимательно изучить контейнеры Weave, запускаемые для управления инфраструктурой:

```
$ docker ps
CONTAINER ID  ...  PORTS                                              NAMES
0b7693194bb9                                                          weaveproxy
b6e515f4d02b       172.17.42.1:53->53/udp, 0.0.0.0:6783->6783/t...    weave
```

Эти контейнеры и контейнеры приложения identidock показаны на рис. 11.6.

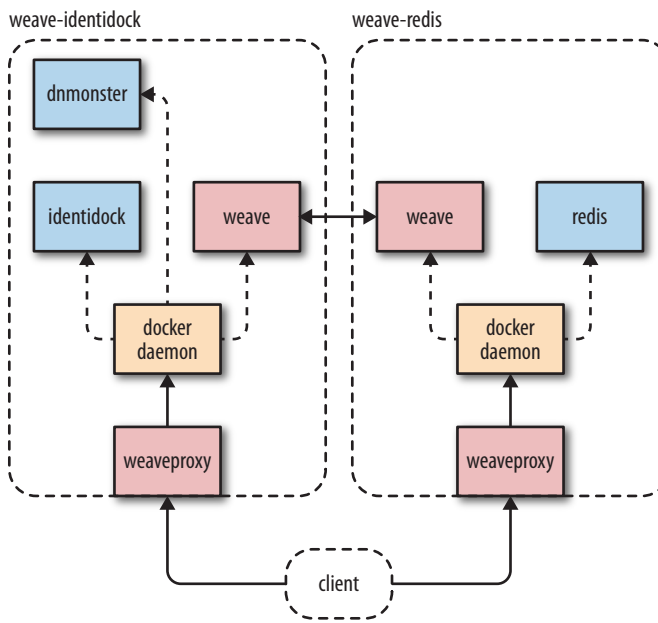

Рис. 11.6. Приложение identidock, работающее в сетевой среде Weave

На обоих хостах работают одинаковые образы Weaves, предварительно запущенные командами `weave launch`. Эти контейнеры поддерживают функционирование различных компонентов инфраструктуры Weave:

- *weave* – этот контейнер содержит маршрутизатор Weave, отвечающий за формирование сетевых маршрутов и обмен данными с другими хостами в сетевой среде Weave. Маршрутизаторы Weave обеспечивают связь с другими узлами по протоколу TCP для организации обмена данными и совместного использования информации в конкретной сетевой конфигурации (топологии). Сетевой трафик через UDP-соединения настраивается отдельно. Маршрутизаторы Weave постепенно «изучают» топологию сети, что позволяет им эффективно реализовать маршрутизацию и работать с изменяющимися конфигурациями неполносвязных сетей. Кроме того, маршрутизатор обрабатывает DNS-запросы, позволяя разработчикам обращаться по именам к контейнерам, расположенным на разных хостах;
- *weaveproxy* – этот контейнер отвечает за все скрытые от пользователя операции, которые позволяют выполнять обычные команды `docker run` в сетевой среде Weave. Контейнер перехватывает запросы `docker run` к демону Docker, позволяя Weave соответствующим образом настроить сетевую среду и изменить выполнение запроса таким образом, чтобы контейнер использовал сетевой стек Weave. После этого измененный запрос перенаправляется в демон Docker. Методика такого перехвата реализуется командной строкой `eval $(weave proxy-env)`, устанавливающей значение переменной среды `DOCKER_HOST`, которая указывает на прокси Weave, а не на реальный демон Docker.

Weave создает шлюз weave, который можно увидеть, выполнив команду `ifconfig`. Каждый контейнер, содержащий маршрутизатор weave, устанавливает соединение с этим шлюзом через пару интерфейсов veth.

Weave предоставляет возможность размещения контейнеров в различных подсетях, что позволяет отделить приложения друг от друга, а также поддерживает шифрование трафика, поэтому в сетях Weave могут использоваться ненадежные каналы связи.

Полную информацию об архитектуре и функциональных возможностях Weave можно получить из официальной документации (http://docs.weave.works/weave/latest_release/index.html).

Главная цель Weave – создание удобной среды для разработчиков, в которой контейнеры можно распределять по сетевым хостам и находить без малейших затруднений.

Работа Weave в качестве подключаемого модуля

Weave можно также запустить с помощью рабочей среды подключаемых модулей Docker, тем самым эффективно заменяя прокси-контейнер.

Во время написания книги существовали некоторые ограничения в применении этой методики, поэтому следует рассчитывать в основном на скорейшее завершение разработки прикладного программного интерфейса сетевых под-

ключаемых модулей, чтобы предоставить Weave и прочим сетевым модулям всю необходимую информацию. Например, в настоящее время одна из проблем заключается в том, что Weave теряет информацию о конфигурации при перезапуске кластера.
Сейчас, когда вы читаете эти строки, возможно, эти проблемы уже успешно решены.

Flannel

Flannel (https://github.com/coreos/flannel) – сетевое решение с распределением контейнеров по разным хостам на основе CoreOS. В основном используются кластеры на базе CoreOS, но нет никаких ограничений на использование других сетевых стеков.

Flannel назначает подсеть для каждого хоста и в дальнейшем присваивает соответствующие IP-адреса контейнерам. Flannel успешно работает вместе cKubernetes (см. раздел «Kubernetes» главы 12), которое может использоваться для присваивания уникальных и маршрутизируемых IP-адресов каждой группе контейнеров. Flannel запускает на каждом хосте демон, который считывает свою конфигурацию из etcd (см. раздел «etcd» текущей главы), поэтому для кластера обязательно должна быть предварительно создана конфигурация с использованием etcd. Доступны механизмы поддержки (backends):

- *udp* – принимаемый по умолчанию компонент, формирующий оверлейную сеть, в которой сетевая информация уровня 2 инкапсулируется в UDP-пакетах, пересылаемых через существующую сеть;
- *vxlan* – используется технология сетевой виртуализации VXLAN для инкапсуляции сетевых пакетов. Так как эта технология реализована в ядре, можно предположить, что компонент будет работать намного быстрее, чем UDP в пространстве пользователя;
- *aws-vpc* – для настройки сетей в среде Amazon EC2;
- *host-gw* – устанавливает IP-маршруты к подсетям, используя удаленные IP-адреса. Обязательно требуются сетевая связанность на уровне 2 между хостами;
- *gce* – для настройки сетей в среде Google Compute.

Для работы с Flannel мы снова воспользуемся механизмом Docker Machine, но на этот раз все будет немного сложнее – для демона Flannel необходимо создать конфигурацию сетевого шлюза `flannel0` перед началом работы механизма Docker, что затрудняет запуск Flannel как контейнера. Можно было бы организовать оригинальную процедуру загрузки, при которой Flannel загружает второй, вспомогательный демон Docker, но проще запустить flannel как обычный процесс на хосте. К тому же Flannel зависит от сервиса хранилища etcd, который также запускается как обычный процесс хоста.

Надо признать, что виртуальные машины, создаваемые с помощью Docker Machine, в действительности представляют собой не самый лучший вариант для данного примера использования Flannel из-за необходимости создания дополнительной конфигурации вручную, в противовес автоматизированным способам

конфигурирования. Тем не менее это полезный практический пример, который поможет лучше понять, как использовать Flannel в конкретной сетевой инфраструктуре.

Версии Flannel и etcd
В приведенных ниже примерах используются версия 2.0.13 etcd и версия 0.5.1 Flannel. Поскольку разработка Flannel ведется непрерывно и весьма активно, могут обнаружиться некоторые отличия, если вы воспользовались более новой версией.

В данном примере мы создаем и настраиваем два хоста `flannel-1` и `flannel-2` и проверяем возможность установления соединения между контейнерами на каждом хосте. Начнем с создания двух виртуальных машин, действующих в качестве этих хостов:

```
$ docker-machine create -d virtualbox flannel-1
...
$ docker-machine create -d virtualbox flannel-2
...
$ docker-machine ip flannel-1 flannel-2
192.168.99.102
192.168.99.103
```

Обратите внимание на IP-адреса каждой виртуальной машины. Они необходимы для настройки etcd, который будет устанавливаться на хост `flannel-1`. Но сначала нужно остановить демон Docker и удалить шлюз `docker0`:

```
$ docker-machine ssh flannel-1
...
docker@flannel-1:~$ sudo /usr/local/etc/init.d/docker stop
docker@flannel-1:~$ sudo ip link delete docker0
```

Загрузим и разархивируем etcd:

```
docker@flannel-1:~$ curl -sL https://github.com/coreos/etcd/releases/download/\
v2.0.13/etcd-v2.0.13-linux-amd64.tar.gz -o etcd.tar.gz
docker@flannel-1:~$ tar xzvf etcd.tar.gz
```

Теперь нужно запустить etcd. Для удобства формирования команд используем пару переменных среды:

```
docker@flannel-1:~$ HOSTA=192.168.99.102
docker@flannel-1:~$ HOSTB=192.168.99.103
docker@flannel-1:~$ nohup etcd-v2.0.13-linux-amd64/etcd \
  -name etcd-1 -initial-advertise-peer-urls http://$HOSTA:2380 \
  -listen-peer-urls http://$HOSTA:2380 \
  -listen-client-urls http://$HOSTA:2379,http://127.0.0.1:2379 \
  -advertise-client-urls http://$HOSTA:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster etcd-1=http://$HOSTA:2380,etcd-2=http://$HOSTB:2380 \
  -initial-cluster-state new &
```

Утилита nohup используется для того, чтобы процесс etcd продолжал работать после того, как мы завершим сеанс на этом хосте. Утилита будет записывать вывод в файл *nohup.out*.

Итак, установка etcd на хосте flannel-1 завершена. Позже мы вернемся и закончим установку Flannel, после того как настроим состояние хоста flannel-2:

```
docker@flannel-1:~$ exit
$ docker-machine ssh flannel-2
docker@flannel-2:~$ sudo /usr/local/etc/init.d/docker stop
docker@flannel-2:~$ sudo ip link delete docker0
docker@flannel-2:~$ curl -sL https://github.com/coreos/etcd/releases/\
download/v2.0.13/etcd-v2.0.13-linux-amd64.tar.gz -o etcd.tar.gz
docker@flannel-2:~$ tar xzvf etcd.tar.gz
```

Запуск etcd выполняется точно так же, только здесь IP-адреса меняются местами в команде установки:

```
docker@flannel-2:~$ HOSTA=192.168.99.102
docker@flannel-2:~$ HOSTB=192.168.99.103
docker@flannel-2:~$ nohup etcd-v2.0.13-linux-amd64/etcd \
  -name etcd-2 -initial-advertise-peer-urls http://$HOSTB:2380 \
  -listen-peer-urls http://$HOSTB:2380 \
  -listen-client-urls http://$HOSTB:2379,http://127.0.0.1:2379 \
  -advertise-client-urls http://$HOSTB:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster etcd-1=http://$HOSTA:2380,etcd-2=http://$HOSTB:2380 \
  -initial-cluster-state new &
```

Теперь можно загрузить Flannel:

```
docker@flannel-2:~$ curl -sL https://github.com/coreos/flannel/releases/\
download/v0.5.1/flannel-0.5.1-linux-amd64.tar.gz -o flannel.tar.gz
docker@flannel-2:~$ tar xzvf flannel.tar.gz
```

Далее необходимо создать определенную конфигурацию в etcd, чтобы сообщить Flannel, какой диапазон IP-адресов можно использовать:

```
docker@flannel-2:~$ ./etcd-v2.0.13-linux-amd64/etcdctl
                    set /coreos.com/network/config '{ "Network": "10.1.0.0/16" }'
```

После этого можно запустить демон Flannel. Отметим, что здесь обязательно нужно сообщить Flannel о необходимости использования интерфейса eth1, позволяющего обмениваться данными с другой виртуальной машиной:

```
docker@flannel-2:~$ nohup sudo ./flannel-0.5.1/flanneld -iface=eth1 &
```

Flannel работает, и если выполнить команду ifconfig, то вы должны увидеть шлюз Flannel:

```
docker@flannel-2:~$ ifconfig flannel0
flannel0   Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-...
           inet addr:10.1.37.0  P-t-P:10.1.37.0  Mask:255.255.0.0
           UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1472  Metric:1
```

```
RX packets:4 errors:0 dropped:0 overruns:0 frame:0
TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:216 (216.0 B)  TX bytes:221 (221.0 B)
```

Следует отметить, что IP-адрес шлюза взят из диапазона, заданного в конфигурации Flannel.

Теперь необходимо настроить Docker на использование Flannel. Если бы мы работали с другим образом виртуальной машины или непосредственно без использования виртуализации, то можно было бы воспользоваться скриптом `mk-docker-opts.sh`, распространяемым вместе с Flannel для автоматического конфигурирования механизма Docker. Но в наш образ VirtualBox командная оболочка bash не включена, поэтому все придется делать вручную. Сначала просмотрим содержимое файла */run/flannel/subnet.env*, созданного при установке Flannel:

```
docker@flannel-2:~$ cat /run/flannel/subnet.env
FLANNEL_SUBNET=10.1.79.1/24
FLANNEL_MTU=1472
FLANNEL_IPMASQ=false
```

Для демона Docker нужно установить значение аргумента `--bip` равным значению `FLANNEL_SUBNET`, а значение аргумента `--mtu` – равным значению `FLANNEL_MTU`, чтобы Docker использовал параметры IP-адреса и MTU[1], совместимые с Flannel. Аргументы демона Docker определяются в файле */var/lib/boot2docker/profile*. После редактирования (которое можно выполнить с помощью команды `sudo vi` в соответствующей виртуальной машине) этот файл должен выглядеть приблизительно так:

```
docker@flannel-2:~$ cat /var/lib/boot2docker/profile

EXTRA_ARGS='
--label provider=virtualbox
--bip 10.1.79.1/24
--mtu 1472
'
CACERT=/var/lib/boot2docker/ca.pem
DOCKER_HOST='-H tcp://0.0.0.0:2376'
DOCKER_STORAGE=aufs
DOCKER_TLS=auto
SERVERKEY=/var/lib/boot2docker/server-key.pem
SERVERCERT=/var/lib/boot2docker/server.pem
```

Теперь можно перезапустить механизм Docker:

```
docker@flannel-2:~$ sudo /etc/init.d/docker start
hostname: flannel-2: Unknown host
Need TLS certs for flannel-2,,10.0.2.15,192.168.99.103
docker@flannel-2:~$ exit
```

---

[1] MTU – Maximum Transmission Unit – максимальный размер полезного блока одного пакета, управляющий размером пакетов данных, передаваемых в данной сети.

На заключительном этапе необходимо повторить все описанные выше действия для хоста flannel-1:

```
$ docker-machine ssh flannel-1
...
docker@flannel-1:~$ curl -sL https://github.com/coreos/flannel/releases/\
download/v0.5.1/flannel-0.5.1-linux-amd64.tar.gz -o flannel.tar.gz
docker@flannel-1:~$ tar xzvf flannel.tar.gz
...
docker@flannel-1:~$ nohup sudo ./flannel-0.5.1/flanneld -iface=eth1 &
docker@flannel-1:~$ cat /run/flannel/subnet.env
FLANNEL_SUBNET=10.1.83.1/24  ❶
FLANNEL_MTU=1472
FLANNEL_IPMASQ=false
docker@flannel-1:~$ sudo vi /var/lib/boot2docker/profile
...
docker@flannel-1:~$ sudo /etc/init.d/docker start
hostname: flannel-1: Unknown host
Need TLS certs for flannel-1,,10.0.2.15,192.168.99.102
docker@flannel-1:~$ exit
```

❶ Следует отметить, что это значение должно отличаться от значения для хоста flannel-2, чтобы хосты назначали контейнерам IP-адреса из различных диапазонов.

Теперь все должно работать, так что проверим, могут ли созданные контейнеры обмениваться данными. На хосте flannel-1 запустим утилиту Netcat для отслеживания соединений через заданный порт:

```
$ eval $(docker-machine env flannel-1)
$ docker run --name nc-test -d amouat/network-utils nc -l 5001
...
```

Контейнер с сетевыми утилитами

Для тестирования сети при возникновении проблем удобно иметь образ, в котором установлен набор специализированных сетевых инструментальных средств. Я создал образ с именем `amouat/network-utils`, который можно использовать для этих целей. В нем вы найдете такие инструменты, как curl, Netcat, traceroute, dnsutils, а также утилиту jq для форматирования в удобочитаемом виде JSON-данных, выводимых REST API.

Пример использования:

```
$ docker run -it amouat/network-utils
root@e80c9731ea0:/# curl -s https://api.github.com/repos/amouat/network-utils-container \
     | jq '. .description'
"Docker container with some network utilities"
```

Теперь можно определить IP-адрес, присвоенный Flannel этому контейнеру:

```
$ IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} nc-test
$ echo $IP
10.1.83.2
```

Отметим, что адрес находится в диапазоне, который мы задали ранее. Запустим Netcat на хосте flannel-2 и протестируем соединение с контейнером nc-test:

```
$ eval $(docker-machine env flannel-2)
$ docker run -e IP=$IP \
    amouat/network-utils sh -c 'echo -n "hello" | nc -v $IP 5001'
Unable to find image 'amouat/network-utils:latest' locally
...
Status: Downloaded newer image for amouat/network-utils:latest
Connection to 10.1.83.2 5001 port [tcp/*] succeded!
```

В журнале контейнера nc-test можно увидеть посланное нами сообщение:

```
$ eval $(docker-machine env flannel-1)
$ docker logs nc-test
hello
```

Итак, что мы получили: два контейнера, обменивающихся данными и размещенных на различных хостах, использующих собственные IP-адреса. Возможно, для достижения этой цели объем ручной работы был слишком велик, но помните, что в реальных условиях часть этой работы может быть автоматизирована для новых хостов, присоединяемых к кластеру, а пользователи сетевых стеков CoreOS вообще могут рассчитывать на работу кластера прямо «из коробки».

Но здесь мы не сможем запустить identidock. Несмотря на наличие работающей сетевой среды с распределенными хостами, у нас нет поддержки обнаружения сервисов, для организации которой потребуется одно из ранее описанных решений, таких как SkyDNS, или же нужно будет переписать identidock для использования etcd.

Project Calico

Project Calico (в дальнейшем просто Calico) предлагает несколько другой подход к организации сетевой среды. Проще всего его описать в терминах модели OSI (https://en.wikipedia.org/wiki/OSI_model, https://ru.wikipedia.org/wiki/Сетевая_модель_OSI), в которой сетевая среда разделена на семь концептуальных уровней. В большинстве других сетевых решений, таких как Weave и Flannel (при использовании UDP в качестве внутреннего компонента), создается оверлейная сеть посредством инкапсуляции трафика уровня 2[1]. Вместо этого в Calico применяет стандартную IP-маршрутизацию и стандартные сетевые инструментальные средства для реализации решения на уровне 3[2].

Главными преимуществами решения, реализованного исключительно на уровне 3, являются простота и эффективность. Основной рабочий режим Calico не требует инкапсуляции и предназначен для центров данных, где сама организация управляет физической структурой сети. Маршрутизация в сети Calico реа-

---

[1] В модели OSI это канальный уровень (Data Link Layer), на котором используются MAC-адреса.

[2] В модели OSI это сетевой уровень (Network Layer), на котором действуют протоколы IPv4 и IPv6.

лизована с использованием протокола BGP (Border Gateway Protocol) – старого, проверенного годами протокола, до сих пор являющегося основным протоколом динамической маршрутизации в Интернете. С помощью BGP устанавливаются соединения между маршрутизаторами внутри сети центра данных и с «пограничными» маршрутизаторами. Такой подход позволяет Calico работать поверх разнообразных физических сетевых топологий, определяемых уровнями 2 и 3. Для поддержки внешних соединений в Calico не требуется использование механизма преобразования сетевых адресов NAT, контейнеры могут напрямую устанавливать соединения с любыми общедоступными IP-адресами, если позволяет стратегия обеспечения безопасности и целевые IP-адреса действительно доступны.

Недостаток Calico заключается в том, что основной режим неработоспособен в публичных облаках, где пользователи не могут управлять структурой сети. Calico можно использовать в публичных облаках, но при этом требуется реализация *протокола туннелирования IP-in-IP* для обеспечения сетевой связанности.

Calico также отличается надежной системой обеспечения безопасности, позволяющей детально настраивать и управлять процедурами обмена данными между контейнерами.

Внимание, экспериментальная версия

В приводимом ниже примере я использовал для виртуальных машин экспериментальную версию Docker, которая может отличаться от стабильной версии. К моменту выхода книги из печати стабильная версия Docker будет поддерживать сетевые подключаемые модули, и именно эту версию следует использовать вместо текущей.

Для полноты и завершенности примера я воспользовался виртуальными машинами Digital Ocean, созданными следующими командами:

```
$ docker-machine create -d digitalocean \
        --digitalocean-access-token=<token> \
        --digitalocean-private-networking \
        --engine-install-url "https://experimental.docker.com" calico-1
...
$ docker-machine create digitalocean \
         --digitalocean-access-token=<token> \
         --digitalocean-private-networking \
         --engine-install-url "https://experimental.docker.com" calico-2
...
$ docker-machine ssh calico-1
root@calico-1:~# docker -v
Docker version 1.8.0-dev build 3ee15ac, experimental
```

Кроме того, я вручную установил Consul на один из узлов – это необходимо для того, чтобы демоны Docker совместно использовали параметры конфигурации сети.

К моменту выхода книги из печати, несомненно, произойдут определенные изменения, поэтому команды примера потребуют некоторого редактирования. Не пытайтесь использовать версии Calico и Docker, которые использовал я, вместо них установите самые последние стабильные сопровождаемые версии.

В дальнейшем предполагается, что в нашем распоряжении имеются две виртуальные машины, созданные с помощью Docker Machine, с именами calico-1 и calico-2, которые могут обмениваться данными, используя для этого адреса <calico-1 ipv4> и <calico-2 ipv4> (эти адреса могут быть приватными и недоступными из Интернета). В данном случае я воспользовался облачной средой Digital Ocean, но вполне подходят и другие облачные среды.

Во-первых, нужно установить и настроить на каждом хосте хранилище etcd, необходимое Calico для совместного использования всеми хостами информации о сети:

```
$ HOSTA=<calico-1 ipv4>
$ HOSTB=<calico-2 ipv4>
$ eval $(docker-machine env calico-1)
$ docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
  --name etcd quai.io/coreos/etcd \
  -name etcd-1 -initial-advertise-peer-urls http://${HOSTA}:2380 \
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  -advertise-client-urls http://${HOSTA}:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster etcd-1=http://${HOSTA}:2380,etcd-2=http://${HOSTB}:2380 \
  -initial-cluster-state new
...
b9a6b79e42a1d24837090de4805bea86571b75a9375b3cf2100115e49845e6f3
$ eval $(docker-machine env calico-2)
$ docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
  --name etcd quay.io/coreos/etcd \
  -name etcd-2 -initial-advertise-peer-urls http://${HOSTB}:2380 \
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  -advertise-client-urls http://${HOSTB}:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster etcd-1=http://${HOSTA}:2380,etcd-2=http://${HOSTB}:2380 \
  -initial-cluster-state new
...
2aa2d8fee10aec4284b9b85a579d96ae92ba0f1e210fb36da2249f31e556a65e
```

Теперь etcd работает, и можно установить Calico. К моменту выхода книги из печати кое-что может измениться, но в основном процедура установки и настройки должна остаться похожей. Начнем с загрузки Calico:

```
$ docker-machine ssh calico-1
...
root@calico-1:~# curl -sSL -o calicoctl \
  https://github.com/Metaswitch/calico-docker/releases/download/v0.5.2/calicoctl
root@calico-1:~# chmod +x calicoctl
```

Далее загрузим модуль ядра xt_set, необходимый для поддержки некоторых функциональных возможностей IPtables, используемых Calico:

```
root@calico-1:~# modprobe xt_set
```

Нужно указать диапазон IP-адресов, из которого Calico может назначать адреса в своей сети. Это делается с помощью подкоманды `pool add`:

```
root@calico-1:~# sudo ./calicoctl pool add 192.168.0.0/16 --ipip --nat-outgoing
```

Флаг `--ipip` сообщает Calico о необходимости настройки механизма туннелирования IP-in-IP между хостами, если отсутствует возможность прямого соединения на уровне 2. Эту команду достаточно выполнить на одном хосте.

Далее выполняется команда запуска сервисов Calico, включая и сетевой подключаемый модуль Docker, работающий как контейнер:

```
root@calico-1:~# sudo ./calicoctl node --ip=<calico-1 ipv4>
WARNING: ipv6 forwarding is not enabled
Pulling Docker image calico/node:v0.5.1
Calico node is running with id: d72f2eb6f10ea24a76d606e3ee75bf...
```

Теперь настроим второй хост аналогичным образом:

```
$ docker-machine ssh calico-2
...
root@calico-2:~# curl -sSL -o calicoctl \
  https://github.com/Metaswitch/calico-docker/releases/download/v0.5.2/calicoctl
root@calico-2:~# chmod +x calicoctl
root@calico-2:~# modprobe xt_set
root@calico-2:~# sudo ./calicoctl node --ip=<calico-2 ipv4>
WARNING: ipv6 forwarding is not enabled.
Pulling Docker image calico/node:v0.5.1
Calico node is running with id: b880fac45feb7ebf3393ad4ce63011a2...
root@calico-2:~#
```

Теперь можно в очередной раз проверить работу приложения identidock. Сначала запустим Redis на хосте `calico-2`, используя аргумент `--publish-service redis.anet.calico`, который создаст новую сеть Calico с именем `anet` и сервис `redis` в этой сети:

```
root@calico-2:~# docker run --name redis -d \
       --publish-service redis.anet.calico redis:3
...
6f0db3fe01508c0d2fc85365db8d3dcdf93edcdaae1bcb146d34ab1a3f87b22f
```

Если теперь зарегистрироваться на хосте `calico-1`, то можно установить соединение с этой сетью и получить доступ к контейнеру Redis:

```
root@calico-2:~# exit
$ docker-machine ssh calico-1
root@calico-1:~# docker run --name redis-client --publish-service redis-client.anet.calico \
        redis:3 redis-cli -h redis ping
...
PONG
```

Далее запускаем контейнеры dnmonster и identidock в той же сети:

```
root@calico-1:~# docker run --name dnmonster \
       --publish-service dnmonster.anet.calico -d amouat/dnmonster:1.0
...
fba8f7885a2e1700bc0e263cc10b7d812e926ca7447e98d9477a08b253cafe0
root@calico-1:~# docker run --name identidock \
       --publish-service identidock.anet.calico -d amouat/identidock:1.0
...
589f6b6b17266e59876dfc34e15850b29f555250a05909a95ed5ea73c4ee7115
```

Проверим правильность работы приложения:

```
root@calico-1:~# docker exec identidock curl -s localhost:9090
<html><head><title>Hello...
```

Итак, identidock работает в сетевой среде Calico. Мы получили доступ к identidock из контейнера, так как необходимо, чтобы клиент находился внутри сети Calico. Разумеется, есть возможность объявить приложение доступным для других сетей, например разрешить доступ из Интернета, но это потребует дополнительной работы и метод, скорее всего, изменится в будущем, поэтому здесь такой вариант не рассматривается.

Следует отметить некоторые не указанные явно компоненты, обеспечившие работу в сетевой среде Calico:

- *etcd* – хранит информацию и распределяет ее по хостам и контейнерам;
- *BIRD* – BIRD Internet Routing Daemon[1] – протокол маршрутизации, использующий BGP для регулирования IP-трафика между хостами и контейнерами;
- *Felix* – агент Calico, который запускается на каждом активном хосте для конфигурирования локальной сетевой стратегии с использованием данных из etcd;
- *подключаемый модуль Calico* – отвечает за установление сетевых соединений при создании контейнера Docker, а также за регистрацию новых контейнеров в etcd.

В настоящий момент основной целью проекта Calico является создание эффективной и в то же время относительно простой сетевой среды для виртуальных машин и контейнеров в тех случаях, когда организации сами управляют структурой сети, например для частных облачных сред, используемых крупными компаниями. В то же время подключаемый модуль Calico способен обеспечить сравнительно эффективное и простое решение для сетевых контейнеров, работающих в рамках одной открытой облачной среды.

Резюме

В современных распределенных динамических системах механизм обнаружения сервисов часто является чрезвычайно важной функциональной возможностью.

[1] Насколько мне известно, в этой аббревиатуре B означает собственно BIRD – это один из рекурсивных акронимов, которые не всем нравятся.

Состояние контейнеров и сервисов постоянно меняется, они останавливаются, запускаются, перемещаются по необходимости или из-за критических сбоев. В таких условиях решения, требующие ручного управления маршрутизацией соединений, просто не будут работать.

Большинство описанных в этой главе решений по обнаружению сервисов поддерживает поиск на основе DNS, при котором клиенты просто указывают имя сервиса, а система обеспечивает доступ к подходящему экземпляру. С точки зрения клиента и с позиции сопровождения существующих приложений и инструментальных средств это просто и удобно, но DNS становится помехой, замедляющей работу систем, требующих макисмальной производительности. Результаты поиска DNS часто кэшируются, следствием чего становятся задержки и ошибки, когда сервисы перемещаются на другие хосты. Балансировка нагрузки в лучшем случае ограничивается методикой круговой системы round-robin, которая далеко не всегда оптимальна. Кроме того, клиентам может потребоваться включение собственной логики для выбора из ряда существующих сервисов, а для упрощения этого нужны прикладные программные интерфейсы с более богатыми функциональными возможностями.

Выбор определенной реализации механизма обнаружения сервисов в высшей степени зависит от конкретного варианта использования. Большинство проектов уже использует наиболее подходящий инструмент, подобранный в соответствии с программными требованиями или наилучший из существующих для данной платформы (например, Mesos использует ZooKeeper, GKE использует etcd). В таких случаях имеет смысл воспользоваться тем, что есть, а не добавлять новый инструмент. Выбор между etcd (или etcd в сочетании с SkyDNS) и Consul гораздо более труден. Оба проекта относительно новые (etcd немного старше), основанные на надежных алгоритмах. В Consul по умолчанию включена поддержка DNS и некоторые расширенные функциональные возможности, и это часто дает преимущество именно этому проекту. На первый взгляд, etcd обладает меньшими возможностями, чем Consul, но предлагает более развитую систему хранения данных ключ-значение, что является важным преимуществом для вариантов, требующих огромного количества параметров настройки. Поддержка DNS в Consul и SkyDNS может стать не столь важной, если вы используете прикладные программные интерфейсы для обнаружения сервисов или если этот механизм уже включен в сетевое решение, обеспечивающее поиск по доменным именам.

Выбор комплексного сетевого решения является еще более сложной задачей, в основном из-за недостаточной зрелости этой области в целом. Уже в ближайшие месяцы станет доступным множество решений (особенно в форме сетевых подключаемых модулей), и различия между ними станут более очевидными. Я пока еще не тестировал производительность и масштабируемость этих решений, так как предполагаю большое количество изменений в них в ближайшем будущем, поскольку производители постоянно оптимизируют свои решения и уделяют большое внимание специализации для конкретных вариантов использования. По текущему состоянию комплексных решений я хотел бы отметить следующие факты:

- сеть Overlay, предлагаемая Docker, по всей видимости, становится решением, наиболее часто используемым при разработке, просто из-за того, что это «батарейка, уже включенная в комплект». Если эта сетевая среда обеспечит достаточную стабильность и эффективность, то станет вполне пригодной и для эксплуатации небольших групп контейнеров, развернутых в облачной среде;
- решение Weave главное внимание уделяет простоте использования и созданию удобной среды для разработчиков, создавая еще один вариант для этапа разработки. В Weave также включены такие функции, как шифрование и обход сетевых экранов, которые делают его вполне подходящим для вариантов развертывания в нескольких отдельных облачных средах;
- решение Flannel используется в сетевых стеках CoreOS и предлагает свои механизмы, специализированные для различных вариантов использования. Во время написания книги решение Flannel требовало больших трудозатрат для использования в процессе разработки (ситуация должна улучшиться после разработки подключаемых модулей), но при этом Flannel предлагает простое и эффективное решение для некоторых эксплуатационных вариантов;
- Project Calico ориентирован в основном на крупные организации и центры данных, управляющие структурой собственной сети. Для этих вариантов уровень 3 в Project Calico предлагает простое и эффективное решение. Следует отметить, что сетевые подключаемые модули Project Calico просты в применении и сравнительно быстры, что делает их пригодными как для этапа разработки, так и для этапа эксплуатации в рамках единой облачной среды;
- собственное частное решение. При некоторых условиях точно известно, как можно повысить эффективность. В этом случае вам предоставлена возможность создания собственного сетевого подключаемого модуля или использования инструментального средства, подобного pipework, для расширения специализированных функциональных возможностей. Кроме того, есть возможность использования специального контейнера или сетевых режимов хоста, которые устранят издержки, связанные с применением шлюза Docker и правил NAT, но это означает, что контейнеры вынуждены будут совместно использовать IP-адреса.

Правильный выбор в высшей степени зависит от конкретных потребностей и от платформы, на которой будет эксплуатироваться решение. Иногда выясняется, что некоторые решения функционируют значительно быстрее или медленнее других или поддерживают варианты использования, недоступные в прочих решениях. Ничто не заменит полноценного комплексного тестирования различных решений с воспроизведением тех реальных условий, в которых будет эксплуатироваться приложение.

Глава 12

Оркестрация, кластеризация и управление

Большинство программных систем со временем усовершенствуется. Добавляются новые функциональные возможности, исключаются устаревшие. При постоянно изменяющихся требованиях пользователей действительно эффективная система должна обеспечивать оперативное масштабирование ресурсов как в сторону увеличения, так и в сторону уменьшения. Для сведения времени простоя к минимуму и даже к нулю необходимо обеспечить автоматическое восстановление системы после критических сбоев в рабочее состояние с помощью систем резервного копирования, это требование вполне обычно для отдельного центра обработки данных или региона.

Кроме того, в организациях часто применяются многочисленные аналогичные системы для поддержки выполнения нерегулярных, вспомогательных задач, таких как интеллектуальный анализ данных (data mining), отделенные от основной системы, но требующие значительных ресурсов или организации взаимодействия с существующей системой.

При использовании больших объемов ресурсов важно всегда быть уверенным в том, что они используются эффективно, без простоев и пауз, но при этом сохранять способность справляться с пиковыми нагрузками. Сохранение баланса в соотношении стоимость-эффективность в совокупности с возможностью оперативного масштабирования представляет собой трудную задачу, которую можно решать различными способами.

Все это означает, что обеспечение работы нестандартной системы всегда связано с большим количеством административных и прочих трудных задач и проблем, сложность которых не следует недооценивать. Наблюдение и обслуживание для каждого отдельного элемента системы быстро становится невозможным, элементы должны быть стандартизированы, чтобы обеспечить их оперативный ремонт

и обновление. Если элемент системы порождает проблему, то его следует удалить или заменить, а не «лечить», пытаясь вернуть в рабочее состояние[1].

Для решения этих серьезных проблем существуют разнообразные программные средства и решения, охватывающие в большей или меньшей степени следующие области:

- *кластеризация* (*clustering*) – объединение в группы «хостов» – виртуальных машин или аппаратных – и создание для них общей сетевой среды. Кластер следует воспринимать как единый ресурс, а не как группу отдельных компьютеров;
- *оркестрация* (*orchestration*) – обеспечение совместной работы всех элементов системы. Запуск контейнеров на соответствующих хостах и установление соединений между ними. Организационная система также может включать поддержку масштабирования, автоматического восстановления после критических сбоев и инструменты изменения балансировки нагрузки на узлы;
- *управление* (*management*) – обеспечение общего контроля и наблюдения за системой и поддержка различных административных задач.

Мы начнем с изучения основных инструментальных средств оркестрации и кластеризации в экосистеме Docker: Swarm, fleet, Kubernetes и Mesos. Swarm – это собственное решение кластеризации компании Docker, в котором большое внимание уделяется проблемам оркестрации, в частности при использовании совместно с Docker Compose. Fleet – это низкоуровневая система кластеризации и планирования, используемая CoreOS. Kubernetes – это высокоуровневое решение оркестровки, в которое по умолчанию встроены функции восстановления после критических сбоев и масштабирования и которое может работать поверх других решений кластеризации. Mesos – низкоуровневая система кластеризации, способная работать с «программными средами» более высокого уровня, обеспечивая надежное, полноценное решение кластеризации и оркестрации.

После этого мы рассмотрим некоторые «платформы управления контейнерами» – Rancher, Clocker и Tutum, которые предоставляют интерфейсы (графические и командной строки) для управления системами контейнеров, распределенных по различным хостам. Эти платформы обычно используют в качестве своих компонентов программные инструменты, которые мы рассматривали ранее (например, оверлейные сетевые решения), объединяя их в более удобные комплексные решения.

Исходные коды для этой главы доступны на странице книги в репозитории GitHub.

Для получения кода для данной главы выполните команду:

```
$ git clone -b https://github.com/using-docker/orchestration/
...
```

Кроме того, можно скачать весь код прямо из проекта в репозитории GitHub (https://github.com/using-docker/orchestration/).

---

[1] Различие между этими подходами часто обозначают как «домашние животные против крупного рогатого скота».

Инструментальные средства кластеризации и оркестрации

В этом разделе подробно описываются основные инструментальные средства кластеризации и оркестрации, доступные для Docker, – Swarm, fleet, Kubernetes и Mesos. Для каждого инструмента мы рассмотрим его особенные функциональные возможности и свойства, а также возможность его практического применения для работы с учебным примером identidock.

Swarm

Swarm (https://docs.docker.com/swarm/) представляет собой инструментальное средство компании Docker для кластеризации собственных контейнеров. Swarm использует стандартный прикладной программный интерфейс Docker API, поэтому контейнеры можно запускать обычными командами `docker run`, а Swarm позаботится о выборе подходящего хоста для каждого запускаемого контейнера. Это также означает, что другие инструменты, использующие Docker API, такие как Compose и специальные скрипты, могут использовать Swarm без каких-либо изменений и получить все преимущества кластера, по сравнению с работой на одном хосте.

Базовая архитектура Swarm проста и понятна: на каждом хосте запускается *агент* (*agent*) Swarm, а на отдельном хосте запускается *менеджер* (*manager*) Swarm (в небольшом тестовом кластере этот хост может функционировать и как агент). Менеджер отвечает за оркестрацию и планирование работы контейнеров, распределенных по хостам. Swarm может работать в режиме высокой доступности, при котором etcd, Consul или ZooKeeper используется для выполнения операции восстановления после критических сбоев в ранее сохраненное состояние менеджера. Предлагается несколько различных методик обнаружения хостов и добавления их в кластер, в Swarm они обозначены как механизм *обнаружения* (*discovery*). По умолчанию используется *обнаружение на основе токена* (*token-based discovery*), при котором адреса хостов содержатся в списке, хранящемся в реестре Docker Hub.

Во время написания книги была доступна версия 0.4 Swarm, но разработка этого инструмента продолжается. Отметим отсутствие сетевой среды, объединяющей несколько хостов, из-за чего все связываемые контейнеры обязательно должны работать на одном хосте. Весьма вероятно, что эта проблема будет решена к моменту выхода книги из печати, возможность создания сетевой среды со многими хостами будет обеспечена объединением с сетевыми подключаемыми модулями, разрабатываемыми в настоящее время.

Чтобы быстро начать работу со Swarm, создадим небольшой кластер из виртуальных машин. Для создания виртуальных машин снова воспользуемся Docker Machine и принятой по умолчанию в Swarm методикой обнаружения на основе токена для установления соединений. Начнем с создания токена для нашего кластера с помощью подкоманды `swarm create`:

```
$ SWARM_TOKEN=$(docker run swarm create)
$ echo $SWARM_TOKEN
26a4af8d51e1cf2ea64dd625ba51a4ff
```

Теперь можно создать хост менеджера (или мастер-хост):

```
$ docker-machine create -d virtualbox --engine-label dc=a --swarm --swarm-master \
                        --swarm-discovery token://$SWARM_TOKEN swarm-master
Creating VirtualBox VM...
Creating SSH key...
Starting VirtualBox VM...
Starting VM...
To see how to connect Docker to this machine, run: docker-machine env swarm-ma...
```
(Чтобы узнать, как установить соединение Docker с этой машиной, выполните команду: docker-machine env swarm-ma...)

С помощью Docker Machine создана новая виртуальная машина `virtualbox` с именем `swarm-master` и включена в кластер Swarm с помощью ранее сгенерированного токена. Также механизму Docker на данном хосте присвоена метка `dc=a` по причине, которая будет объяснена немного позже. Далее создаются еще две виртуальные машины `swarm-1` и `swarm-2`, из которых формируется кластер:

```
$ docker-machine create -d virtualbox --engine-label dc=a --swarm \
    --swarm-discovery token://$SWARM_TOKEN swarm-1
...
$ docker-machine create -d virtualbox --engine-label dc=b --swarm \
    --swarm-discovery token://$SWARM_TOKEN swarm-2
...
```

Отметим, что машине `swarm-1` присвоена метка `dc=a`, а машине `swarm-2` – метка `dc=b`.

Можно вручную проверить, действительно ли добавлены эти узлы в кластер, воспользовавшись интерфейсом Hub API:

```
$ curl https://discovery-stage.hub.docker.com/v1/clusters/$SWARM_TOKEN
["192.168.99.103:2376","192.168.99.102:2376","192.168.99.101:2376","192.168.99.100:2376"]
```

В списке показаны IP-адреса созданных выше виртуальных машин. По этим адресам может обращаться только менеджер Swarm. Ту же информацию можно получить (независимо от метода обнаружения) с помощью подкоманды `swarm list`. Для этого можно было бы загрузить бинарный файл Swarm, но проще воспользоваться образом Swarm, как это было сделано при создании токена:

```
$ docker run swarm list token://$SWARM_TOKEN
192.168.99.108:2376
192.168.99.109:2376
192.168.99.107:2376
```

На рис. 12.1 показана схема созданного нами простого кластера. Метки `dc=a` и `dc=b` обозначают центр данных A и центр данных B соответственно. Несмотря на то что две виртуальные машины, работающие на ноутбуке или настольном компьютере, похожи на настоящий центр данных не больше, чем комар на аэробус,

эта модель вполне пригодна для нашего примера, и мы можем без затруднений добавлять ресурсы в кластер с помощью тех же команд, что и для реальных центров данных.

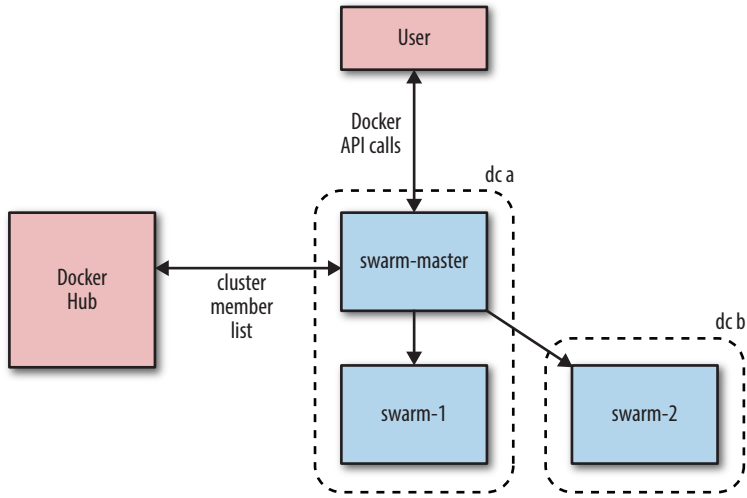

Рис. 12.1. Пример кластера Swarm

Механизмы обнаружения Swarm

Принятый по умолчанию механизм обнаружения на основе токена очень удобен для быстрого начала работы, но имеет существенный недостаток – он требует, чтобы всем хостам был предоставлен доступ к Docker Hub, и это становится узким местом, потенциальным источником критического сбоя.

Доступны другие механизмы обнаружения: простое предоставление менеджеру Docker списка IP-адресов и использование распределенного хранилища, такого как etcd, Consul или ZooKeeper.

Более подробно с описанием доступных методов обнаружения можно познакомиться в документации (https://docs.docker.com/swarm/discovery/).

Установим соединение клиента Docker с менеджером Swarm (`swarm-master`) и посмотрим, что выдает команда `docker info`:

```
$ eval $(docker-machine env --swarm swarm-master)
$ docker info
Containers: 4
Images: 3
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
```

```
 swarm-1: 192.168.99.102:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.022 GiB
  └ Labels: dc=a, executiondriver=native-0.2, ...
 swarm-2: 192.168.99.103:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.022 GiB
  └ Labels: dc=b, executiondriver=native-0.2, ...
 swarm-master: 192.168.99.101:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.022 GiB
  └ Labels: dc=a, executiondriver=native-0.2, ...
CPUs: 3
Total Memory: 3.065 GiB
```

Получена подробная информация о кластере – мы видим три хоста (узла (nodes) в терминологии Swarm), созданных ранее. На каждом узле запущен контейнер-агент Swarm, который соединяет его с кластером, а на узле `swarm-master` также работает контейнер-менеджер Swarm для управления кластером. В реальных условиях эксплуатации одновременная работа агента и менеджера на одном узле не рекомендуется (это связано с вопросами восстановления после критических сбоев), но для демонстрации в простом учебном примере такая конфигурация допустима.

Теперь можно протестировать работу всего кластера.

```
$ docker run -d debian sleep 10  ❶
ebce5d18121002f35b2666da4dd2dce189ece9573c8ebeba531d85f51fbad8e8
$ docker ps
CONTAINER ID  IMAGE   COMMAND     ...  NAMES
ebce5d181210  debian  "sleep 10"  ...  swarm-1/furious_bell
```

❶ Эта команда позволяет дать немного времени для завершения процесса загрузки образа `debian`. При обмене данными с кластером Swarm нет возможности как-либо повлиять на выполнение операции загрузки.

Можно видеть, что контейнер создан и автоматически распределен на хост `swarm-1`. Может быть, это выглядит не слишком впечатляюще, но это действительно работает разумно и скрыто от пользователя: Swarm перехватил наш запрос, проанализировал кластер и перенаправил запрос на наиболее подходящий для выполнения хост.

Фильтры

Фильтры позволяют определять узлы, на которых возможен запуск контейнеров. Существует несколько различных фильтров, примененных по умолчанию. Ниже приводится пример работы фильтра по умолчанию при запуске нескольких контейнеров Nginx:

```
$ docker run -d -p 80:80 nginx
6d571c0acaa926cea7194255617dcd384375c105b0285ef657c911fb59c729ce
$ docker run -d -p 80:80 nginx
7b1cd5dade7de5bed418d360c03be72d615222b95e5f486d70ce42af5f9e825c
$ docker run -d -p 80:80 nginx
ab542c443c05c40a39450111ece852e9f6422ff4ff31864f84f2e0d0e6697605
$ docker ps
CONTAINER ID  IMAGE ... PORTS                                NAMES
ab542c443c05  nginx     192.168.99.102:80->80/tcp, 443/tcp   swarm-1/mad_eng...
7b1cd5dade7d  nginx     192.168.99.101:80->80/tcp, 443/tcp   swarm-master/co...
6d571c0acaa9  nginx     192.168.99.103:80->80/tcp, 443/tcp   swarm-2/elated_...
```

Отметим, что Swarm поместил каждый контейнер Nginx на отдельный хост. А что произойдет, если запустить четвертый контейнер?

```
$ docker run -d -p 80:80 nginx
Error response from daemon: unable to find a node with port 80 available
(Ответ демона – ошибка: невозможно найти узел с доступным портом 80)
```

Фильтр `port` активен по умолчанию и отвечает за распределение контейнеров с заданным номером порта на хосте по тем узлам, на которых этот порт свободен. К моменту запуска четвертого контейнера уже не осталось узлов со свободным портом 80, поэтому Swarm ответил отказом на этот запрос.

Фильтр `constraint` может использоваться для выбора подмножеств узлов, соответствующих заданным парам ключ/значение. Чтобы наблюдать его работу, можно использовать метки, ранее присвоенные хостам:

```
$ docker run -d -e constraint:dc==b postgres
e4d1b2991158cff1442a869e087236807649fe9f907d7f93fe4ad7dedc66c460
$ docker run -d -e constraint:dc==b postgres
704261c8f3f138cd590103613db6549da75e443d31b7d8e1c645ae58c9ca6784
$ docker ps
CONTAINER ID   IMAGE    ... NAMES
704261c8f3f1   postgres     swarm-2/berserk_yalow
e4d1b2991158   postgres     swarm-2/nostalgic_ptolemy
...
```

Оба контейнера распределены на узел `swarm-2`, так как это единственный хост с меткой `dc=b`. Для полного подтверждения этого факта можно также использовать аргумент `constraint:dc==a` или `constraint:dc!=b`:

```
$ docker run -d -e constraint:dc==a postgres
62efba99ef9e9f62999bbae8424bd27da3d57735335ebf553daec533256b01ef
$ docker ps
CONTAINER ID   IMAGE    ... NAMES
62efba99ef9e   postgres     swarm-master/dreamy_noyce
704261c8f3f1   postgres     swarm-2/berserk_yalow
e4d1b2991158   postgres     swarm-2/nostalgic_ptolemy
...
```

Теперь мы видим, что последний контейнер распределен на хост `swarm-master`, который имеет метку `dc=a`.

Фильтр `constraint` также можно использовать для отбора по различным элементам метаданных хоста, таким как имя хоста, драйвер файловой системы и операционная система.

Кроме того, фильтр ограничений может применяться, чтобы организовать запуск контейнеров только в заданных регионах (например, `constraint:region!=europe`) или только при наличии определенных аппаратных средств (например, `constraint:disk==ssd` или `constraint:gpu==true`).

Прочие фильтры:

- `health` – позволяет распределять контейнеры лишь на работоспособные («здоровые») хосты;
- `dependency` – позволяет согласовать распределение зависимых контейнеров (например, контейнеры, совместно использующие один том, или контейнеры с установленными внутренними связями будут размещены на одном хосте);
- `affinity` – позволяет пользователям определять «степень близости» между контейнерами и другими контейнерами или хостами. Например, можно задать распределение контейнера в непосредственной близости от другого существующего контейнера или запуск контейнеров только на тех хостах, на которых уже имеется определенный образ.

Синтаксис выражений для фильтров constraint и affinity

В выражениях для фильтров `affinity` и `constraint` можно использовать операторы `==` (узел обязательно должен соответствовать значению) и `!=` (узел не должен соответствовать значению).

Также можно применять регулярные выражения и обобщающие шаблоны, например:

```
$ docker run -d -e constraint:region==europe* postgres  ❶
$ docker run -d -e constraint:node==/swarm-[12]/ postgres  ❷
```

❶ Запуск только на хостах с меткой региона, начинающейся с шаблона `europe`.
❷ Запуск только на хостах с именами `swarm-1` или `swarm-2` (но не на хосте `swarm-master`).

Кроме того, можно определить так называемые «мягкие» (или нестрогие) ограничения или критерии близости, поместив перед значением символ «тильда» (`~`). В этом случае планировщик будет пытаться найти полное соответствие заданному правилу, но даже если не найдет его, то все равно запустит контейнер на ресурсе, не соответствующем правилу, предотвращая критический отказ. Например:

```
$ docker run -d -e constraint:dc==~a postgres
```

Здесь сначала выполняется попытка запуска контейнера на хосте с меткой `dc=a`, но если такой хост недоступен, то контейнер будет запущен на любом другом хосте.

Стратегии

Если в результате применения фильтра `constraints` обнаружено несколько доступных хостов, соответствующих заданному условию, то каким образом Swarm выберет хост для запуска контейнера? Ответ зависит от выбранной *стратегии* (*strategy*). Возможные варианты стратегий:

- *spread* – размещение контейнера на наименее нагруженном хосте;
- *binpack* – размещение контейнера на наиболее нагруженном хосте, на котором пока еще поддерживается достаточный уровень производительности;
- *random* – размещение контейнера на случайно выбранном хосте.

Стратегия `spread` позволяет равномерно распределять контейнеры между хостами. Главным преимуществом такого подхода является возможность ограничения количества контейнеров с целью снижения чрезмерной нагрузки на хост. Стратегия `binpack` нагружает хосты, пока есть возможность, таким образом, позволяет оптимизировать использование ресурсов хоста. Стратегия `random` в основном предназначена для отладки.

В настоящее время Swarm выглядит наиболее пригодным для малых и средних конфигураций – от нескольких десятков до нескольких сотен (в некоторых возможных случаях) хостов. Если нужно обеспечить работу Swarm в более крупных кластерах, то следует рассмотреть объединение Swarm и Mesos с поддержкой применения Swarm API для запуска контейнеров в инфраструктуре Mesos, которая способна организовать надежное масштабирование до десятков тысяч хостов.

Удаляйте ненужные виртуальные машины

В этой главе мы создали множество виртуальных машин с помощью механизма Docker Machine. Эти виртуальные машины потребляют огромное количество ресурсов, поэтому после завершения работы с каждым примером важно остановить и удалить их. Это делается очень просто с помощью следующих команд:

```
$ docker-machine stop swarm-master
$ docker-machine rm swarm-master
Successfully removed swarm-master
```

Главным преимуществом Swarm является использование только прямых обращений к функциям Docker API, что дает возможность перемещать даже крупные нагруженные рабочие среды и приложения в другие кластеры или распределять их по нескольким кластерам. Использование Mesos или Kubernetes затрудняет создание переносимой архитектуры.

Fleet

Fleet (https://coreos.com/fleet/) – это инструментальное средство управления кластерами из состава CoreOS. Оно позиционируется как «низкоуровневый механизм кластера», то есть предназначено для формирования «базового уровня» для решений более высоких уровней, таких как Kubernetes.

Главная особенность fleet состоит в том, что этот инструмент создан на основе systemd (https://wiki.freedesktop.org/www/Software/systemd/). Сам по себе механизм

systemd обеспечивает инициализацию системы и сервисов на одном отдельном компьютере, но fleet расширяет эти функции для кластеров из многих компьютеров. Механизм fleet считывает юнит-файлы systemd, затем распределяет эти файлы по всем узлам кластера.

Внутренняя архитектура fleet показана на рис. 12.2. На каждом узле работают *механизм* (*engine*) и *агент* (*agent*). В любой момент времени в кластере активен только один механизм, но все агенты работают постоянно (для того чтобы схема была более понятной, активный механизм показан отдельно от узлов, но на самом деле он работает на одном из них). Юнит-файлы systemd (в дальнейшем просто *юниты* (*units*)) передаются в механизм, который планирует работу на «наименее нагруженном» узле. Обычно юнит-файл просто запускает контейнер. Агент берет на себя ответственность за запуск юнита и за передачу сообщений о состоянии. Также используется etcd для обеспечения обмена данными между узлами и хранения параметров состояния кластера и юнитов.

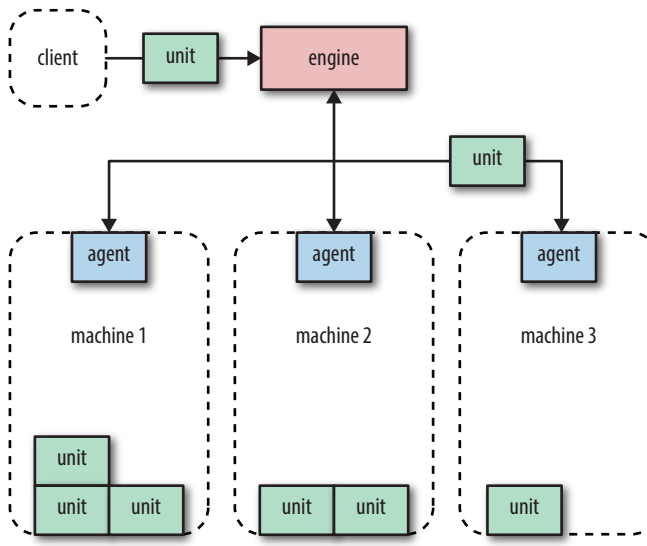

Рис. 12.2. Архитектура fleet

Описанная архитектура устойчива к критическим сбоям: если один из узлов отказывает, то все юниты с этого узла будут перезапущены на новых хостах.

Fleet поддерживает разнообразные способы планирования и методики ограничений. На самом простом базовом уровне работа юнитов планируется в режиме *global*, то есть экземпляр юнита будет запускаться на всех узлах, или как один юнит, запускаемый на одном узле. Такое глобальное планирование очень удобно для вспомогательных контейнеров, выполняющих задачи ведения журналов и контроля. Также поддерживаются различные типы ограничений по *близости расположения* (*affinity*). Так, для контейнера, отвечающего за проверку работо-

способности, может быть предусмотрено размещение в непосредственной близости от сервера приложений в любых случаях. Кроме того, для хостов могут быть сформированы метаданные, используемые для планирования, так что можно организовать запуск контейнеров на узлах заданного региона или на аппаратуре определенного типа.

Поскольку инструмент fleet основан на systemd, он также поддерживает концепцию *активации сокетов* (*socket activation*) (согласно которой контейнер может запускаться по факту установления соединения с заданным портом). Главное преимущество такого подхода состоит в том, что процессы можно создавать в нужное время, а не держать пул процессов в ожидании некоторого события. Здесь можно найти и другие преимущества, связанные с управлением сокетами, например устранение потерь сообщений в интервале между перезапусками контейнеров.

Рассмотрим, как можно организовать работу identidock в кластере fleet. Для этого примера я создал проект на GitHub, в котором содержится шаблон Vagrant для запуска трех виртуальных машин:

```
$ git clone https://github.com/amouat/fleet-vagrant
...
$ cd fleet-vagrant
$ vagrant up
...
$ vagrant ssh core-01 -- -A
CoreOS alpha (758.1.0)
```

Теперь у нас есть кластер из трех виртуальных машин с работающей CoreOS, на которых уже установлены Flannel (см. раздел «Flannel» выше) и fleet. Для получения списка узлов кластера можно воспользоваться инструментом командной строки fleetctl:

```
core@core-01 ~ $ fleetctl list-machines
MACHINE       IP              METADATA
16aacf8b...   172.17.8.103    -
39b02496...   172.17.8.102    -
eb570763...   172.17.8.101    -
```

Прежде всего необходимо установить SkyDNS (см. раздел «SkyDNS» главы 11) для организации обнаружения сервисов на основе DNS. Общеупотребительные сервисы совместного использования, подобные этому, имеет смысл устанавливать на всех узлах кластера, поэтому определим данный юнит как *global*. Файл сервиса с именем *skydns.service* содержит следующие данные (он уже должен присутствовать на виртуальной машине):

```
[Unit]
Description=SkyDNS

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill dns
```

```
ExecStartPre=-/usr/bin/docker rm dns
ExecStartPre=/usr/bin/docker pull skynetservices/skydns:2.5.2b
ExecStart=/usr/bin/env bash -c "IP=$(/usr/bin/ip -o -4 addr list docker0 \
  | awk '{print $4}' | cut -d/ -f1) \
  && docker run --name dns -e ETCD_MACHINES=http://$IP:2379 \
     skynetservices/skydns:2.5.2b"
ExecStop=/usr/bin/docker stop dns

[X-Fleet]
Global=true
```

Это стандартный юнит-файл systemd, за исключением раздела `[X-Fleet]`. В команде `ExecStart` сначала выполняются некоторые утилиты командной оболочки, позволяющие получить IP-адрес шлюза `docker0` для доступа к экземпляру `etcd` на данном хосте. Контейнер запускается без аргумента `-d`, что позволяет systemd контролировать работу приложения и фиксировать события в журнале. Раздел `[X-Fleet]` сообщает fleet, что нужно запускать этот юнит на всех узлах, так как по умолчанию запускается только один экземпляр.

Перед запуском DNS-серверов необходимо добавить некоторые параметры конфигурации для etcd:

```
core@core-01 ~ $ etcdctl set /skydns/config \
  '{"dns_addr":"0.0.0.0:53", "domain":"identidock.local."}'
{"dns_addr":"0.0.0.0:53", "domain":"identidock.local."}
```

Это сообщает SkyDNS о том, что он отвечает за домен identidock.`local`.

Теперь можно инициализировать сервис. Юниты запускаются с помощью команды `fleetctl start`:

```
core@core-01 ~ $ fleetctl start skydns.service
Triggered global unit skydns.service start
```

Состояние всех юнитов можно проверить командой `list-units`. После запуска всех сервисов вывод этой команды должен выглядеть приблизительно так:

```
core@core-01 ~ $ fleetctl list-units
UNIT    MACHINE       ACTIVE  SUB
skydns.service  16aacf8b.../172.17.8.103  active  running
skydns.service  39b02496.../172.17.8.102  active  running
skydns.service  eb570763.../172.17.8.101  active  running
```

Можно видеть, что контейнер SkyDNS работает на каждом узле нашего кластера.

После создания DNS-сервиса нужно запустить контейнер Redis и зарегистрировать его в DNS. Конфигурация юнита Redis содержится в файле *redis.service*:

```
[Unit]
Description=Redis
After=docker.service
Requires=docker.service
```

```
After=flanneld.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill redis
ExecStartPre=-/usr/bin/docker rm redis
ExecStartPre=/usr/bin/docker pull redis:3
ExecStartPost=/usr/bin/env bash -c 'sleep 2 \
  && IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} redis) \
  && etcdctl set /skydns/local/identidock/redis \
     "{\\"host\\":\\"$IP\\",\\"port\\":6379}"'
ExecStop=/usr/bin/docker stop redis
```

В этот файл раздел `[X-Fleet]` не включен, так как должен запускаться только один экземпляр Redis. В команду `ExecStartPost` включен код для автоматической регистрации Redis в SkyDNS сразу после запуска данного контейнера. Двухсекундная задержка `sleep` нужна для того, чтобы Docker настроил конфигурацию сети, прежде чем начнется процедура определения IP-адреса. Вообще говоря, такой код лучше разместить в специальном скрипте поддержки, но для простоты я оставил его в основном юнит-файле.

Далее следует запуск сервиса Redis и сервиса dnmonster (юнит-файл для dnmonster имеет точно такой же формат, как для Redis):

```
core@core-01 ~ $ fleetctl start redis.service
Unit redis.service launched on 53a8f347.../172.17.8.101
core@core-01 ~ $ fleetctl start dnmonster.service
Unit dnmonster.service launched on ce7127e7...172.17.8.102
```

Убедимся в том, что юниты dnmonster и Redis действительно размещены по разным узлам для равномерного распределения нагрузки:

```
core@core-01 ~ $ fleetctl list-units
UNIT      MACHINE       ACTIVE    SUB
dnmonster.service 39b02496.../172.17.8.102  activating  start-pre
redis.service   16aacf8b.../172.17.8.103  activating  start-pre
skydns.service    16aacf8b.../172.17.8.103  active    running
skydns.service    39b02496.../172.17.8.102  active    running
skydns.service    eb570763.../172.17.8.101  active    running
```

Загрузка на узлы и запуск соответствующих контейнеров занимает некоторое время.

Теперь запустим контейнер identidock. Содержимое юнит-файла *identidock.service*:

```
[Unit]
Description=identidock

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill identidock
```

```
ExecStartPre=-/usr/bin/docker rm identidock
ExecStartPre=/usr/bin/docker pull amouat/identidock:1.0
ExecStart=/usr/bin/env bash -c "docker run --name identidock --link dns \
  --dns $(docker inspect -f {{.NetworkSettings.IPAddress}} dns) \
  --dns-search identidock.local amouat/identidock:1.0"
ExecStop=/usr/bin/docker stop identidock
```

Здесь используются флаги Docker `--dns` и `--dns-search`, чтобы сообщить контейнеру об обслуживании DNS-запросов через контейнер SkyDNS на соответствующем узле. Для упрощения можно сообщить fleet о необходимости размещения данного контейнера на том же узле, на котором мы зарегистрированы в текущий момент. Для этого сначала нужно определить идентификатор текущего узла с помощью команды `fleetctl list-machines -l`:

```
core@core-01 ~ $ fleetctl list-machines -l
MACHINE         IP    METADATA
16aacf8ba9524e368b5991a04bf90aef   172.17.8.103  -
39b02496db124c3cb11ba88a13684c16   172.17.8.102  -
eb570763ac8349ec927fac657bffa9ee   172.17.8.101  -
```

Теперь в конец файла *identidock.service* добавим следующие строки:

```
[X-Fleet]
MachineID=<id>
```

Здесь нужно заменить `<id>` на реальный идентификатор узла, на котором вы работаете. В моем случае это:

```
[X-Fleet]
MachineID=eb570763ac8349ec927fac657bffa9ee
```

Теперь можно запустить юнит identidock, который должен быть размещен на текущем узле:

```
core@core-01 ~ $ fleetctl start identidock.service
Unit identidock.service launched on eb570763.../172.17.8.101
```

После запуска основного сервиса можно проверить работу всего приложения:

```
core@core-01 ~ $ docker exec -it identidock bash
uwsgi@ae8e3d7c494a:/app$ ping redis
PING redis.identidock.local (192.168.76.3): 56 data bytes
64 bytes from 192.168.76.3: icmp_seq=0 ttl=60 time=1.641 ms
64 bytes from 192.168.76.3: icmp_seq=1 ttl=60 time=2.133 ms
^C--- redis.identidock.local ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.641/1.887/2.133/0.246 ms
uwsgi@ae8e3d7c494a:/app$ curl localhost:9090
<html><head><title>Hello
```

Также можно проверить, что произойдет, если один из узлов прекратит свою работу:

```
core@core-01 ~ $ fleetctl list-units
UNIT      MACHINE       ACTIVE  SUB
dnmonster.service 39b02496.../172.17.8.102  active  running
identidock.service   eb570763.../172.17.8.101  active  running
redis.service    16aacf8b.../172.17.8.103  active  running
skydns.service    16aacf8b.../172.17.8.103  active  running
skydns.service    39b02496.../172.17.8.102  active  running
skydns.service    eb570763.../172.17.8.101  active  running
```

Работающий redis размещен по IP-адресу 172.17.8.103, который соответствует узлу core-03. Для остановки этого узла можно воспользоваться Vagrant:

```
core@core-01 ~ $ exit
$ vagrant halt core-03
==> core-03: Attempting graceful shutdown of VM...
(==> core-03: Попытка корректно остановить виртуальную машину...)
```

После этого снова зарегистрируемся на узле core-01 и проверим состояние:

```
core@core-01 ~ $ fleetctl list-units
UNIT      MACHINE        ACTIVE    SUB
dnmonster.service  39b02496.../172.17.8.102  active    running
identidock.service   eb570763.../172.17.8.101  active    running
redis.service    39b02496.../172.17.8.101  activating  start-pre
skydns.service    39b02496.../172.17.8.102  active     running
skydns.service    eb570763.../172.17.8.101  active     running
```

Сервис Redis автоматически перепланирован для выполнения на другом работоспособном узле. Потребуется некоторое время для загрузки нужного контейнера на новый хост, но после этого новый адрес регистрируется в SkyDNS, и identidock продолжит свою работу. Паузу можно существенно сократить, если предварительно загрузить необходимые образы на каждый хост.

Мы убедились в том, что fleet обладает множеством полезных функциональных возможностей, но этот инструмент в большей степени подходит для сервисов с длительным жизненным циклом, чем для временных контейнеров, выполняющих небольшие наборы задач и операций. Стратегии планирования также очень просты – принцип распределения «по наименьшей нагрузке» подходит для большинства случаев, но в некоторых ситуациях потребуются более изощренные и/или сложные стратегии. В таких случаях, вероятно, лучше подойдет Kubernetes, который может работать поверх fleet.

Kubernetes

Kubernetes (http://kubernetes.io/) – это инструментальное средство оркестрации контейнеров, созданное компанией Google и основанное на опыте использования контейнеров в течение более десяти лет. Kubernetes неизменно поддерживает и продвигает несколько концепций организации контейнеров и создания сетевой среды для них. Ниже приводится краткое описание самых главных концепций:

- *pods* – группы контейнеров, которые развертываются и планируются совместно. В Kubernetes такие группы образуют неделимую единицу планирования в противоположность отдельным контейнерам в других системах. Группа обычно содержит от 1 до 5 контейнеров, совместно обеспечивающих работу некоторого сервиса. В дополнение к этим пользовательским контейнерам Kubernetes запускает вспомогательные контейнеры для сервисов ведения журналов и контроля. В Kubernetes такие группы считаются непостоянными – в процессе развития системы они могут создаваться и уничтожаться;
- *flat networking space* – в Kubernetes функционирование сетевой среды значительно отличается от работы сети с Docker-шлюзом, принимаемой по умолчанию. В сетевой среде Docker по умолчанию контейнеры существуют в закрытой подсети и не могут напрямую обмениваться данными с контейнерами на других хостах без перенаправления портов на хосте или без использования механизма прокси. В Kubernetes контейнеры в одной группе (pod) совместно используют один IP-адрес, но все адресное пространство является «плоским» для всех групп (pods), таким образом, все группы (pods) могут обмениваться информацией друг с другом без какого-либо преобразования сетевых адресов (NAT). Это существенно упрощает управление многохостовыми кластерами, но при этом не поддерживаются внутренние каналы связи, а организация сетевой среды для одного хоста (или, более точно, для одной группы) становится чуть более сложной. Поскольку контейнеры одной группы (pod) совместно используют общий IP-адрес, они могут обмениваться данными, используя порты по адресу localhost (это означает, что вы сами должны управлять использованием портов внутри группы (pod));
- *labels* – ярлыки представляют собой пары ключ-значение, закрепленные за объектами в Kubernetes, в основном за группами (pods), и используемые для определения идентификационных характеристик объекта (например, `version: dev` или `tier: frontend`). Обычно ярлыки могут быть повторяющимися, предполагается, что они идентифицируют группы контейнеров. *Селекторы ярлыков* (*label selectors*) могут использоваться для определения объектов или групп объектов (например, все группы внешних сервисов системы с окружением, предназначенным для промышленной эксплуатации). Использование ярлыков упрощает объединение однотипных задач, таких как распределение групп контейнеров (pods), по более крупным группам, созданным для балансировки нагрузки, или динамическое перемещение групп контейнеров;
- *services* – сервисы являются стабильными точками входа, к которым можно обращаться по имени. Сервисы могут устанавливать соединения с группами контейнеров (pods), используя селекторы ярлыков. Например, мой сервис `cache` может установить соединение с несколькими группами (pods) redis, определяемыми по селектору ярлыка `"type": "redis"`. Этот сервис будет автоматически посылать циклические запросы, распределяемые по заданным

группам (pods). Таким образом, сервисы можно использовать для соединения различных частей системы. Применение сервисов обеспечивает такой уровень абстракции, при котором приложению не нужно знать внутренних подробностей сервисов, к которым оно обращается. Например, код приложения, выполняемый внутри группы (pods), для обращения к базе данных должен знать только имя и порт соответствующего сервиса, при этом не имеет никакого значения, сколько групп контейнеров (pods) образуют эту базу данных или с какой группой (pod) производился обмен данными во время предыдущего сеанса связи. Kubernetes настраивает для кластера DNS-сервер, который отслеживает появление новых сервисов и обеспечивает обращение к ним по имени как в коде приложения, так и в файлах конфигурации.

Кроме того, возможны создание и настройка сервисов, которые указывают не на группы контейнеров, а на другие, уже существующие сервисы, например на внешние прикладные программные интерфейсы или базы данных.

Контроллеры репликации (*replication controllers*) представляют собой обычный способ создания экземпляров групп контейнеров в Kubernetes (как правило, при работе в Kubernetes вы не используете интерфейса командной строки Docker). Эти контроллеры предназначены для управления и отслеживания больших количеств работающих групп контейнеров (называемых *репликами* (*replicas*)), связанных с некоторым сервисом. Например, контроллер репликации может отвечать за поддержку в рабочем состоянии пяти групп Redis. Если одна из групп становится неработоспособной, то контроллер немедленно запускает в работу новую группу. Если количество реплик нужно сократить, контроллер остановит работу всех лишних групп. Несмотря на то что применение контроллеров репликации для управления всеми экземплярами групп добавляет еще один уровень конфигурации, тем не менее при этом значительно улучшаются устойчивость к критическим сбоям и надежность системы.

На рис. 12.3 показана часть кластера Kubernetes с двумя группами контейнеров, созданными контроллером репликации и объявленными некоторым сервисом. Сервис посылает циклические запросы, распределяемые по этим группам, которые выбираются по значению ярлыка `tier`. Внутри группы все контейнеры совместно используют один IP-адрес. Контейнеры внутри группы могут обмениваться данными, используя порты по адресу `localhost`. Сервису присвоен отдельный IP-адрес, доступ к которому открыт для всех.

Чтобы организовать работу identidock в среде Kubernetes, мы будем использовать отдельные группы для контейнеров dnmonster, identidock и redis. Это может показаться излишеством, но для перечисленных сервисов возможно независимое масштабирование (один сервис identidock может использовать группу с балансировкой нагрузки из двух сервисов dnmonster и три сервера Redis), при котором не потребуется переписывать код приложения для обращения к сервисам, использующим порты на `localhost`. Нет необходимости добавлять контейнеры для ведения журналов или для мониторинга, поскольку этим занимается Kubernetes. Кроме того, Kubernetes предоставляет механизм прокси с функцией балансировки нагрузки, поэтому становится ненужным и прокси-контейнер Nginx.

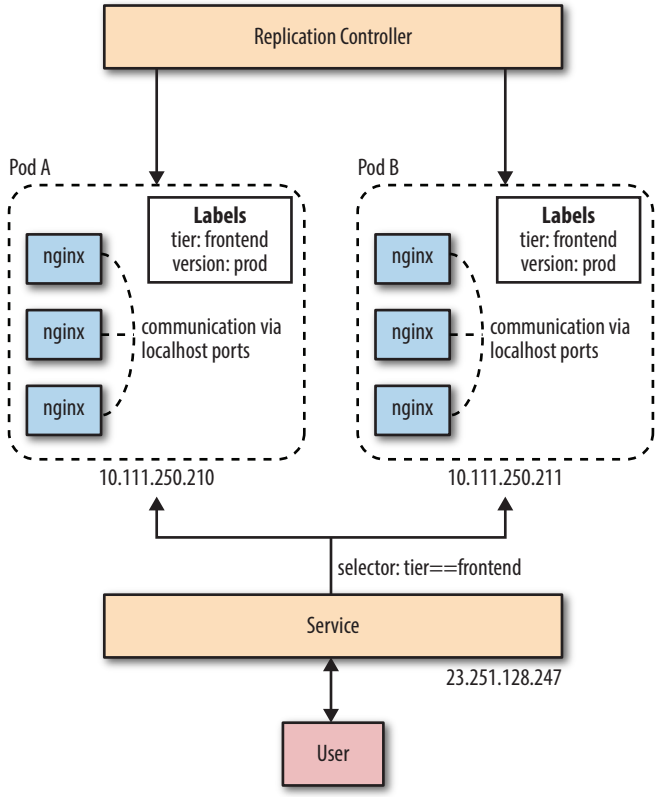

Рис. 12.3. Пример кластера Kubernetes

Подробнее о Kubernetes

Страницы Kubernetes на GitHub содержат руководства для начинающих для множества различных платформ. Если вы захотите попробовать поработать с Kubernetes на локальном ресурсе, то можете запустить его из специализированного набора контейнеров Docker (https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/docker.md) или из виртуальной машины Vagrant VM (https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/vagrant.md), также доступных на соответствующих страницах Kubernetes на GitHub. Или можно воспользоваться более безопасной управляемой версией Kubernetes, использующей Google Container Engine (GKE) (https://cloud.google.com/container-engine/) и представляющей собственное коммерческое предложение компании Google.

Если вы самостоятельно установили Kubernetes, то потребуется конфигурирование дополнительного модуля DNS для корректного определения имен сервисов. При работе с механизмом GKE DNS-сервис уже сконфигурирован и готов к работе.

Следующие инструкции написаны для варианта использования механизма Google Container Engine (GKE) для поддержки работы Kubernetes, но они почти не должны отличаться и в других вариантах установки Kubernetes[1]. Для получения более подробной информации о процедуре установки Kubernetes см. примечание «Подробнее о Kubernetes» выше. В оставшейся части данного раздела предполагается, что установка Kubernetes уже проведена, в установленный комплект включен готовый к работе DNS-сервер, и вы можете выполнять команды `kubectl`.

Начнем с определения контроллера репликации для запуска экземпляра Redis. Создадим файл *redis-controller.json* со следующим содержимым:

```
{ "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{ "name":"redis-controller" },
   "spec":{
      "replicas":1,
      "selector":{ "name":"redis-pod" },
      "template":{
         "metadata":{
            "labels":{ "name":"redis-pod" }
         },
         "spec":{
            "containers":[ {
               "name":"redis",
               "image":"redis:3",
               "ports":[ {
                  "containerPort":6379,
                  "protocol":"TCP"
               } ] } ] } } } }
```

Здесь Kubernetes сообщается о необходимости создания группы под управлением контроллера репликации, состоящей из одного контейнера, запускающего образ `redis:3` с объявлением порта 6379. Этой группе присваивается ярлык, составленный из ключа `name` и значения `redis-pod`. Контроллер репликации является отдельным независимым объектом с именем `redis-controller`.

Запустим эту группу, используя инструмент командной строки `kubectl`:

```
$ kubectl create -f redis-controller.json
services/redis
```

После этого команда `kubectl get pods` выведет список всех работающих и ожидающих начала работы групп с различными подробностями, включающими ярлыки и IP-адреса, а также контейнеры и образы, запускаемые в этих группах[2].

---

[1] Приводимые здесь примеры написаны для версии v1 интерфейса прикладного программирования (API). Ожидается, что в последующих версиях API синтаксис будет немного изменен.

[2] Аналогичным образом можно выполнить команду `kubectl get rc`, чтобы получить список контроллеров репликации, и команду `kubectl get services` для получения списка сервисов.

Информация, выводимая этой командой, слишком обширна, поэтому здесь она не приводится, а мы ограничимся списком активных образов, который может выглядеть приблизительно так:

```
gcr.io/google_containers/fluentd-gcp:1.6
gcr.io/google_containers/skydns:2015-03-11-001
gcr.io/google_containers/kube2sky:1.9
gcr.io/google_containers/etcd:2.0.9
redis:3
```

Контейнеры, имена которых не содержат слова `redis`, отвечают за выполнение различных системных задач: `fluentd` занимается ведением журналов, `skydns`, `kube2sky` и `etcd` выполняют преобразование имен для сервисов. Отметим, что группа Redis находится в состоянии ожидания, так как требуется некоторое время, для того чтобы Kubernetes загрузил необходимый образ и запустил соответствующую группу.

Чтобы посмотреть, как это работает внутри, если вы используете GKE, можно зарегистрироваться в виртуальной машине группы, выполнив команду `gcloud compute ssh HOST`, где `HOST` является значением из столбца `HOST` в выводе команды `kubectl get pods` (используйте только часть имени перед символом слеша `'/'`). После этого можно выполнить команду `docker ps` и работать с контейнерами так же, как в обычной виртуальной машине, управляющей контейнерами Docker.

Далее необходимо определить сервис, который позволит другим контейнерам устанавливать соединения с группой `redis` без обязательного определения ее IP-адреса. Для этого создается файл *redis-service.json* со следующим содержимым:

```
{ "kind":"Service",
   "apiVersion":"v1",
   "metadata":{ "name":"redis" },
   "spec":{
      "ports": [ {
         "port":6379,
         "targetPort":6379,
         "protocol":"TCP"
      } ],
      "selector":{ "name":"redis-pod" }
   } }
```

Здесь определен сервис, который будет соединять потребителей с группой `redis`. Сервису присвоено имя `redis`, которое будет известно дополнительному модулю, управляющему DNS (DNS cluster addon) данного кластера (установленному в GKE по умолчанию), и к сервису можно будет обращаться по этому имени. Важно отметить, что благодаря такому подходу код приложения identidock будет работать без каких-либо правок имен хостов.

Группа `redis` идентифицируется по селектору `"name":"redis-pod"`. При наличии нескольких групп с ярлыком `"name":"redis-pod"` этот селектор будет соответствовать всем совпадениям. Если в результате поиска отобрано более одной группы, то сервис выбирает случайную группу для обработки текущего запроса (также

возможна установка критерия *близости* (*affinity*) для выбора группы, например "ClientIP" позволяет постоянно назначать для клиентов конкретные группы по IP-адресам последних). Изменяя ярлыки групп, можно динамически перемещать их в разнообразные выборки, что позволяет выполнять такие задачи, как временный вывод групп контейнеров из режима эксплуатации для отладки или технического обслуживания.

Теперь можно создать контроллер dnmonster и соответствующий сервис почти так же, как это было сделано раньше для Redis. Файл *dnmonster-controller.json* содержит следующие данные:

```
{ "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{ "name":"dnmonster-controller" },
   "spec":{
      "replicas":1,
      "selector":{ "name":"dnmonster-pod" },
      "template":{
         "metadata":{
            "labels":{ "name":"dnmonster-pod" }
         },
         "spec":{
            "containers":[ {
               "name":"dnmonster",
               "image":"amouat/dnmonster:1.0",
               "ports":[ {
                  "containerPort":8080,
                  "protocol":"TCP"
               } ] } ] } } } }
```

Файл для сервиса dnmonster *dnmonster-service.json*:

```
{ "kind":"Service",
   "apiVersion":"v1",
   "metadata":{ "name":"dnmonster" },
   "spec":{
      "ports": [ {
         "port":8080,
         "targetPort":8080,
         "protocol":"TCP"
      } ],
      "selector":{ "name":"dnmonster-pod" }
   }
}
```

Запуск контроллера и сервиса:

```
$ kubectl create -f dnmonster-controller.json
replicationcontrollers/dnmonster-controller
$ kubectl create -f dnmonster-service.json
services/dnmonster
```

Принцип здесь точно такой же, как для контроллера и сервиса redis, в результате мы получили сервис dnmonster, доступный по имени хоста dnmonster, и все запросы перенаправляются к единственному экземпляру dnmonster, созданному контроллером репликации.

Теперь можно создать группу identidock и объединить все компоненты. Создадим файл *identidock-controller.json* со следующим содержимым:

```
{ "kind":"ReplicatinController",
   "apiVersion":"v1",
   "metadata":{ "name":"identidock-controller" },
   "spec":{
      "replicas":1,
      "selector":{ "name":"identidock-pod" },
      "template":{
         "metadata":{
            "labels":{ "name":"identidock-pod" }
         },
         "spec":{
            "containers":[ {
               "name":"identidock",
               "image":"amouat/identidock:1.0",
               "ports":[ {
                  "containerPort":9090,
                  "protocol":"TCP"
               } ] } ] } } } }
```

Запуск контроллера:

```
$ kubectl create -f identidock-controller.json
replicationcontrollers/identidock-controller
```

Приложение identidock теперь работает, но нужно создать соответствующий сервис, чтобы обеспечить доступ к нему из внешнего мира. Для этого создается файл *identidock-service.json* со следующим содержимым:

```
{ "kind":"Service",
   "apiVersion":"v1",
   "metadata":{ "name":"identidock" },
   "spec":{
      "type": "LoadBalancer",
      "ports": [ {
         "port":80,
         "targetPort":9090,
         "protocol":"TCP"
      } ],
   "selector":{ "name":"identidock-pod" }
   }
}
```

Этот сервис немного отличается от предыдущих. Здесь для ключа "type" установлено значение "LoadBalancer", позволяющее создать доступный извне баланси-

ровщик нагрузки, который будет отслеживать соединения с портом 80 и перенаправлять их в сервис identidock на порт 9090.

При использовании GKE также может возникнуть необходимость в открытии порта 80 на сетевом экране. Это можно сделать, создав соответствующее правило с помощью инструмента `gcloud`:

```
$ gcloud compute firewall-rules create --allow=tcp:80 identidock-80
```

Но мы предположим, что в данном примере нет сетевого экрана, поэтому возможно соединение с использованием общедоступного IP-адреса, объявленного для сервиса identidock:

```
$ kubectl get services identidock
NAME         LABELS    SELECTOR              IP(S)            PORT(S)
identidock   <none>    name=identidock-pod   10.111.250.210   80/TCP
                                             23.251.128.247
$ curl 23.251.128.247
<html><head><title>Hello...
```

Тома в Kubernetes

В Kubernetes тома отличаются от томов Docker. Главное различие состоит в том, что тома объявляются на уровне групп (pods), а не на уровне контейнера, и могут совместно использоваться всеми контейнерами группы. Kubernetes предлагает несколько типов томов для различных вариантов использования:

- *emptyDir* – в группе инициализируется пустой каталог, в который могут писать контейнеры данной группы. При удалении группы удаляется и каталог. Это удобно для хранения временных данных, требуемых только в течение жизненного цикла группы, или для данных, которые регулярно копируются в другое, более надежное постоянное хранилище;
- *gcePersistentDisk* – для пользователей GKE. Этот том можно использовать для хранения данных в Google Cloud. Данные сохраняются и после завершения жизненного цикла группы;
- *awsElasticBlockStore* – для пользователей AWS. Этот том можно использовать для хранения данных в Amazon's Elastic Block Store (EBS). Данные сохраняются и после завершения жизненного цикла группы;
- *nfs* – для доступа к файлам, совместно используемым в сетевой файловой системе Network File System (NFS). И в этом случае данные будут сохранены после завершения жизненного цикла группы;
- *secret* – для хранения важной закрытой информации, такой как пароли и API-токены, используемые группой. Тома типа secret обязательно должны создаваться только через Kubernetes API. Они хранятся в файловой системе tmpfs, которая существует исключительно в оперативной памяти и никогда не записывается на диск.

Использование Kubernetes требует немного больше трудозатрат на настройку, но в итоге предоставляет систему, поддерживающую устойчивость к критическим сбоям и балансировку нагрузки непосредственно «из коробки». Вместо контейне-

ров, жестко связанных друг с другом, использование сервисов дает нам дополнительный уровень абстракции, позволяющий с легкостью масштабировать систему и менять местами составляющие ее группы и контейнеры. Недостатком является то, что Kubernetes существенно «утяжелил» наше простое приложение identidock. Дополнительная инфраструктура ведения журналов и контроля требует значительных ресурсов, следовательно, увеличивает издержки при эксплуатации.

Для некоторых приложений неприемлемы дизайн и архитектура системы, навязываемые Kubernetes. Но для большинства приложений, особенно для микросервисов и приложений с четко определенным разделением на небольшие группы контейнеров, Kubernetes предоставляет простое, гибкое и масштабируемое решение, требующее совсем небольшого объема дополнительной работы.

Mesos и Marathon

Apache Mesos (https://mesos.apache.org/) представляет собой менеджер кластеров с открытым исходным кодом. Он предназначен для обеспечения масштабирования очень больших кластеров, содержащих сотни или тысячи хостов. Mesos поддерживает механизм распределения нагрузки при одновременной работе нескольких арендаторов, таким образом, контейнеры Docker одного пользователя могут функционировать рядом с задачами Hadoop другого пользователя.

Проект Apache Mesos начал разрабатываться в Калифорнийском университете в Беркли (University of California, Berkeley) и впоследствии развился во внутреннюю инфраструктуру, используемую для поддержки Twitter, а также стал важным инструментальным средством для многих крупных компаний, таких как eBay и Airbnb. Большая часть дальнейшей разработки Mesos и инструментов поддержки (например, Marathon) выполняется Mesosphere, компанией, основанной Беном Хиндмэном (Ben Hindman), одним из тех, кто начинал разработку проекта Mesos.

Архитектура Mesos ориентирована на обеспечение высокой доступности и эластичности (resilience). Основные компоненты кластера Mesos перечислены ниже:

- *Mesos agent nodes*[1] – узлы-агенты Mesos – отвечают непосредственно за запуск задач. Все агенты предоставляют список своих доступных ресурсов координатору (master). Обычная численность узлов-агентов – от нескольких десятков до нескольких тысяч;
- *Mesos master* – координатор Mesos – отвечает за распределение задач между агентами. Он ведет список доступных ресурсов, которые могут предлагаться фреймворкам. Координатор определяет объем предлагаемых ресурсов на основе стратегии распределения. Обычно создаются два или четыре дополнительных координатора, находящихся в состоянии ожидания и полной готовности к работе в случае возникновения критического сбоя;
- *ZooKeeper* – используется в процедурах голосования и для поиска адреса текущего координатора. Обычно запускаются три или пять экземпляров

[1] Ранее обозначались как «подчиненные узлы» (slave nodes).

ZooKeeper для обеспечения высокой доступности и для оперативной обработки критических сбоев;

- *frameworks – фреймворки* – согласовывают свои действия с координатором для планирования выполнения задач на узлах-агентах. Фреймворки состоят из двух частей: процесс executor, инициализирующий узлы-агенты и контролирующий выполнение задач, и планировщик scheduler, который регистрирует для координатора ресурсы и выбирает их для различных вариантов использования на основе предложений от координатора. В кластере Mesos может быть несколько фреймворков, предназначенных для различных типов задач. Пользователи, желающие передать какую-либо работу для выполнения в кластер, взаимодействуют с фреймворками, а не напрямую с Mesos.

На рис. 12.4 изображен пример кластера Mesos с фреймворком Marathon в качестве планировщика. Планировщик Marathon использует ZooKeeper для обнаружения текущего координатора Mesos для передачи ему задач, требующих выполнения. И для планировщика Marathon, и для координатора Mesos имеются дублеры, находящиеся в состоянии полной готовности для продолжения работы, если вдруг текущий координатор станет недоступным.

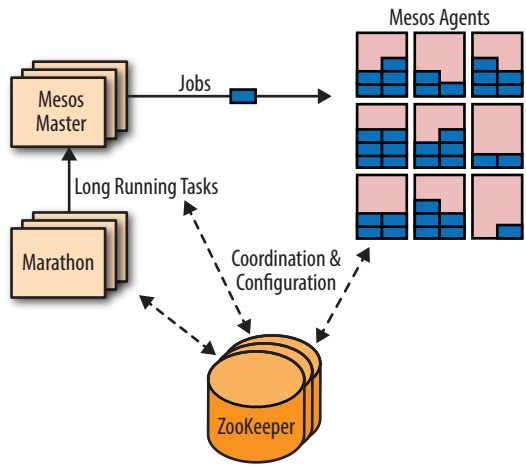

Рис. 12.4. Кластер Mesos

Обычно ZooKeeper запускается на тех же хостах, на которых расположены координатор Mesos и его дублеры. В небольшом кластере на этих же хостах могут запускаться и агенты, но для больших кластеров требуется организация обмена данными с координатором, вследствие чего такое размещение становится не вполне пригодным. Marathon также может размещаться на этих же хостах или работать на отдельных хостах, расположенных на границе сети, чтобы обеспечить точки доступа для клиентов, таким образом изолируя кластер Mesos от пользователей.

Marathon (https://mesosphere.github.io/marathon/) (продукт компании Mesosphere) предназначен для запуска, контроля и масштабирования приложений с длительным жизненным циклом[1]. Marathon обеспечивает гибкость при запуске приложений и может даже использоваться для запуска вспомогательных фреймворков, таких как Chronos (специальная версия демона cron для центров данных). Он предоставляет возможность выбора наиболее подходящего фреймворка для запускаемых контейнеров Docker, для которых в Marathon обеспечена прямая поддержка. Подобно другим инструментам оркестровки, рассмотренным ранее, Marathon поддерживает разнообразные правила определения близости и наборы ограничений. Клиенты взаимодействуют с Marathon через REST API. Среди других функциональных возможностей следует отметить поддержку процедур проверки работоспособности и потоки событий, которые можно использовать для объединения с балансировщиками нагрузки или для анализа метрик.

Чтобы лучше понять, как работают Mesos и Marathon, с помощью Docker Machine создадим кластер из трех узлов, в некоторой степени соответствующий схеме на рис. 12.4. Но в нашем кластере будут работать только один экземпляр ZooKeeper, один планировщик Marathon и один координатор Mesos. Агенты Mesos запускаются на всех узлах. Разумеется, реальная архитектура для эксплуатации будет совсем другой: в нее должны быть включены дополнительные экземпляры основных сервисов для обеспечения высокой доступности.

Начнем с создания хостов `mesos-1`, `mesos-2` и `mesos-3`:

```
$ docker-machine create -d virtualbox mesos-1
Creating VirtualBox VM...
...
$ docker-machine create -d virtualbox mesos-2
...
$ docker-machine create -d virtualbox mesos-3
...
```

Также необходимо выполнить некоторые действия для создания конфигурации, позволяющей правильно определять имена хостов кластера Mesos. Это не обязательно, но без такой настройки иногда возникают проблемы:

```
$ docker-machine ssh mesos-1 'sudo sed -i "\$a127.0.0.1 mesos-1" /etc/hosts'
$ docker-machine ssh mesos-2 'sudo sed -i "\$a127.0.0.1 mesos-2" /etc/hosts'
$ docker-machine ssh mesos-3 'sudo sed -i "\$a127.0.0.1 mesos-3" /etc/hosts'
```

Начнем с создания конфигурации для хоста `mesos-1`, на котором будут работать координатор Mesos, ZooKeeper, фреймворк Marathon и агент.

В первую очередь нужно запустить ZooKeeper, который другие контейнеры будут использовать для регистрации и поиска сервисов. Здесь используется образ, который я создал при написании этой главы, а не официальный образ. Для этого контейнера задается аргумент `--net=host`, в основном для обеспечения эффективно-

[1] Вероятно, именно поэтому этот инструмент и получил название «марафон». Та-дааа! Это шутка.

сти и для согласованности с контейнерами координатора и агента, для которых требуется сетевая среда хоста, чтобы они могли открывать новые порты для сервисов.

```
$ eval $(docker-machine env mesos-1)
$ docker run --name zook -d --net=host amouat/zookeeper
...
Status: Downloaded newer image for amouat/zookeeper:latest
dfc27992467c9563db05af63ecb6f0ec371c03728f9316d870bd4b991db7b642
```

Чтобы упростить следующие команды создания конфигурации, сохраним IP-адреса созданных узлов в соответствующих переменных:

```
$ MESOS1=$(docker-machine ip mesos-1)
$ MESOS2=$(docker-machine ip mesos-2)
$ MESOS3=$(docker-machine ip mesos-3)
```

После этого можно запустить контейнер координатора:

```
$ docker run --name master -d --net=host \
             -e MESOS_ZK=zk://$MESOS1:2181/mesos \  ❶
             -e MESOS_IP=$MESOS1 \  ❷
             -e MESOS_HOSTNAME=$MESOS1 \
             -e MESOS_QUORUM=1 \  ❸
             mesosphere/mesos-master:0.23.0-1.0.ubuntu1404  ❹
...
Status: Downloaded newer image for mesosphere/mesos-master:0.23.0-1.0.ubuntu1404
9de83f40c3e1c5908381563fb28a14c2e23bb6faed569b4d388ddfb46f7d7403
```

❶ Сообщение координатору, где найти ZooKeeper и зарегистрироваться.
❷ Установка IP-адреса, используемого для координаторов.
❸ В этом примере создается только один узел координатора – для упрощения.
❹ Используется образ `mesos` от компании Mesosphere. Образ не формируется автоматически, и трудно сказать, что у него внутри, поэтому я не рекомендую использовать его в реальном эксплуатационном режиме.

Запуск агента на том же хосте:

```
$ docker run --name agent -d --net=host \
             -e MESOS_MASTER=zk://$MESOS1:2181/mesos \
             -e MESOS_CONTAINERIZERS=docker \  ❶
             -e MESOS_IP=$MESOS1 \
             -e MESOS_HOSTNAME=$MESOS1 \
             -e MESOS_EXECUTOR_REGISTRATION_TIMEOUT=10mins \  ❷
             -e MESOS_RESOURCES="ports(*):[80-32000]" \  ❸
             -e MESOS_HOSTNAME=$MESOS1 \
             -v /var/run/docker.sock:/run/docker.sock \  ❹
             -v /usr/local/bin/docker:/usr/bin/docker \
             -v /sys:/sys:ro \  ❺
             mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
...
Status: Downloaded newer image for mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
38aaec1d08a41e5a6deeb62b7b097254b5aa2b758547e03c37cf2dfc686353bd
```

❶ В Mesos существует концепция *контейнеризаторов* (*containerizers*), обеспечивающих изоляцию задач друг от друга и запускаемых на узле-агенте. Здесь добавление аргумента `docker` позволяет использовать контейнеры Docker как задачи на узле-агенте.
❷ Необходимо увеличить «тайм-аут регистрации», чтобы предоставить агенту время для загрузки образов, прежде чем он будет удален.
❸ По умолчанию агенты предлагают для фреймворков только небольшое подмножество портов с большими номерами. Так как identidock использует некоторые порты с малыми номерами, необходимо явно добавить их в предлагаемые ресурсы. Для краткости я не стал удалять из этого списка порты, используемые Mesos, но это может привести к конфликту, если фреймворк запросит уже использующийся порт.
❹ Для того чтобы агент мог запускать новые контейнеры, монтируются `docker.sock` и бинарный файл `docker`.
❺ Монтирование каталога */sys* необходимо, чтобы агент сообщал точные и подробные данные о ресурсах, доступных на этом хосте.

```
$ docker run -d --name marathon -p 9000:8080 \  ❶
             mesosphere/marathon:v0.9.1 --master zk://$MESOS1:2181/mesos \
                 --zk zk://$MESOS1:2181/marathon \
                 --task_launch_timeout 600000  ❷
...
Status: Downloaded newer image for mesosphere/marathon:v0.9.1
697d78749c2cfd6daf6757958f8460963627c422710f366fc86d6fcdce0da311
```

❶ Необходимо назначить для Marathon другой порт вместо порта по умолчанию 8080, чтобы избежать возможного конфликта с контейнером dnmonster.
❷ Установка тайм-аута длиной 600 000 мс, соответствующего 10-минутному тайм-ауту при регистрации агента под управлением executor. Этот параметр предназначен для устранения временной проблемы и будет удален в следующей версии.

Далее следует инициализация агентов на других хостах:

```
$ eval $(docker-machine env mesos-2)
$ docker run --name agent -d --net=host \
             -e MESOS_MASTER=zk://$MESOS1:2181/mesos \
             -e MESOS_CONTAINERIZERS=docker \
             -e MESOS_IP=$MESOS2 \
             -e MESOS_HOSTNAME=$MESOS2 \
             -e MESOS_EXECUTOR_REGISTRATION_TIMEOUT=10mins \
             -e MESOS_RESOURCES="ports(*):[80-32000]" \
             -v /var/run/docker.sock:/run/docker.sock \
             -v /usr/local/bin/docker:/usr/bin/docker \
             -v /sys:/sys:ro \
             mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
...
Status: Downloaded newer image for mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
ac1216e7eedbb39475404f45a5655c7dc166d118db99072ed3d460322ad1a1c2
$ eval $(docker-machine env mesos-3)
$ docker run --name agent -d --net=host \
             -e MESOS_MASTER=zk://$MESOS1:2181/mesos \
             -e MESOS_CONTAINERIZERS=docker \
```

```
            -e MESOS_IP=$MESOS3 \
            -e MESOS_HOSTNAME=$MESOS3 \
            -e MESOS_EXECUTOR_REGISTRATION_TIMEOUT=10mins \
            -e MESOS_RESOURCES="ports(*):[80-32000]" \
            -v /var/run/docker.sock:/run/docker.sock \
            -v /usr/local/bin/docker:/usr/bin/docker \
            -v /sys:/sys:ro \
            mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
...
Status: Downloaded newer image for mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
b5eecb7f56903969d1b7947144617050f193f20bb2a59f2b8e4ec30ef4ec3059
```

Если в браузере открыть адрес *http://$MESOS1:5050* (заменив *$MESOS1* на IP-адрес хоста `mesos-1`), то вы должны увидеть веб-интерфейс для Mesos. А интерфейс для Marathon должен быть доступен через порт 9000.

Теперь у нас есть готовая инфраструктура для запуска контейнеров на агентах Mesos через Marathon. Но перед запуском identidock необходимо добавить механизм обнаружения сервисов. Для данного примера воспользуемся mesos-dns (https://github.com/mesosphere/mesos-dns), который будет работать на хосте `mesos-1`.

Задания Marathon определяются в JSON-файле, который содержит подробную информацию, необходимую для выполнения каждого задания, а также информацию о требуемых ресурсах.

Для запуска `mesos-dns` на хосте `mesos-1` можно использовать следующий JSON-файл:

```
{
  "id": "mesos-dns",
  "container": {
    "docker": {
      "image": "bergerx/mesos-dns",  ❶
      "network": "HOST",  ❷
      "parameters": [
          { "key": "env",
            "value": "MESOS_DNS_ZK=zk://192.168.99.100:2181/mesos" },  ❸
          { "key": "env", "value": "MESOS_DNS_MASTERS=192.168.99.100:5050" },
          { "key": "env", "value": "MESOS_DNS_RESOLVERS=8.8.8.8" }
        ]
    }
  },
  "cpus": 0.1,  ❹
  "mem": 120.0,
  "instances": 1,  ❺
  "constraints": [["hostname", "CLUSTER", "192.168.99.100"]]  ❻
}
```

❶ Здесь применяется пользовательская версия mesos-dns, поскольку она автоматически считывает конфигурацию из переменных среды, а официальный образ *mesosphere/mesos-dns* во время написания книги эту возможность не поддерживал.

❷ И здесь по соображениям обеспечения эффективности имеет смысл использовать тип сети `host`, хотя можно было бы установить тип сети `bridge` и открыть порт 53.
❸ Установка переменных среды для конфигурирования mesos-dns. Это делается с помощью ключа `parameter`, который добавляет флаги в команду `docker run`, используемую для запуска данного образа. IP-адрес `192.168.99.100` следует заменить на реальный IP-адрес хоста `mesos-1` в вашем кластере.
❹ Для всех задач необходимо определить ресурсы, обеспечивающие их выполнение. В данном случае запрашиваются `"0.1"` ресурса процессора и 120 мегабайт оперативной памяти.
❺ Для текущего теста достаточно одного экземпляра mesos-dns.
❻ Привязка `mesos-dns` к хосту `mesos-1` посредством определения ограничения по имени хоста с указанием IP-адреса `mesos-1`.

Точное распределение ресурсов для контейнеров зависит от конфигурируемого агента *isolator*, используемого Mesos. Обычно для «процессора» (т. е. CPU в конфигурации Mesos) определяется весовой коэффициент, таким образом, при возникновении конкуренции контейнеру с весовым коэффициентом 0,2 процессор будет предоставляться в два раза чаще, чем контейнеру с весовым коэффициентом 0,1. Этот агент также управляет количеством процессоров, вычитая затребованное значение из общего ресурса, которым располагает Mesos (если в распоряжении агента находятся `"8"` процессоров и запускается задача с использованием `"1"` процессора, то для прочих задач агент предложит `"7"` процессоров).

Сохраните приведенный выше текст в файле *dns.json* и передайте его в Marathon, используя REST API и следующую команду:

```
$ curl -X POST http://$MESOS1:9000/v2/apps -d @dns.json \
       -H "Content-type: application/json" | jq .
{
  "id": "/mesos-dns",
  "cmd": null,
  "args": null,
  "user": null,
...
```

Для передачи заданий также можно было бы воспользоваться инструментом командной строки `marathonctl` или веб-интерфейсом.

Если теперь взглянуть на веб-интерфейс Marathon, то вы должны увидеть приложение `mesos-dns` в процессе развертывания. После приведения `mesos-dns` в полную готовность необходимо сообщить ему данные обо всех используемых хостах. Проще всего отредактировать файлы *resolv.conf* на каждом хосте, и необходимые данные будут автоматически передаваться во все контейнеры при запуске.

Это можно сделать, выполняя соответствующую команду с применением утилиты `sed` на каждом хосте:

```
$ docker-machine ssh mesos-1 \
  "sudo sed -i \"1s/^/domain marathon.mesos\nnameserver $MESOS1\n/\" /etc/resolv.conf"
$ docker-machine ssh mesos-2 \
  "sudo sed -i \"1s/^/domain marathon.mesos\nnameserver $MESOS1\n/\" /etc/resolv.conf"
$ docker-machine ssh mesos-3 \
  "sudo sed -i \"1s/^/domain marathon.mesos\nnameserver $MESOS1\n/\" /etc/resolv.conf"
```

В результате содержимое файла *resolv.conf* на каждом хосте должно выглядеть приблизительно так:

```
domain marathon.mesos
nameserver 192.168.99.100
...
```

Если в файле *resolv.conf* есть строка `search`, то ее также необходимо передать для включения в файл *marathon.mesos*, иначе процедура определения имен не будет выполняться.

Теперь создадим файлы для запуска каждого контейнера. Мы поместим их в сеть типа `bridge`, но для них потребуется явное объявление порта, поэтому доступ к ним возможен через хост.

Как обычно, начнем с Redis. Содержимое файла *redis.json*:

```
{
    "id": "redis",
    "container": {
      "docker": {
        "image": "redis:3",
        "network": "BRIDGE",
        "portMappings": [
          {"containerPort": 6379, "hostPort": 6379}
        ]
      }
    },
    "cpus": 0.3,
    "mem": 300.0,
    "instances": 1
}
```

Передача файла в Mesos:

```
$ curl -X POST http://$MESOS1:9000/v2/apps -d @redis.json \
       -H "Content-type: application/json"
...
```

Для dnmonster создается такой же файл *dnmonster.json*:

```
{
    "id": "dnmonster",
    "container": {
      "docker": {
        "image": "amouat/dnmonster:1.0",
        "network": "BRIDGE",
        "portMappings": [
          {"containerPort": 8080, "hostPort": 8080}
        ]
      }
    },
```

```
    "cpus": 0.3,
    "mem": 200.0,
    "instances": 1
}
```

Передача файла в Mesos:

```
$ curl -X POST http://$MESOS1:9000/v2/apps -d @dnmonster.json \
       -H "Content-type: application/json"
...
```

И последний файл *identidock.json*:

```
{
    "id": "identidock",
    "container": {
      "docker": {
        "image": "amouat/identidock:1.0",
        "network": "BRIDGE",
        "portMappings": [
          {"containerPort": 9090, "hostPort": 80}
        ]
      }
    },
    "cpus": 0.3,
    "mem": 200.0,
    "instances": 1
}
```

Передача файла в Mesos:

```
$ curl -X POST http://$MESOS1:9000/v2/apps -d @identidock.json \
       -H "Content-type: application/json"
...
```

После загрузки и запуска агентами соответствующих образов должна появиться возможность доступа к identidock через IP-адрес любого хоста, которому была назначена задача identidock. Это можно проверить с помощью веб-интерфейса или с помощью REST API, например:

```
$ curl -s http://$MESOS1:9000/v2/apps/identidock | jq '.app.tasks[0].host'
"192.168.99.101"
$ curl 192.168.99.101
<html><head><title>Hello...
```

Marathon обеспечивает перезапуск любых остановленных приложений при наличии достаточных для этого ресурсов. В качестве учебного эксперимента попробуем остановить и перезапустить узлы `mesos-2` и `mesos-3`, чтобы наблюдать перемещение и перезапуск наших задач в условиях изменения доступности ресурсов.

В приложения можно без затруднений добавлять более сложные процедуры проверки работоспособности, обычно с применением протокола HTTP для точки

входа, которую Marathon может опрашивать через постоянные интервалы времени. Например, содержимое файла *identidock.json* можно изменить следующим образом:

```
{
    "id": "identidock",
    "container": {
    "docker": {
      "image": "amouat/identidock:1.0",
      "network": "BRIDGE",
      "portMappings": [
        {"containerPort": 9090, "hostPort": 80}
      ]
    }
    },
    "cpus": 0.3,
    "mem": 200.0,
    "instances": 1,
    "healthChecks": [
    {
      "protocol": "HTTP",
      "path": "/",
      "gracePeriodSeconds": 3,
      "intervalSeconds": 10,
      "timeoutSeconds": 10,
      "maxConsecutiveFailures": 3
    }]
}
```

Здесь определено повторение попытки получить домашнюю страницу приложения identidock с интервалом в 10 секунд. Если по каким-либо причинам попытка неудачна (например, код возврата выходит за пределы диапазона 200–399 или попытка прервана по тайм-ауту), то Marathon удвоит процедуры тестирования точки входа, прежде чем принудительно завершить данную задачу.

Для инициализации процедуры проверки работоспособности нужно просто остановить старый экземпляр identidock и запустить его с обновленной конфигурацией:

```
$ curl -X DELETE http://$MESOS1:9000/v2/apps/identidock
{"version":"2015-09-02T13:53:23.281Z","deploymentId":"1db18cce-4b39-49c0-8f2f...
$ curl -X POST http://$MESOS1:9000/v2/apps -d @identidock.json \
       -H "Content-type: application/json"
...
```

После этого при переходе к веб-интерфейсу Marathon вы должны обнаружить пункт **Health Check Results** (Результаты проверки работоспособности).

Ранее мы уже видели действие ограничений в Marathon при планировании работы контейнера `mesos-dns` на хосте с заданным IP-адресом. Также можно опре-

делить ограничения для выбора хостов с заданными атрибутами (или без оных), чтобы обеспечить возможность распределения контейнеров по хостам для повышения устойчивости к критическим сбоям.

В текущей конфигурации есть одна проблема: адрес сервиса identidock зависит от IP-адреса хоста, с которым он связан. Очень важно иметь в своем распоряжении способ надежного определения маршрута к сервису identidock из статической точки входа. Это можно было бы сделать, используя `mesos-dns` для обнаружения текущей точки входа, но Marathon, кроме всего прочего, предоставляет инструмент servicerouter (https://github.com/mesosphere/marathon/blob/master/bin/servicerouter.py), который генерирует файл конфигурации HAProxy (http://www.haproxy.org/) для определения маршрутов к приложениям Marathon. Другое решение заключается в развертывании собственного прокси-сервера или сервиса балансировки нагрузки, который отслеживает в потоке событий Marathon события создания и удаления приложений и перенаправляет запросы соответствующим образом.

Вторая проблема – использование жестко заданных портов, как, например, 6379 для Redis и 8080 для dnmonster. Выше нам уже пришлось переключить интерфейс Marathon на порт 9000, чтобы избежать конфликта с dnmonster. Эту проблему можно было бы устранить, переписав приложение так, чтобы в нем использовалось динамическое назначение портов механизмом Marathon, но более удачным решением выглядит использование программно-конфигурируемой сети (SDN – Software-Defined Networking). В Интернете можно найти несколько руководств, описывающих установку Weave для совместной работы с Mesos, кроме того, компания Mesosphere работает над проектом полной интеграции Project Calico (см. раздел «Project Calico» в главе 11) с Mesos.

Marathon также предлагает возможность создания *групп приложений* (*application groups*), которые можно использовать для объединения приложений, чтобы обеспечить их развертывание в правильном порядке с учетом удовлетворения зависимостей (например, база данных будет развернута раньше сервера приложений) как при первоначальном запуске, так и в процессе масштабирования или выполнения процедур роллинг-релиза.

Главной особенностью Mesos, отличающей его от других решений оркестрации, является поддержка смешанных типов нагрузки. В одном кластере может работать несколько фреймворков, обеспечивая выполнение задач по обработке данных Hadoop или Storm, вместе с контейнерами Docker, управляющими микросервисным приложением. Это одно из тех свойств, которые делают Mesos особенно полезным для достижения высокой эффективности использования ресурсов: предоставляется возможность планирования на одном хосте задач с интенсивным использованием процессоров и низким требованием к пропускной способности сети вместе с задачами, имеющими противоположные характеристики, для максимального задействования ресурсов. Эффективное использование кластера зависит от точного указания запрашиваемых ресурсов, а не от избыточного их резервирования при распределении задач для Marathon или других фреймворков. В примере приложения identidock отслеживается повышенное потребление оперативной памяти,

чтобы не допускать назначения всех задач одному агенту, хотя в реальной обстановке один хост мог бы справиться с решением этой проблемы. Для устранения подобных затруднений Mesos поддерживает *овербукинг* (*over-subscription*), позволяющий запускать «отменяемые» задачи на агенте, который в текущий момент не предлагает достаточного объема ресурсов, но в соответствии с наблюдаемыми контрольными данными продолжает обладать достаточной мощностью и производительностью. Такие отменяемые задачи будут остановлены, если нагрузка на ресурсы станет максимальной, поэтому обычно таким задачам присваивается низкий приоритет, как и аналитическим задачам, работающим в фоновом режиме. Более подробное описание овербукинга в Mesos можно найти на сайте Mesos (http://mesos.apache.org/documentation/latest/oversubscription/) и на сайте GitHub (https://github.com/mesosphere/serenity).

Запуск Swarm или Kubernetes в Mesos

Поскольку Mesos предоставляет инфраструктуру кластеризации и планирования низкого уровня, существует возможность использования интерфейсов более высокого уровня, таких как Kubernetes и Swarm, поверх Mesos. На первый взгляд это может показаться неразумным – слишком высока степень дублирования функциональности. Тем не менее работа поверх Mesos позволяет воспользоваться всеми его преимуществами для обеспечения устойчивости к критическим сбоям, высокой доступности и рационального использования ресурсов. Кроме того, следует учесть и мобильность, позволяющую легко перенести систему в любой центр данных или облачную среду, поддерживающую Mesos, сохраняя при этом все функциональные возможности и простоту использования Kubernetes или Swarm. Значительные преимущества получают и инженеры по эксплуатации: они могут сосредоточиться на сопровождении низкоуровневой инфраструктуры Mesos для поддержки функций распределения нагрузки и рационального использования ресурсов.
Более подробную информацию об этом см.: https://github.com/mesosphere/kubernetes-mesos.

Платформы управления контейнерами

Существует несколько платформ, специально предназначенных для автоматизации задач по развертыванию, оркестрации и управлению контейнерами. Эти платформы не обеспечивают напрямую хостинга для контейнеров, вместо этого они предоставляют внешний интерфейс для публичных облаков и закрытой частной инфраструктуры. Во всех примерах, приводимых ниже, рассматривается веб-интерфейс и дается общий обзор конкретной системы. Описываемые платформы могут существенно упростить управление развертыванием контейнеров, создавая дополнительный уровень абстракции от инфраструктуры конкретного провайдера (провайдеров) и облегчая процедуры перемещения между различными облачными платформами или использование нескольких облачных платформ одновременно.

Rancher

Rancher (http://rancher.com/rancher/) – платформа управления контейнерами, в наибольшей степени ориентированная на использование Docker, по сравнению с другими платформами, предположительно получила свое название в честь известного сравнения «домашние животные против стада коров»[1].

Начать работу с Rancher очень просто: запустите контейнер сервера Rancher на одном из хостов:

```
$ docker run -d --restart=always -p 8080:8080 rancher/server
```

Если после этого зайти браузером на порт 8080 этого хоста, то вы должны увидеть интерфейс Rancher. Начиная с этого момента, появляется возможность добавлять хосты – виртуальные машины из собственной инфраструктуры или облачные ресурсы. При использовании открытой облачной среды, такой как Digital Ocean или AWS, Rancher автоматически подготовит виртуальную машину, если ему предоставлены соответствующие ключи доступа. При установке Rancher на существующей виртуальной машине простейшим способом запуска агента Rancher на том же хосте с использованием аргументов, полученных от внешнего интерфейса Rancher, является следующий вариант:

```
$ docker run -d --privileged -v /var/run/docker.sock:/var/run/docker.sock \
    rancher/agent:v0.8.1 http://<host_ip>:8080/v1/scripts/<token>
```

Монтирование сокета Docker необходимо для того, чтобы Rancher мог запускать новые контейнеры на этом хосте. После запуска агента и приведения его в рабочее состояние он должен появиться на вкладке **HOSTS** в интерфейсе Rancher. Экран инфраструктуры показывает список всех работающих контейнеров на хостах (за исключением агентов Rancher). При тестировании сервер Rancher можно разместить на том же хосте, что и агент, но в режиме реальной эксплуатации рекомендуется выделить отдельный хост для сервера.

Запуск и поддержка работы приложения identidock в Rancher представляют собой тривиальную задачу. Нужно создать сервис для каждого контейнера и установить соединения сервиса identidock с сервисами dnmonster и Redis. Для сервиса identidock также можно опубликовать порт 9000 как порт 80. Для других портов в такой публикации нет необходимости, так как Rancher берет на себя ответственность по организации сетевой среды для хостов и по распределению контейнеров по соответствующим хостам. На рис. 12.5 показана работа приложения identidock в кластере из двух хостов, управляемом Rancher.

Можно было бы воспользоваться Rancher Compose (https://github.com/rancher/rancher-compose), инструментом командной строки для развертывания файлов Docker Compose в Rancher.

Rancher упрощает наблюдение за подробностями работы контейнеров, включая доступ к журналам и даже запуск командной оболочки для отладки.

---

[1] Rancher в пер. с англ. – фермер-скотовод.

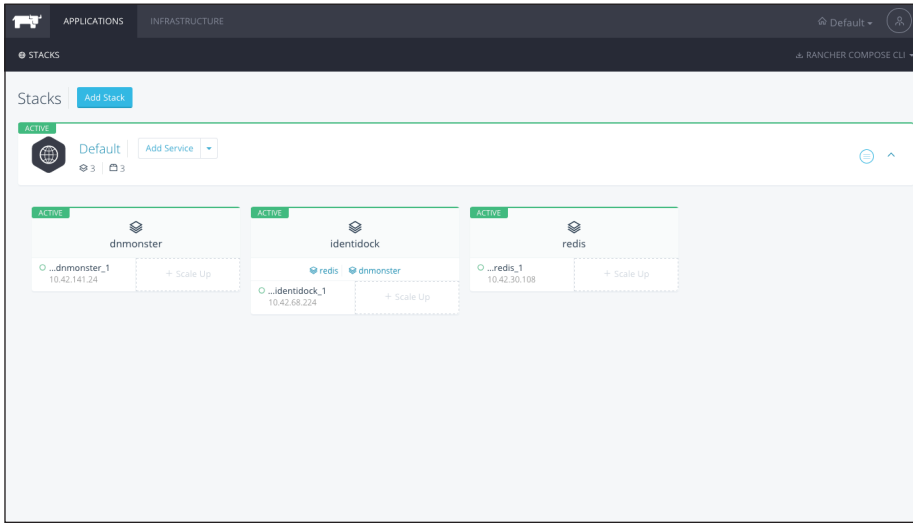

Рис. 12.5. Приложение identidock, работающее под управлением Rancher

Несмотря на то что в настоящее время Rancher предлагает собственное решение по организации сетевой среды, объединяющей множество хостов (с использованием IPsec (https://en.wikipedia.org/wiki/IPsec; https://ru.wikipedia.org/wiki/IPsec)), с обнаружением сервисов и простой оркестровкой, его основная идея заключается в полном внедрении в стек Docker (то есть использование Swarm для оркестровки, а также применение новых сетевых подключаемых модулей). Кроме того, авторы Rancher работают над интеграцией с Kubernetes и Mesos. Все эти разработки заслуживают внимания, если учесть, насколько просто начать работу с Rancher и добавить его в существующую систему.

Clocker

Clocker (https://brooklyncentral.github.io/clocker/) – самодостаточная платформа для управления контейнерами с открытым исходным кодом, работающая поверх Apache Brooklyn (https://brooklyn.incubator.apache.org/). По сравнению с Rancher, это решение в большей степени ориентировано на приложения, эта платформа поддерживает использование различных типов виртуальных машин и контейнеров внутри одного приложения. Кроме того, Clocker обеспечивает возможность работы с многими провайдерами облачных сред, используя для этого инструментальный пакет jclouds (https://jclouds.apache.org/).

Для начала работы с Clocker потребуются некоторые подготовительные действия. После загрузки дистрибутивного пакета необходимо определить конфигурацию токенов и ключей доступа для развертывания в облачной среде. После этого можно запустить Clocker, и программа автоматически подготовит хосты и создаст соответствующую сетевую среду, используя для этого Weave или Project Calico.

После создания работающей облачной среды Clocker можно сформировать новое приложение для запуска контейнеров Docker. Для обеспечения работы identidock в Clocker предназначен следующий YAML-файл:

```
id: identidock
name: "Identidock"
location: my-docker-cloud
services:
- type: docker:redis:3
  name: redis
  id: redis
  openPorts:
  - 6379
- type: docker:amouat/dnmonster:1.0
  name: dnmonster
  id: dnmonster
  openPorts:
  - 8080
- type: docker:amouat/identidock:1.0
  name: identidock
  id: identidock
  portBindings:
    80: 9090
  links:
  - $brooklyn:component("redis")
  - $brooklyn:component("dnmonster")
```

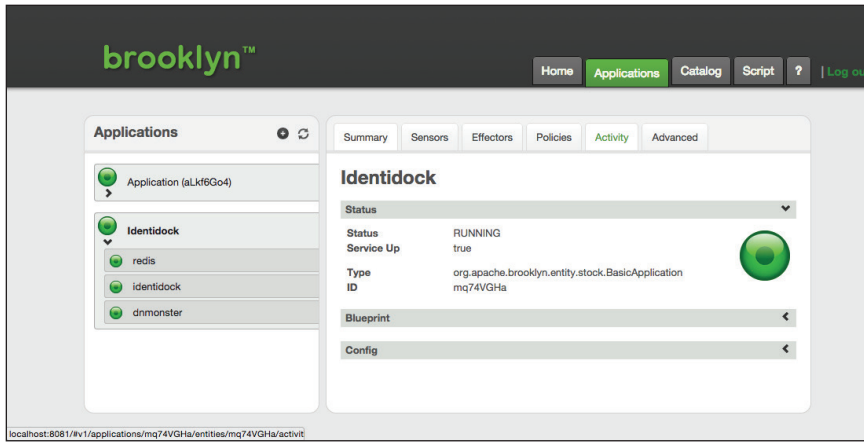

Рис. 12.6. Clocker управляет работой identidock, используя для этого Weave и AWS

Основное преимущество Clocker состоит в том, что он может без проблем совместно использовать Docker-системы и неконтейнеризованные ресурсы. Напри-

мер, можно заменить контейнер Redis на виртуальную машину, на которой работает Redis, использовав для сервиса Redis следующий YAML-файл:

```
- type: org.apache.brooklyn.entity.nosql.redis.RedisStore
  name: redis
  location: jclouds:softlayer:lon02
  id: redis
  install.version: 3.0.0
  start.timeout: 10m
```

Здесь будет предоставлена виртуальная машина в центре обработки данных Softlayer в Лондоне, затем Redis будет установлен и запущен на ней.

Во время написания книги Clocker активно разрабатывается. Несмотря на некоторые текущие недостатки, использование технологий Brooklyn и jcloud означает, что Clocker может быть развернут на широком спектре различных систем и успешно работать с любыми комбинациями разнообразных типов инфраструктур.

Tutum

Tutum (https://www.tutum.co/) предоставляет базовую платформу для развертывания и управления контейнеров. Основное внимание сосредоточено на удобстве использования и понятном, дружелюбном интерфейсе.

Добавление узлов в Tutum может выполняться с помощью задания данных учетной записи в публичном облаке или посредством установки агента Tutum на существующий узел. Агент запускается как демон, а не как контейнер, таким образом, он поддерживается не всеми операционными системами (особенно критично отсутствие поддержки образов `boot2docker`, созданных с помощью Docker Machine).

Tutum использует *файлы стека* (*stackfiles*) для определения наборов связанных сервисов. Эти файлы очень похожи на файлы Compose, но в них добавлено несколько дополнительных полей, относящихся к оркестрации и масштабированию, например `target_num_containers` и `deployment_strategy`, при этом некоторые поля исключены, например `user` и `cap_add`.

В качестве внутреннего механизма Tutum использует Weave (см. раздел «Weave» главы 11) для поддержки сетевой среды со многими хостами и механизма обнаружения сервисов.

Обеспечить работу приложения identidock в Tutum очень легко. На рис. 12.7 показана панель управления Tutum, где identidock работает на двух узлах Digital Ocean.

Кроме веб-интерфейса, к Tutum можно получить доступ через REST API и с помощью интерфейса командной строки Tutum.

Для всех, кому необходим централизованный сервис, который позволяет избавиться от больших объемов работ по настройке и запуску контейнеризованных сервисов, Tutum вполне подходит. Но если требуются полный контроль и детальное управление сервисами или вы недостаточно доверяете централизованным сервисам, то следует поискать другое решение.

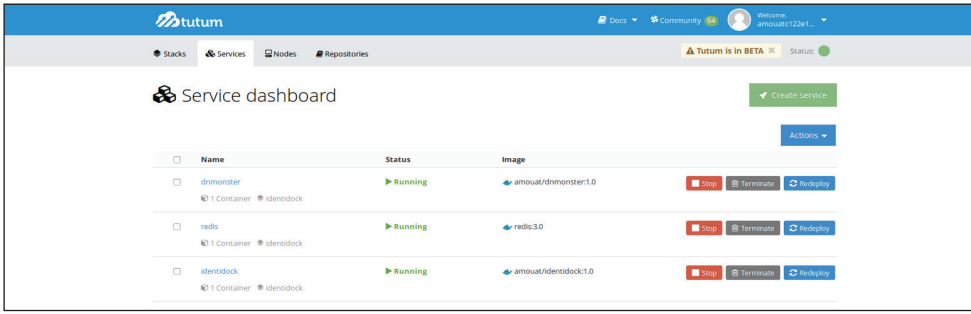

Рис. 12.7. Обеспечение работы identidock в Tutum

Резюме

Существует большое количество вариантов выбора решений по оркестрации, кластеризации и управлению контейнерами. Вообще говоря, каждый вариант имеет собственные очевидные отличия от других. С точки зрения оркестрации можно отметить следующее:

- преимущество (оно же является недостатком) Swarm состоит в использовании стандартного интерфейса Docker. Это существенно упрощает применение Swarm и его интеграцию в существующие рабочие потоки, но при этом возникают трудности при поддержке более сложных схем распределения, которые могут определяться в нестандартных интерфейсах;
- Fleet представляет собой относительно простой механизм оркестрации низкого уровня, который можно использовать как основу для инструментов оркестрации более высокого уровня, таких как Kubernetes или систем, разрабатываемых по заказу;
- Kubernetes является самодостаточным инструментом оркестрации, в который включены механизмы обнаружения сервисов и репликации. Может потребоваться некоторое перепроектирование существующих приложений, но при правильном использовании этот инструмент в результате дает устойчивую к критическим сбоям и масштабируемую систему;
- проверенный годами инструмент управления низкого уровня Mesos поддерживает несколько фреймворков, предназначенных для оркестрации контейнеров, включая Marathon, Kubernetes и Swarm.

Во время написания книги Kubernetes и Mesos являлись более тщательно проработанными и стабильными инструментами, чем Swarm. С точки зрения обеспечения масштабируемости Mesos вполне доказал свою пригодность для поддержки крупных систем, состоящих из сотен и даже тысяч узлов. Но для небольших кластеров, состоящих, например, из менее чем десятка узлов, Mesos выглядит излишне усложненным решением.

Если рассматривать системы управления, то Rancher наилучшим образом подходит для систем, состоящих исключительно из контейнеров Docker. Его можно легко добавлять в существующие конфигурации и удалять из них, поэтому даже пробное его применение связано с минимальным риском.

Глава 13

Обеспечение безопасности контейнеров и связанные с этим ограничения

Для правильного использования Docker следует хорошо знать все потенциальные проблемы, связанные с угрозой безопасности, и основные инструментальные средства и методики защиты систем, основанных на контейнерах. В этой главе рассматривается задача обеспечения безопасности с точки зрения применения Docker в реальной эксплуатационной среде, но большинство рекомендаций вполне применимо и в процессе разработки. Даже при наличии системы защиты важно сохранять среды разработки и эксплуатации в одинаковом состоянии, чтобы избежать проблем, связанных с переносом кода между средами, для которых создается Docker-система.

При чтении онлайн-публикаций и новостей[1] о Docker может сложиться впечатление, что Docker по своей природе небезопасен и не готов к промышленной эксплуатации. А о проблемах, связанных с безопасным применением контейнеров, скажу лишь, что при правильном использовании контейнеры могут обеспечить более безопасную и эффективную работу, чем виртуальные машины или аппаратные решения.

В начале главы рассматриваются некоторые вопросы, касающиеся безопасности систем, основанных на контейнерах, о которых следует серьезно задуматься при использовании контейнеров.

Внимание!
Материал этой главы основан на моем личном мнении. Я не являюсь исследователем проблем безопасности и не отвечаю за безопасность какой-либо крупной

[1] Полезные материалы по обеспечению безопасности в Docker: серия статей Дэна Уолша (Dan Walsh) из Red Hat на opensource.com (https://opensource.com/business/14/7/docker-security-selinux) и статья Джонатана Руденберга (Jonathan Rudenberg) об угрозах для безопасности образов (https://titanous.com/posts/docker-insecurity), но следует отметить, что статья Джонатана в большей степени касается проблем разработки дайджестов и проекта Notary.

открытой системы. Но должен сказать: я уверен в том, что любая система, для которой выполняются рекомендации из этой главы, будет защищена лучше, чем большинство прочих систем. Советы и рекомендации из этой главы не являются полноценным решением, их следует использовать как основу для разработки собственной стратегии и конкретных процедур по обеспечению безопасности.

На что следует обратить особое внимание

О каких угрозах для безопасности следует подумать в первую очередь при создании среды, основанной на контейнерах? Следующий список не претендует на полноту, но он должен дать вам пищу для размышлений:

- *уязвимости в ядре* – в отличие от виртуальной машины, ядро совместно используется всеми контейнерами и самим хостом, что увеличивает степень влияния любых уязвимостей, имеющихся в ядре. Если контейнер способен привести ядро в аварийное состояние (kernel panic), это выведет из строя весь хост. В виртуальных машинах ситуация намного лучше: атакующий должен сначала пройти через ядро виртуальной машины и гипервизор, прежде чем получит доступ к ядру хоста;
- *атаки типа DoS* (*Denial-of-Service*) – все контейнеры совместно используют ресурсы ядра. Если один контейнер способен монополизировать доступ к определенным ресурсам, например к оперативной памяти и к некоторым нематериальным ресурсам, таким как идентификаторы пользователей (UID), то другие контейнеры хоста окажутся «на голодном пайке», результатом чего станет полный отказ в обслуживании (denial of service), при котором пользователи лишаются доступа к некоторым компонентам системы или к системе в целом;
- *взлом контейнеров* – даже если атакующий получил доступ к одному контейнеру, то должна быть исключена возможность получения им доступа к другим контейнерам или к хосту. Так как пользователи не разделены по пространствам имен, любой процесс, «взломавший» контейнер, получит на хосте те же привилегии, которые были у него в контейнере: если в контейнере владельцем процесса был суперпользователь root, то и на хосте этот процесс будет иметь права суперпользователя root[1]. Кроме того, это означает, что вы должны принять меры против атак с использованием *повышения привилегий* (*privilege escalation*), при которых пользователь захватывает более высокие права доступа, например привилегии root, чаще всего вследствие ошибок в коде приложений, требующих запуска с расширенными привилегиями. Учитывая юный возраст технологии контейнеров, следует

[1] В настоящее время продолжается работа по обеспечению автоматического преобразования суперпользователя root в контейнере в обычного непривилегированного пользователя на хосте. Это существенно снизит потенциальные возможности атакующего при взломе контейнера, но может привести к проблемам при определении владельца томов.

организовать систему безопасности на основе предположения: ситуации взлома контейнеров маловероятны, но возможны;

- *зараженные образы* – как узнать, что используемые образы безопасны, что они не поддельные, что они взяты из правильного источника? Если атакующий может обмануть вас и подменить образ, то опасности подвергаются и хост, и все ваши данные. Кроме того, необходимо всегда проверять актуальность запускаемых образов: они не должны содержать версий ПО с известными уязвимостями;
- *нарушение защиты закрытых данных* – если контейнер работает с базой данных или с сервисом, то наверняка потребуется использование некоторых закрытых данных, таких как API-ключ или имена пользователей и пароли. Когда атакующий узнает закрытые данные, он получает доступ ко всему сервису. Эта проблема крайне обостряется в архитектуре микросервисов, где контейнеры непрерывно останавливаются и запускаются, в отличие от архитектуры с малым количеством виртуальных машин с длинным жизненным циклом. Решения, обеспечивающие безопасное совместное использование закрытых данных, обсуждались выше, в разделе «Совместное использование закрытых данных» главы 9.

Контейнеры и пространства имен

В своей часто цитируемой статье Дэн Уолш (Dan Walsh) из Red Hat пишет: «Контейнеры не защищают (не изолируют)» (Containers do not contain). В первую очередь он имеет в виду то, что не все ресурсы, к которым контейнеры имеют доступ, размещены в каком-либо пространстве имен (namespace). Ресурсы, которые действительно размещены в определенном пространстве имен, отображаются в отдельное независимое значение на хосте (например, идентификатор процесса PID 1 внутри контейнера никоим образом не соответствует PID 1 на хосте или в любом другом контейнере). Но ресурсы, не принадлежащие к какому-либо пространству имен, одинаковы как на хосте, так и в контейнере.

К ресурсам, к которым не применяются разграничения при помощи пространств имен, относятся:

- идентификатор пользователя (UID) – пользователи внутри контейнера имеют тот же UID, что и на хосте. И если контейнер запущен с правами root, то при взломе этого контейнера атакующий получит права root на всем хосте. В настоящее время продолжается разработка, которая позволит преобразовать пользователя root в контейнере в обычного непривилегированного пользователя на хосте, но она еще не завершена;
- комплект ключей ядра (kernel keyring) – если ваше приложение или зависящие от него приложения используют комплект ключей ядра для обработки криптографических ключей и т. п., то очень важно знать об этом. Ключи разделены по UID, следовательно, контейнеры, запускаемые с одинаковым UID, получат доступ к одному и тому же набору ключей, как и соответствующий пользователь на хосте;
- само ядро и все модули ядра – если контейнер загружает модуль ядра (для этого требуются расширенные привилегии), то этот модуль становится доступным

всем контейнерам на данном хосте. Это относится и к модулям защиты Linux Security Modules, обсуждаемым ниже;
- устройства – дисковые накопители, звуковые карты, устройства обработки графики (GPU);
- системное время – при изменении времени внутри любого контейнера изменяется системное время соответствующего хоста и всех прочих контейнеров. Такое изменение возможно только в тех контейнерах, для которых явно определен параметр `SYS_TIME`, который по умолчанию не разрешен.

Необходимо признать, что Docker в сочетании с лежащими в его основе функциональными возможностями ядра Linux пока еще остается не вполне зрелой технологией и по большому счету пока не готов серьезно конкурировать с технологией виртуальных машин. До самого последнего времени контейнеры не способны обеспечить уровень обеспечения безопасности, гарантируемый виртуальными машинами[1].

Глубокая защита

Так что же делать? Принять как должное потенциальную уязвимость и создавать глубокую защиту (defence-in-depth). Рассмотрим аналогию с крепостью, имеющей несколько оборонительных рубежей, предназначенных для отражения разнообразных способов штурма. Обычно крепость обнесена рвом с водой или использует естественный рельеф местности для регулирования доступа внутрь. Толстые стены из прочного камня способны противостоять огню и артиллерийским снарядам. В стенах крепости устроены парапеты с бойницами для защитников на нескольких уровнях. А когда атакующие преодолеют первую линию обороны, на их пути встает следующая.

Защита любой системы также должна состоять из нескольких уровней. Например, контейнеры с большой вероятностью будут запускаться в виртуальных машинах, поэтому при взломе контейнера следующий уровень защиты может предотвратить доступ атакующего к хосту или к контейнерам, принадлежащим другим пользователям. Необходимо обеспечить контроль системы с оперативным оповещением администраторов обо всех случаях необычного поведения. Сетевые экраны должны ограничивать сетевой доступ к контейнерам, сокращая внешнюю поверхность атаки.

---

[1] Постоянно ведется интересная дискуссия по поводу способности контейнеров обеспечить такой же уровень безопасности, как у виртуальных машин. Сторонники виртуальных машин утверждают, что отсутствие гипервизора и необходимость совместного использования ресурсов ядра означают, что контейнеры навсегда останутся менее безопасными. Сторонники контейнеров возражают, что виртуальные машины более уязвимы из-за того, что имеют бóльшую поверхность атаки, отмечая большой объем сложного кода с расширенными привилегиями в виртуальных машинах, необходимого для эмуляции внутренней аппаратуры (в качестве примера можно привести недавно выявленную уязвимость VENOM (http://venom.crowdstrike.com/), использующую недостатки кода в эмуляции привода флоппи-дисков).

Принцип минимальных привилегий

Необходимо всегда придерживаться важного принципа *минимальных привилегий* (*least privilege*): каждый процесс и контейнер должен запускаться только с минимальным набором прав доступа и ресурсов, которые нужны ему для выполнения поставленной задачи[1]. Главное преимущество этого подхода заключается в том, что при взломе одного контейнера система защиты должна сохранить жесткое ограничение доступа атакующего к другим данным и/или ресурсам.

Исходя из принципа минимальных привилегий, для ограничения возможностей контейнеров можно предложить следующие меры:

- исключение возможности запуска процессов в контейнерах от имени root, чтобы какая-либо уязвимость в процессе не давала атакующему привилегии суперпользователя;
- монтирование файловых систем в режиме «только для чтения», чтобы лишить атакующего возможности перезаписать данные или сохранить вредоносные скрипты в файл;
- исключение возможности для контейнера выполнения системных вызовов (обращений непосредственно к ядру) для сокращения потенциальной поверхности атаки;
- ограничение ресурсов контейнера может использоваться для того, чтобы избежать DoS-атак, при которых взломанный контейнер захватывает чрезмерное количество ресурсов (таких как оперативная память и/или процессор(ы)), что приводит к потере работоспособности хоста.

Привилегии Docker == привилегии root

В этой главе основное внимание уделено безопасности работающих контейнеров, но, кроме того, очень важно помнить о тех пользователях, кому предоставлен доступ к демону Docker. Любой пользователь, имеющий возможность запускать контейнеры Docker, сразу же получает права root на данном хосте. Например, предположим, что вы можете выполнить следующую команду:

```
$ docker run -v /:/homeroot -it debian bash
...
```

После этого вы получаете доступ к любому файлу, в том числе и к исполняемым (бинарным) файлам на текущем хосте.

Если вы используете удаленный API для доступа к демону Docker, подумайте над тем, как обеспечить безопасный доступ и кому предоставлять права и привилегии. Если возможно, следует ограничить доступ к локальной сети.

---

[1] Изначально концепция минимальных привилегий была сформулирована в следующем виде: «Каждая программа и каждый привилегированный пользователь системы должны действовать, используя минимальный набор привилегий, необходимых для выполнения конкретного задания» Джеромом Зальтцером (Jerome Saltzer) в статье «Protection and the control of information sharing in Multics» (Communications of the ACM, vol. 17). Не так давно Диого Моника (Diogo Mónica) и Натан Макколи (Nathan McCauley) из Docker поддержали идею «микросервисов с минимальными привилегиями», основанную на принципе Зальтцера.

Обеспечение безопасности identidock

Следуя основной теме данной главы, рассмотрим возможности обеспечения безопасности нашего приложения identidock при эксплуатации. В identidock не хранится какая-либо закрытая информация, поэтому основными целями атаки являются проникновение на хост и переключение сервера на распространение спама или чего-то подобного. Предположим, что identidock имеет некоторую ценность и некоторый контингент пользователей, которые, по меньшей мере, будут недовольны, если identidock прекратит работу[1].

В описанной ситуации рекомендуется предпринять следующие меры:

- контейнеры identidock запускаются в виртуальной машине или на специально выделенном хосте, чтобы исключить явную возможность атаки другими пользователями или сервисами (см. раздел «Разделение контейнеров по хостам» ниже);
- единственным контейнером, который открывает порт во внешний мир, должен быть балансировщик нагрузки или обратный прокси. Это существенно сокращает поверхность атаки. Все сервисы контроля и ведения журналов должны быть объявлены через закрытые внутренние интерфейсы или через VPN (см. раздел «Ограничения сетевой среды контейнеров» ниже);
- все образы приложения identidock явно определяют пользователя и не запускаются от имени root (см. раздел «Настройка пользователя» ниже);
- все образы приложения identidock загружаются с обязательным контролем хэш-значения или из надежных и проверенных источников (см. раздел «Подтверждение происхождения образов» ниже);
- необходимо организовать контроль и систему оповещения для выявления нестандартного поведения и трафика (см. главу 10);
- все контейнеры используют только самые свежие версии программного обеспечения, и в режиме эксплуатации вывод отладочной информации отключен (см. раздел «Обновления» ниже);
- если есть возможность, то на хосте включаются механизмы AppArmor и SELinux (см. разделы «AppArmor» и «SELinux» ниже);
- необходимо добавить некоторую форму управления доступом или механизм защиты паролей для доступа к Redis.

Если время позволяет, то можно принять дополнительные меры:

- удалить все ненужные выполняемые (бинарные) файлы с установленным битом setuid из образов identidock. Это уменьшает вероятность подвергнуться атаке с целью повышения привилегий атакующего (см. раздел «Удаление выполняемых бинарных файлов с установленными битами setuid и setgid»);
- по возможности работать с файловыми системами в режиме, защищенном от записи (только для чтения). Контейнеры identidock, dnmonster и redis могут быть запущены с внутренними файловыми системами, защищен-

[1] Снова отмечу: будет проще, если вы примете эту условную ситуацию всерьез.

ными от записи, но на том redis должна быть разрешена запись (см. раздел «Ограничения файловых систем» ниже);

- исключить все ненужные привилегии уровня ядра. Контейнеры identidock и dnmonster будут работать без каких-либо исключительных возможностей (см. раздел «Ограничение возможностей» ниже).

При более скрупулезном отношении (которое часто называют «параноидальным») или при организации сервиса с повышенными требованиями к обеспечению безопасности можно рекомендовать следующие меры:

- ограничение памяти для каждого контейнера с помощью флага `-m`. Это должно предотвратить утечки памяти и некоторые типы DoS-атак. При этом требуется профилирование контейнеров с целью определения максимального потребления памяти или ограничения с широким допуском;
- запуск SELinux со специально определенными типами для контейнеров. Это может стать весьма эффективным средством обеспечения безопасности, но требует большого объема работ и возможно только при использовании драйвера системы хранения данных devicemapper (см. раздел «SELinux» ниже);
- применение системной утилиты `ulimit` для управления количеством процессов. Ограничение накладывается на пользователя конкретного контейнера, поэтому выполнить это не так просто, как может показаться. Такая мера исключит опасность применения так называемой fork-бомбы в качестве разновидности DoS-атаки (см. раздел «Ограничения ресурсов (ulimit)» ниже);
- шифрование внутренних коммуникаций, чтобы затруднить искажение данных атакующим.

Кроме того, должны регулярно проводиться контрольные проверки системы, позволяющие поддерживать все ее компоненты в актуальном состоянии и убедиться в отсутствии контейнеров, захватывающих бóльшую часть ресурсов. Даже для учебного приложения, подобного identidock, существует множество мер обеспечения безопасности, которые следует применить безоговорочно, и множество мер, применение которых подлежит обсуждению.

В оставшейся части главы перечисленные выше способы защиты и прочие меры безопасности будут рассматриваться более подробно. Главное – помнить, что чем больше проверок и ограничений введено в действие, тем выше шансы остановить атаку до того, как она причинит действительный вред. При неправильном использовании Docker снижает уровень защиты системы, открывая новые векторы атаки. Правильное применение укрепляет защиту, добавляя новые уровни изоляции и ограничивая те области действия приложений, в которых они могут причинить ущерб системе.

Разделение контейнеров по хостам

Если в комплексной среде контейнеры запускаются для многих пользователей (не важно, являются они внутренними пользователями одной организации или

внешними клиентами), то следует обеспечить каждого пользователя отдельным Docker-хостом, как показано на рис. 13.1. Это менее эффективно, чем совместное использование хостов несколькими пользователями, и влечет за собой увеличение количества виртуальных машин и/или аппаратных узлов, но более важно для обеспечения безопасности. Главное обоснование такого решения – предотвращение взлома контейнеров, в результате которого пользователь получает доступ к контейнерам и данным других пользователей. При взломе контейнера атакующий останется на отдельной виртуальной машине или компьютере, и ему будет нелегко получить доступ к контейнерам, принадлежащим другим пользователям.

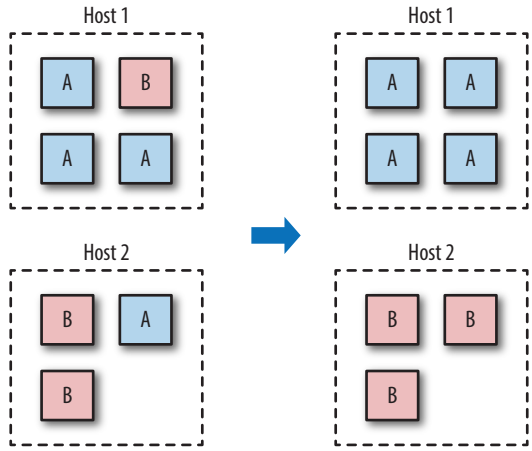

Рис. 13.1. Разделение контейнеров по хостам

Если имеются контейнеры, обрабатывающие или хранящие важную закрытую информацию, то необходимо разместить их на специальном хосте, отдельно от контейнеров, работающих с менее важной информацией, в особенности от контейнеров с приложениями, взаимодействующими непосредственно с конечными пользователями. Например, контейнеры, обрабатывающие параметры кредитных карт, должны быть отделены от контейнеров, запускающих внешний компонент Node.js.

Разделение по хостам и использование виртуальных машин также может обеспечить дополнительную защиту от DoS-атак: пользователи не смогут захватить всю память на хосте и заставить конкурентов «голодать», если они размещены в изолированной виртуальной машине.

Короче говоря, огромное количество развернутых контейнеров предполагает использование виртуальных машин. Это не самая благоприятная ситуация, но в этом случае мы можем совместить эффективность контейнеров с безопасностью виртуальных машин.

Обновления

Возможность оперативного применения обновлений компонентов работающей системы чрезвычайно важна для обеспечения безопасности, особенно когда уязвимости обнаружены в широко используемых утилитах и программных средствах.

Процесс обновления системы, состоящей из контейнеров, в общем виде состоит из следующих этапов:

1. Определение образов, требующих обновления. Рассматриваются как основные образы, так и все образы, обеспечивающие разрешение зависимостей. Способы выполнения этой операции в командной строке см. в примечании «Получение списка работающих образов» ниже.
2. Получение или создание обновленной версии каждого основного образа. Выгрузка этой версии в соответствующий реестр или на сайт загрузки.
3. Для каждого образа, обеспечивающего разрешение зависимостей, выполняется команда `docker build` с аргументом `--no-cache`. Новые образы также выгружаются в реестр.
4. На каждом Docker-хосте выполняется команда `docker pull`, обеспечивающая полную актуальность всех образов.
5. На каждом Docker-хосте выполняется перезапуск обновляемых контейнеров.
6. После проверки корректности работы системы в целом старые образы удаляются с хостов. Если есть возможность, следует удалить старые образы и из реестра.

Некоторые из этих этапов проще описать, чем выполнить на деле. Определение образов, для которых необходимо обновление, может потребовать больших объемов ручной работы и отличного знания командной оболочки. При перезапуске контейнеров предполагается наличие какого-либо способа поддержки механизма непрерывных обновлений или допустимость некоторого небольшого интервала простоя. Во время написания книги функциональные возможности по полному удалению образов из реестра и освобождению дискового пространства пока еще находились в процессе разработки[1].

Если для создания образов используется Docker Hub, то следует помнить о возможности установления таких связей в репозитории, которые позволяют запустить процесс создания нового образа при любых изменениях в связанных с ним образах. При правильной настройке связей с основным образом генерация новой версии вашего образа будет выполняться автоматически при внесении изменений в основной образ.

Получение списка работающих образов

Следующая команда выдает идентификаторы для всех активных образов:

```
$ docker inspect -f "{{.Image}}" $(docker ps -q)
42a3cf88f3f0cce2b4bfb2ed714eec5ee937525b4c7e0a0f70daff18c3f2ee92
41b730702607edf9b07c6098f0b704ff59c5d4361245e468c0d551f50eae6f84
```

[1] Эту проблему можно решить с помощью следующего «обходного маневра»: загрузить все необходимые новые образы, затем выгрузить их в новый пустой реестр.

Средства командой оболочки можно использовать для получения более подробной информации:

```
$ docker images --no-trunc | \
    grep $(docker inspect -f "-e {{.Image}}" $(docker ps -q))
nginx   latest  42a3cf88f...  2 weeks ago  132.8 MB
debian  latest  41b730702...  2 weeks ago  125.1 MB
```

Для получения списка всех образов и соответствующих базовых или промежуточных (вспомогательных) образов выполняется команда (используйте ключ `--no-trunc` для вывода полных идентификаторов):

```
$ docker inspect -f "{{.Image}}" $(docker ps -q) | \
    xargs -L 1 docker history -q
41b730702607
3cb35ae859e7
42a3cf88f3f0
e59ba510498b
50c46b6286b9
ee8776c93fde
439e7909f795
0b5e8be9b692
e7e840eed70b
7ed37354d38d
55516e2f2530
97d05af69c46
41b730702607
3cb35ae859e7
```

Эти команды можно объединить и усовершенствовать, чтобы получить подробную информацию по образам:

```
$ docker images | \
   grep $(docker inspect -f "{{.Image}}" $(docker ps -q) | \
          xargs -L 1 docker history -q | sed "s/^/\-e /")
nginx   latest  42a3cf88f3f0  2 weeks ago  132.8 MB
debian  latest  41b730702607  2 weeks ago  125.1 MB
```

Если нужны подробности, касающиеся вспомогательных образов и именованных образов, то в команду `docker images` нужно добавить аргумент `-a`. Отметим, что в этой команде есть одна тонкость: если на хосте нет версии основного образа, снабженной тегом, то эта версия не будет выведена в списке. Например, официальный образ Redis основан на `debian:wheezy`, но сам основной образ появится как `<None>` в выводе команды `docker images -a`, если только данный хост в явной форме отдельной командой не загрузит образ `debian:wheezy` (то есть в точности ту версию, которая является основой Redis).

При необходимости исправить уязвимость, найденную в образе от стороннего производителя, включая официальные образы, все зависит от своевременности обновления данного образа его производителем. В прошлом создателей крити-

ковали за медленную реакцию на уязвимости. В подобной ситуации вам приходится ждать обновлений или создавать собственный образ. Если предположить возможность доступа к файлу Dockerfile и исходному коду для нужного образа, то поддержка генерации образа при необходимости может стать простым и эффективным решением.

Такой подход следует рассматривать как противоположный обычной методике применения виртуальных машин с использованием ПО управления конфигурацией (CM), такого как Puppet, Chef или Ansible. При CM-подходе виртуальные машины не создаются повторно, но обновляются и корректируются при необходимости либо с помощью SSH-команд, либо через агента, установленного на данной виртуальной машине. Такая методика вполне работоспособна, но нужно учитывать, что отдельные виртуальные машины часто находятся в различных состояниях, что создает значительные сложности для контроля и для процесса обновления виртуальных машин. Этот подход необходим, чтобы избежать издержек, связанных с повторным созданием виртуальных машин и сопровождением главного или «золотого» образа для сервиса. CM-методику можно применять и для контейнеров, но при этом возрастает сложность без какой-либо выгоды – методика упрощенного «золотого» образа хорошо работает для контейнеров благодаря быстрому запуску контейнеров и простоте создания и сопровождения образов[1].

Присваивайте метки своим образам

Идентификацию образов и их содержимого можно существенно упростить, используя содержательные метки при создании образов. Такая возможность появилась в версии 1.6. Она позволяет автору связать произвольную пару ключ-значение с создаваемым образом. Это можно сделать в Dockerfile:

```
FROM debian
LABEL version 1.0
LABEL description "A test image for describing labels"
```

Эту возможность можно использовать более широко, например добавлять данные, такие как хэш-значение git, получаемое на основе исходного кода, из которого был скомпилирован образ, но при этом потребуются некоторые дополнительные инструменты формирования шаблонов для автоматического обновления хэш-значения.

Метки также можно добавлять в контейнер во время запуска:

```
$ docker run -d --name label-test -l group=a debian sleep 100
1d8d8b622ec86068dfa5cf251cbaca7540b7eaa67664a13c620006...
$ docker inspect -f '{{json .Config.Labels}}' label-test
{"group":"a"}
```

---

[1] Это очень похоже на современные идеи *неизменяемой инфраструктуры* (*immutable infrastructure*), где инфраструктура – в том числе аппаратные средства, виртуальные машины и контейнеры – никогда не изменяется частично, а полностью заменяется, если требуется внесение каких-либо изменений.

Это может оказаться полезным, если необходимо обрабатывать определенные события во время выполнения, например при динамическом распределении контейнеров по группам балансировки нагрузки.

Иногда возникает необходимость в обновлении демона Docker, чтобы получить доступ к новым функциональным возможностям, исправлениям ошибок и усовершенствованиям системы безопасности. Приходится перемещать все контейнеры на новый хост или временно останавливать их работу, чтобы выполнить процедуру обновления. Настоятельно рекомендую подписаться на группы docker-user или docker-dev в Google для получения сообщений о важных обновлениях.

Не используйте неподдерживаемых драйверов

Несмотря на малый срок существования, программная среда Docker уже прошла несколько стадий разработки, и некоторые ее функции стали выводиться из употребления, их поддержка прекращена. Использование таких функций представляет собой потенциальную опасность, так как устаревшие функции не обновляются, и вообще им не уделяется столько внимания, сколько прочим компонентам Docker. То же относится к драйверам и модулям расширения, связанным с Docker.

В особенности не следует использовать устаревший драйвер поддержки выполнения LXC. По умолчанию он отключен, но с помощью аргумента `-e lxc` можно убедиться в том, что демон действительно не запускает его.

Драйверы файловых систем также активно разрабатываются и постоянно изменяются. Во время написания книги в Docker осуществлялся переход от AUFS к Overlay как к основному драйверу файловой системы. Для драйвера AUFS прекращается поддержка в ядре, и его разработка прекращена. Пользователям AUFS предлагается как можно быстрее перейти к использованию драйвера Overlay.

Подтверждение происхождения образов

Для безопасного использования образов необходимо получить подтверждение их *происхождения* (*provenance*): из какого источника они взяты и кто их автор. Следует убедиться в том, что получен именно тот образ, который тщательно протестирован его разработчиком, а не какой-либо другой образ, измененный во время хранения или передачи. Если проверка невозможна, то образ может оказаться поврежденным или – хуже того – намеренно подмененным со злым умыслом. С учетом описанных выше проблем безопасности Docker это важнейший вопрос, и вы должны учитывать вероятность того, что образ, подмененный злоумышленником, может получить полный доступ к хосту.

Подтверждение происхождения – достаточная старая задача в области ИТ. Одним из первых инструментов подтверждения происхождения ПО или данных являлась *криптографическая хэш-функция* (*secure hash*), представляющая собой что-то, подобное отпечаткам пальцев для данных, – генерируется (относительно) небольшая строка, однозначно соответствующая заданным исходным данным. Любые изменения в данных приводят к изменению хэш-значения. Для вычисления

такого хэш-значения разработано несколько алгоритмов с различными степенями сложности и обеспечения уникальности генерируемого значения. Чаще всего используются алгоритмы SHA (и его варианты) и MD5 (его применение не рекомендуется из-за обнаруженных существенных проблем). Если имеются некоторые данные и предоставлено хэш-значение для них, то можно повторно вычислить хэш и сравнить с исходным. Если хэш-значения совпадают, то данные не повреждены и не искажены. Но возникает вопрос: почему и в какой степени можно доверять хэш-значению? Что мешает взломщику изменить и данные, и хэш-значение? Наилучшим ответом на эти вопросы является использование *криптографической подписи* (*cryptographic signing*) и пары открытый ключ – закрытый ключ.

С помощью криптографической подписи можно проверить подлинность стороны, предоставляющей (передающей) любой объект. Если передающая сторона подписывает свой объект собственным *закрытым ключом* (*private key*)[1], то любая сторона, принимающая этот объект, может проверить подлинность подписи отправителя с помощью соответствующего *открытого ключа* (*public key*). Предполагая, что клиент предварительно получил копию открытого ключа отправителя и что этот ключ правильный, мы можем быть уверены в том, что объект передан отправителем и его содержимое не изменено каким бы то ни было образом.

Дайджесты Docker

В программной среде Docker криптографические хэш-значения называются дайджестами. *Дайджест* (*digest*) – это вычисленное по алгоритму SHA256 хэш-значения уровня файловой системы или манифеста, где манифест представляет собой файл метаданных, описывающих составные части образа Docker. Поскольку манифест содержит список всех уровней образа, идентифицируемых по дайджесту[2], вы можете проверить правильность содержимого данного манифеста и после успешной проверки без сомнений загружать все уровни образа даже по недоверенным каналам связи (например, HTTP).

Механизм подтверждения контента в Docker

Подтверждение контента в Docker введено, начиная с версии 1.8. Это механизм Docker, позволяющий авторам образов[3] подписывать контент (содержимое), дополняя механизм надежной доставки. При извлечении образа из репозитория пользователь получает сертификат, содержащий открытый ключ автора, позволяющий проверить, действительно ли данный образ получен от объявленного автора.

---

[1] Полное описание криптографии с использованием открытых ключей весьма занимательно, но к теме данной главы и книги не относится. Более подробно об этом можно прочесть в книге Bruce Schneier «Applied Cryptography» (имеется в переводе на русский язык: Брюс Шнайер. Прикладная криптография. М.: Триумф, 2002).

[2] Похожая конструкция используется в протоколах Bittorrent и Bitcoin; она называется список хэш-значений (hash list).

[3] В контексте этой главы автором (publisher) является любой, кто выгружает (публикует) образ. Авторами могут быть не только крупные компании или организации.

Когда механизм подтверждения контента разрешен, механизм Docker будет работать только с подписанными образами, а для образов с несовпадающими подписями или дайджестами запуск будет запрещен.

Рассмотрим, как работает механизм подтверждения контента. Начнем с попытки загрузки подписанного и неподписанного образов:

```
$ export DOCKER_CONTENT_TRUST=1  ❶
$ docker pull debian:wheezy
Pull (1 of 1): debian:wheezy@sha256:c584131da2ac1948aa3e66468a4424b6aea2f33a...
sha256:c584131da2ac1948aa3e66468a4424b6aea2f33acba7cec0b631bdb56254c4fe: Pul...
4c8cbfd2973e: Pull complete
60c52dbe9d91: Pull complete
Digest: sha256:c584131da2ac1948aa3e66468a4424b6aea2f33acba7cec0b631bdb56254c4fe
Status: Downloaded newer image for debian@sha256:c584131da2ac1948aa3e66468a4...
Tagging debian@sha256:c584131da2ac1948aa3e66468a4424b6aea2f33acba7cec0b631bd...
$ docker pull amouat/identidock:unsigned
No trust data for unsigned
(Нет подтвержденных данных для неподписанного образа)
```

❶ В Docker 1.8 механизм подтверждения контента обязательно должен включаться установкой переменной среды `DOCKER_CONTENT_TRUST=1`. В последующих версиях Docker этот механизм будет активен по умолчанию.

Здесь мы видим, что официальный подписанный образ Debian успешно загрузился. Но попытку загрузки неподписанного образа `amouat/identidock:unsigned` Docker запретил.

А как выгрузить подписанный образ? Эта процедура на удивление проста:

```
$ docker push amouat/identidock:newest
The push refers to a repository [docker.io/amouat/identidock] (len: 1)
...
843e2bded498: Image already exists
newest: digest: sha256:1a0c4d72c5d52094fd246ec03d6b6ac43836440796da1043b6ed8...
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase will be used to
protect the most sensitive key in your signing system. Please choose a long, complex pass-
phrase and be careful to keep the password and the key file itself secure and backed up. It
is highly recommended that you use a password manager to generate the passphrase and keep it
safe. There will be no way to recover this key. You can find the key in your config directory.
Enter passphrase for new offline key with id 70878f1:
Repeat passphrase for new offline key with id 70878f1:
Enter passphrase for new tagging key with id docker.io/amouat/identidock ...
Repeat passphrase for new tagging key with id docker.io/amouat/identidock ...
Finished initializing "docker.io/amouat/identidock"
(Подпись и выгрузка подтвержденных метаданных. Вы создаете новый пароль для корневого ключа подписи.
Этот пароль будет использоваться для защиты наиболее важного ключа в подписанной вами системе. Выберите
достаточно длинный сложный пароль и аккуратно храните его и сам файл ключа в безопасном месте с созданием
резервной копии. Настоятельно рекомендуется использовать менеджер паролей для генерации и хранения пароля.
При потере ключа восстановить его невозможно. Ключ вы найдете в каталоге конфигурации.
```

```
Введите пароль для нового офлайнового ключа с id 70878f1:
Повторите ввод пароля для нового офлайнового ключа с id 70878f1:
Введите пароль для нового ключа тегирования с id docker.io/amouat/identidock ...
Повторите ввод пароля для нового ключа тегирования с id docker.io/amouat/identidock ...
Завершена инициализация «docker.io/amouat/identidock»)
```

Поскольку я в первый раз выгружаю образ в этот репозиторий с включенным механизмом подтверждения контента, Docker создал новый *корневой ключ подписи* (*root signing key*) и новый *ключ тегирования* (*tagging key*). Ниже мы рассмотрим ключ тегирования более подробно, но сейчас следует подчеркнуть важность безопасного и надежного хранения корневого ключа. Его потеря крайне затруднит жизнь: все пользователи вашего репозитория лишатся возможности загружать новые образы и обновлять существующие образы без удаления старых сертификатов вручную.

Теперь мы можем загрузить этот образ с использованием механизма подтверждения контента:

```
$ docker rmi amouat/identidock:newest
Untagged: amouat/identidock:newest
$ docker pull amouat/identidock:newest
Pull (1 of 1): amouat/identidock:newest@sha256:1a0c4d72c5d52094fd246ec03d6b6...
sha256:1a0c4d72c5d52094fd246ec03d6b6ac43836440796da1043b6ed81ea4167eb71: Pul...
...
7e7d073d42e9: Already exists
Digest: sha256:1a0c4d72c5d52094fd246ec03d6b6ac43836440796da1043b6ed81ea4167eb71
Status: Downloaded newer image for amouat/identidock@sha256:1a0c4d72c5d52094...
Tagging amouat/identidock@sha256:1a0c4d72c5d52094fd246ec03d6b6ac43836440796d...
```

Если вы ранее не загружали этот образ из данного репозитория, то Docker сначала извлечет сертификат автора (публикатора) этого репозитория. Эта операция выполняется с использованием протокола HTTPS, что снижает риск, но может быть подобным соединению с хостом через SSH в первый раз: придется поверить, что предоставляется подлинное удостоверение личности. Дальнейшие загрузки из этого репозитория могут проверяться с помощью существующего сертификата.

Создавайте резервные копии своих ключей подписи

Docker шифрует все ключи при первой же возможности и гарантирует, что секретные данные никогда не будут записаны на жесткий диск. Учитывая чрезвычайную важность этих ключей, рекомендуется создавать их резервные копии на двух зашифрованных USB-флэш-накопителях, хранящихся в безопасном месте. Чтобы создать tar-файл, содержащий ключи, выполните следующие команды:

```
$ umask 077
$ tar -zcvf private_keys_backup.tar.gz ~/.docker/trust/private
$ umask 022
```

Первая команда `umask` позволяет установить права доступа к файлу в режиме «только для чтения».

Следует отметить, что поскольку корневой ключ необходим лишь для создания или удаления других ключей, он может (и даже должен) находиться в офлайновом хранилище, когда не требуется его использование.

Вернемся к ключу тегирования. Ключ тегирования генерируется для каждого репозитория, принадлежащего автору-владельцу. Ключ тегирования подписывается корневым ключом, что позволяет проверить его любому пользователю с помощью сертификата автора-владельца. Ключ тегирования может совместно использоваться всеми пользователями внутри любой организации для подписи всех образов для данного репозитория. После генерации ключа тегирования корневой ключ следует спрятать, записать на независимый автономный носитель и хранить в безопасном месте.

При каких-либо повреждениях ключа тегирования возможно его восстановление. Смена ключей тегирования позволяет удалить испорченный ключ из системы. Этот процесс происходит незаметно для пользователя и может использоваться как упреждающий способ защиты от невыявленных компроментаций ключей.

Механизм подтверждения контента также предоставляет средства защиты от *атак повторного воспроизведения* (*replay attacks*). При атаке повторного воспроизведения передаваемый объект заменяется объектом, который ранее считался корректным. Например, атакующий может заменить бинарный файл более старой его версией с известной уязвимостью, и эта старая версия ранее была подписана автором-владельцем. Поскольку файл заверен правильной подписью, пользователь, введенный в заблуждение, может запустить уязвимую версию выполняемого файла. Чтобы избежать этого, механизм подтверждения контента использует *ключи меток времени* (*timestamp keys*), связанных с каждым репозиторием. Для метаданных установлен короткий срок действия, по истечении которого требуется новая процедура заверения ключом метки времени, и эта процедура выполняется достаточно часто. С помощью проверки срока действия метаданных перед загрузкой образа клиент Docker может быть уверен в том, что получает актуальный (или самый новый) образ. Ключи меток времени управляются реестром Docker Hub и не требуют каких-либо действий со стороны автора – владельца репозитория.

Репозиторий может содержать как подписанные, так и неподписанные образы. Если механизм подтверждения контента активен и необходимо загрузить неподписанный образ, то используется флаг `--disable-content-trust`:

```
$ docker pull amouat/identidock:unsigned
No trust data for unsigned
$ docker pull --disable-content-trust amouat/identidock:unsigned
unsigned: Pulling from amouat/identidock
...
7e7d073d42e9: Already exists
Digest: sha256:ea9143ea9952ca27bfd618ce718501d97180dbf1b5857ff33467dfdae08f57be
Status: Downloaded newer image for amouat/identidock:unsigned
```

Более подробную информацию о механизме подтверждения контента можно получить из официальной документации Docker (https://docs.docker.com/security/trust/content_trust/), а также на сайте The Update Framework (http://theupdateframework.com/), где размещена техническая спецификация, используемая механизмом подтверждения контента.

Формируя достаточно сложную инфраструктуру с многочисленными наборами ключей, Docker все же стремится сохранить ее простоту для конечных пользователей. С помощью механизма подтверждения контента компания Docker разработала удобную для пользователей, современную защищенную программную среду, предоставляющую средства подтверждения происхождения образов, обеспечения их актуальности и целостности.

В настоящее время механизм подтверждения контента работает в реестре Docker Hub. Чтобы настроить механизм подтверждения контента для локального реестра, необходимо сконфигурировать и развернуть сервер Notary (https://github.com/docker/notary).

Notary

Проект Docker Notary (https://github.com/docker/notary) представляет собой типовой каркас программной среды сервер-клиент для публикации и обеспечения доступа к контенту на основе защищенных надежных связей. Проект Notary основан на технической спецификации The Update Framework, определяющей проектное решение механизма защиты для распространения и обновления контента.

По существу, механизм подтверждения контента Docker является сочетанием программной среды Notary и программного интерфейса Docker API. При совместной работе реестра и сервера Notary организации могут предоставлять пользователям надежные образы. Тем не менее Notary позиционируется как независимая программная среда, которая может использоваться для решения широкого спектра задач.

Основной вариант использования Notary – укрепление защиты и повышение надежности для широко используемого подхода с применением `curl | sh`, обычно указываемого в текущих инструкциях по установке Docker:

```
$ curl -sSL https://get.docker.com/ | sh
```

При вмешательстве в процесс загрузки на сервере или во время передачи атакующий получит возможность выполнять любые команды на взломанном компьютере. Использование протокола HTTPS защитит данные от искажения при передаче, но остается опасность досрочного прерывания процесса загрузки, в результате чего не полностью загруженный код становится источником потенциальной опасности. Аналогичный пример загрузки с использованием Notary выглядит приблизительно так:

```
$ curl http://get.docker.com/ | notary verify docker.com/scripts v1 | sh
```

При вызове команды `notary` контрольная сумма каждого скрипта установки сравнивается с контрольной суммой в защищенном списке скриптов для docker.com. Если контрольные суммы совпадают, то проверка считается успешной и гарантирует, что скрипт действительно получен из docker.com без искажений и повреждений. Если контрольные суммы не совпадают, то Notary останавливает процедуру, и данные в `sh` не передаются. Здесь заслуживает внимания тот факт, что сам скрипт можно передавать по незащищенным каналам (в данном случае по протоколу HTTP), не беспокоясь о его корректности, – если во время передачи скрипт будет каким-либо образом изменен, то изменится и его контрольная сумма, и Notary установит состояние ошибки.

При использовании неподписанных образов сохраняется возможность их проверки посредством загрузки образа по дайджесту, а не по имени и тегу, например:

```
$ docker pull debian@sha256:f43366bc755696485050ce14e1429c481b6f0ca04505c4a3093d\
fdb4fafb899e
```

Здесь будет загружен образ `debian:jessie` как образ, актуальный во время написания данной книги. В отличие от использования тега `debian:jessie`, эта команда всегда гарантирует загрузку указанного образа (или вообще никакого). Если есть возможность каким-либо способом передать и аутентифицировать дайджест (например, отправка по электронной почте с PGP-подписью из надежного источника), то вы можете гарантировать подлинность данного образа. Даже при работающем механизме подтверждения контента сохраняется возможность загрузки образа по его дайджесту.

Если вы не доверяете частному реестру или реестру Docker Hub распространение своих образов, то всегда можете воспользоваться командами `docker load` и `docker save` для экспорта и импорта образов. Образы могут распространяться через внутренний сайт загрузок или просто в виде копий файлов. Но если вы пойдете по этому пути, то, вероятнее всего, вскоре убедитесь в том, что заново изобретаете функциональные возможности, которые уже существуют в реестре Docker и в компонентах механизма подтверждения контента.

Повторно воспроизводимые и надежные файлы Dockerfile

В идеальном случае файлы Dockerfiles должны генерировать каждый раз абсолютно одинаковые образы. На практике это не всегда достижимо. С большой вероятностью один и тот же Dockerfile в разное время будет генерировать различные образы. Это явная проблема, и в очередной раз отмечу: она связана с трудностью определения, что именно находится внутри ваших образов. Но, по крайней мере, есть возможность поддерживать воспроизведение максимально похожих друг на друга образов, если при написании файлов Dockerfile выполнять следующие правила:

- всегда определяйте тег в инструкциях `FROM`. `FROM redis` – неудачный вариант, так как по умолчанию присваивается тег `latest`, которые требует изменения через некоторое время из-за значительных корректировок версии. `FROM redis:3.0` лучше, но все-таки требует изменения после небольших обновлений и исправления ошибок (может потребоваться именно такая слегка подправленная версия). Если нужна полная уверенность в том, что каждый раз загружается в точности требуемый образ, то единственным вариантом является использование дайджеста, описанное выше. Например:

  ```
  FROM redis@sha256:3479bbcab384fa343b52743b933661335448f8166203688006...
  ```

  Кроме того, использование дайджеста защищает образ от случайного повреждения или искажения;
- указывайте номера версий при установке ПО с помощью менеджеров пакетов. Приемлемый вариант `apt-get install cowsay`, поскольку программа `cowsay` вряд ли изменится, но гораздо лучшим вариантом будет `apt-get in-`

stall cowsay=3.03+dfsg1-6. То же самое относится и к другим системам установки пакетов, таким как pip, – если есть возможность, всегда указывайте номер версии. При удалении старой версии пакета собранный образ станет неработоспособным, но, по крайней мере, вы получите предупреждающее сообщение об этом. Но проблема остается: многие пакеты загружаются по зависимостям, а эти зависимости часто определяются отношением >=, следовательно, изменяются со временем. Для окончательного решения проблемы с версиями пакетов следует обратить внимание на инструментальные средства типа aptly (http://www.aptly.info/), которые позволят создавать моментальные снимки (оперативные резервные копии) текущего состояния репозиториев;

- тщательно проверяйте все ПО и все данные, загружаемые из Интернета. Здесь подразумевается использование контрольных сумм или криптографических подписей. Это правило является самым важным из всех перечисленных в данном списке. Если вы не проверяете загружаемые объекты, то ставите себя в зависимость от случайных повреждений и от преднамеренных искажений, вносимых атакующими в процессе загрузки. Это особенно важно при передаче ПО по протоколу HTTP, совершенно не защищенному от атаки типа «человек посередине» (man-in-the-middle attacks). В следующем разделе приводятся рекомендации, позволяющие устранить проблемы такого рода.

Большинство файлов Dockerfile для официальных образов предоставляет удачные примеры использования версий с указанием тегов и проверок загружаемых объектов. Кроме того, обычно они используют специальный тег для основного образа, но не применяют номера версий при установке ПО с помощью менеджеров пакетов.

Обеспечение безопасной загрузки ПО в файлах Dockerfile

В большинстве случаев производители предоставляют контрольные суммы для проверки загружаемых образов. Например, в Dockerfile для официального образа Node.js включена следующая информация:

```
RUN gpg --keyserver pool.sks-keyservers.net \
       --recv-keys 7937DFD2AB06298B2293C3187D33FF9D0246406D \
                   114F43EE0176B71C7BC219DD50A3051F888C628D  ❶
ENV NODE_VERSION 0.10.38
ENV NPM_VERSION 2.10.0
RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/\
node-v$NODE_VERSION-linux-x64.tar.gz" \  ❷
  && curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \  ❸
  && gpg --verify SHASUMS256.txt.asc \  ❹
  && grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" SHASUMS256.txt.asc \
     | sha256sum -c -  ❺
```

❶ Получение GPG-ключей, используемых для подписи загружаемого образа Node.js. Здесь они нужны для того, чтобы удостовериться в их корректности.
❷ Загрузка сжатого tar-архива Node.js.
❸ Получение контрольной суммы для этого архивного файла.
❹ Использование утилиты GPG для проверки подписи контрольной суммы – она должна быть подписана владельцем предоставленных выше ключей.
❺ Проверка контрольной суммы загруженного tar-архива на совпадение с контрольной суммой, вычисленной с помощью утилиты sha256sum.

Если проверка GPG-ключей или проверка контрольной суммы не проходит, то процедура создания образа отменяется.

В некоторых случаях пакеты доступны в репозиториях третьих сторон, тогда безопасность их установки обеспечивается добавлением в Dockerfile заданного репозитория и соответствующих ключей для подписи. Например, в файле Dockerfile для официального образа Nginx содержится следующая информация:

```
RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 \
                --recv-keys 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62
RUN echo "deb http://nginx.org/packages/mainline/debian/ jessie nginx" \
        >> /etc/apt/sources.list
```

Первая команда позволяет получить ключ подписи для Nginx (этот ключ добавляется в хранилище ключей), вторая команда добавляет репозиторий с пакетом Nginx в список репозиториев для управления установкой ПО. После этого Nginx в безопасном режиме просто устанавливается командой `apt-get install -y nginx` (рекомендуется указывать номер версии).

Если для нужного пакета отсутствует подпись и/или контрольная сумма, то можно без особого труда сгенерировать собственные характеристики. Например, генерация контрольной суммы для конкретной версии Redis выполняется командой:

```
$ curl -s -o redis.tar.gz http://download.redis.io/releases/redis-3.0.1.tar.gz
$ sha1sum -b redis.tar.gz  ❶
fe1d06599042bfe6a0e738542f302ce9533dde88 *redis.tar.gz
```

❶ Создание 160-битовой контрольной суммы SHA1. Флаг `-b` сообщает утилите `sha1sum`, что обрабатываются двоичные данные, а не текст.

После тестирования и проверки этого программного пакета можно добавить в Dockerfile строку, подобную следующей:

```
RUN curl -sSL -o redis.tar.gz http://download.redis.io/releases/redis-3.0.1.tar.gz \
    && echo "fe1d06599042bfe6a0e738542f302ce9533dde88 *redis.tar.gz" \
       | sha1sum -c -
```

Эта команда загружает пакет и сохраняет его как *redis.tar.gz*, затем запускает утилиту `sha1sum` для проверки контрольной суммы. Если проверка не проходит, то команда завершается с кодом ошибки, и процедура генерации образа отменяется.

Если вы часто выпускаете версии своих образов, то изменение всех описанных выше деталей потребует большого объема ручной работы, поэтому следует рассмотреть возможности автоматизации этого процесса. Во многих репозиториях официальных образов для этой цели предназначены скрипты `update.sh`.

Рекомендации по обеспечению безопасности

В этом разделе собраны советы и рекомендации по обеспечению безопасности развертываемых систем контейнеров. Не все советы универсальны и применимы в каждом случае, тем не менее следует ознакомиться с основными средствами, которыми вы можете воспользоваться.

Многие рекомендации описывают разнообразные способы ограничения возможностей контейнеров, с тем чтобы они не оказывали какого-либо воздействия на другие контейнеры или на хост. Главные ресурсы, которые надо защищать, – процессор(ы), оперативная память, идентификаторы пользователей и т. п. – совместно используются многими контейнерами. Если контейнер монополизирует любой из этих ресурсов, все прочие контейнеры остаются на голодном пайке. Гораздо хуже, если контейнер может использовать уязвимость в коде ядра, так как это может привести к выводу из строя всего хоста или к получению неограниченного доступа к хосту и другим контейнерам. Это может быть случайностью, следствием некачественного программирования или злоумышленным намерением, когда атакующий ищет способы взлома или вывода из строя хоста.

Всегда определяйте пользователя

Никогда не запускайте приложения в режиме реальной эксплуатации с правами root внутри контейнера. Это правило нужно повторять снова и снова, чтобы оно врезалось в память. Атакующий, который взламывает такое приложение, получает полный доступ к контейнеру, включая все его данные и программы. И что гораздо хуже, атакующий получит права доступа root на хосте. Вы же не запускаете приложения из-под аккаунта root в виртуальной машине или на чистой аппаратуре, так не делайте этого и в контейнере.

Чтобы исключить запуск приложений с правами root, в файлах Dockerfile всегда должен создаваться непривилегированный пользователь, после чего необходимо переключиться на него с помощью инструкции `USER` или в скрипте для точки входа. Например:

```
RUN groupadd -r user_grp && useradd -r -g user_grp user
USER user
```

Здесь создаются группа `user_grp` и новый пользователь `user`, принадлежащий к этой группе. Инструкция `USER` будет действовать во всех последующих инструкциях, а также при запуске контейнера из данного образа. Иногда необходимо ввести инструкцию `USER` несколько позже, после того как в Dockerfile сначала будут выполнены действия, требующие привилегий root, например установка ПО.

Многие официальные образы создают непривилегированного пользователя описанным выше способом, но не содержат инструкции USER. Вместо этого они переключаются на нового пользователя в скрипте для точки входа, используя для этого утилиту *gosu*. Например, скрипт для точки входа в официальном образе Redis выглядит приблизительно так:

```
#!/bin/bash
set -e
if [ "$1" = 'redis-server' ]; then
        chown -R redis .
        exec gosu redis "$@"
fi

exec "$@"
```

В этом скрипте строка chown -R redis . определяет владельца всех файлов в каталогах и подкаталогах данных образа как пользователя redis. Если в Dockerfile включить инструкцию USER, то эта строка не будет работать. В следующей строке exec gosu redis "$@" выполняет заданную команду redis с правами пользователя redis. Использование команды exec позволяет заменить текущую командную оболочку на программу redis, соответствующий процесс которой получает идентификатор PID 1 и принимает все сигналы, перенаправляемые в корневой процесс.

Используйте gosu, а не sudo

Обычно для выполнения команд с правами другого пользователя используют sudo. Хотя sudo является мощным и проверенным инструментом, некоторые его побочные эффекты не позволяют sudo стать наиболее подходящей утилитой для скриптов точки входа. Например, можно наблюдать, что происходит при выполнении команды sudo ps aux внутри контейнера Ubuntu[1]:

```
$ docker run --rm ubuntu:trusty sudo ps aux
USER       PID %CPU ... COMMAND
root         1  0.0     sudo ps aux
root         5  0.0     ps aux
```

Здесь мы видим два процесса: один для sudo, второй для выполняемой команды. Если установить утилиту gosu в образе Ubuntu и воспользоваться ею, то наблюдается следующая картина:

```
$ docker run --rm amouat/ubuntu-with-gosu gosu root ps aux
USER       PID %CPU ... COMMAND
root         1  0.0     ps aux
```

Здесь активен только один процесс – gosu выполнил команду и полностью самоустранился. Важно, что данная команда выполняется с идентификатором процесса PID 1, поэтому будет корректно принимать и обрабатывать все сигналы, посылаемые в контейнер, в отличие от примера с использованием sudo.

---

[1] Здесь я использую Ubuntu вместо Debian, потому что в образ Ubuntu утилита sudo включена по умолчанию.

Если имеется приложение, требующее привилегий суперпользователя root (и этого невозможно избежать), следует рассмотреть варианты использования таких инструментальных средств, как sudo, SELinux (см. раздел «SELinux» ниже) и fakeroot, для ограничения возможностей подобного процесса.

Ограничения сетевой среды контейнеров

Контейнер должен открывать только те порты, которые действительно необходимы в режиме реальной эксплуатации, и доступ к этим портам следует предоставлять только тем контейнерам, которые будут действительно обращаться к ним. Сделать это немного труднее, чем сказать, поскольку по умолчанию контейнеры могут обмениваться данными друг с другом вне зависимости от того, объявлены ли явно открытыми какие-либо порты или нет. Это можно наблюдать с помощью специализированного сетевого инструмента Netcat[1]:

```
$ docker run --name nc-test -d amouat/network-utils nc -l 5001  ❶
f57269e2805cf3305e41303eafefaba9bf8d996d87353b10d0ca577acc731186
$ docker run -e IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} nc-test) \
  amouat/network-utils sh -c 'echo -n "hello" | nc -v $IP 5001'  ❷
Connection to 172.17.0.3 5001 port [tcp/*] succeeded!
$ docker logs nc-test
hello
```

❶ Утилита Netcat должна прослушивать порт 5001 и выводить в консоли любые введенные данные.
❷ Передача строки "hello" в первый контейнер, использующий Netcat.

Второй контейнер может установить соединение с контейнером nc-test, несмотря на то что последний явно не объявлял каких-либо портов открытыми. Эту ситуацию можно изменить, запуская демон Docker с флагом --icc=false, который запрещает обмен данными между контейнерами. Это может защитить контейнеры от вмешательств извне, возможных при неограниченных соединениях между контейнерами. Для всех контейнеров, явно объявляющих о своих связях, сохраняется возможность обмена данными.

Docker управляет обменом информацией между контейнерами с помощью формируемых правил IPtables (для этого при запуске демона необходимо установить флаг --iptables, который должен устанавливаться по умолчанию).

Следующий пример демонстрирует поведение демона после установки флага --icc=false:

```
$ cat /etc/default/docker | grep DOCKER_OPTS=
DOCKER_OPTS="--iptables=true --icc=false"  ❶
$ docker run --name nc-test -d --expose 5001 amouat/network-utils nc -l 5001
d7c267672c158e77563da31c1ee5948f138985b1f451cd2222cf248006491139
$ docker run -e IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} nc-test)
```

[1] Здесь используется версия из OpenBSD.

```
   amouat/network-utils sh -c 'echo -n "hello" | nc -w 2 -v $IP 5001'  ❷
nc: connect to 172.17.0.10 port 5001 (tcp) timed out: Operation now in progress
$ docker run --link nc-test:nc-test \
   amouat/network-utils sh -c 'echo -n "hello" | nc -w 2 -v nc-test 5001'
Connection to nc-test 5001 port [tcp/*] succeeded!
$ docker logs nc-test
hello
```

❶ В Ubuntu демон Docker конфигурируется с помощью установки параметра среды `DOCKER_OPTS` в файле */etc/default/docker*.
❷ Флаг `-w 2` устанавливает для утилиты Netcat тайм-аут, равный двум секундам.

Первая попытка установления соединения неудачна, так как обмен данными между контейнерами запрещен, и какие-либо связи явно не объявлены. Вторая команда выполняется успешно, так как добавлено явное объявление устанавливаемого соединения. Если вы хотите лучше понять, как это работает, выполните команду `sudo iptables -L -n` на хосте со связанными контейнерами и контейнерами без связей.

При открытии портов на хосте Docker по умолчанию объявляет открытыми все интерфейсы (0.0.0.0). Вместо этого можно явно задать интерфейс, с которым необходимо установить соединение:

```
$ docker run -p 87.245.78.43:8080:8080 -d myimage
```

Это сокращает поверхность атаки, разрешая прохождение трафика только через заданный интерфейс.

Удаляйте бинарные файлы с установленными битами setuid/setgid

Вероятность того, что вашим приложениям не нужны какие-либо бинарные файлы с установленными битами setuid и/или setgid, велика[1]. При возможности удаления или запрещения выполнения таких файлов резко снижается вероятность их использования в атаках, направленных на повышение привилегий.

Чтобы получить список бинарных (выполняемых) файлов с установленными битами setuid/setgid, выполните команду `find / -perm +6000 -type f -exec ls -ld {} \;`. Например:

```
$ docker run debian find / -perm +6000 -type f -exec ls -ld {} \; 2> /dev/null
-rwsr-xr-x 1 root root 10248 Apr 15 00:02 /usr/lib/pt_chown
-rwxr-sr-x 1 root shadow 62272 Nov 20  2014 /usr/bin/chage
-rwsr-xr-x 1 root root 75376 Nov 20  2014 /usr/bin/gpasswd
-rwsr-xr-x 1 root root 53616 Nov 20  2014 /usr/bin/chfn
```

[1] Бинарные файлы с установленными битами setuid и setgid запускаются с привилегиями владельца, а не текущего пользователя. Обычно это используют для временного повышения привилегий пользователя, требуемого для выполнения конкретной задачи, например для установки или смены пароля.

```
-rwsr-xr-x 1 root root 54192 Nov 20  2014 /usr/bin/passwd
-rwsr-xr-x 1 root root 44464 Nov 20  2014 /usr/bin/chsh
-rwsr-xr-x 1 root root 39912 Nov 20  2014 /usr/bin/newgrp
-rwxr-sr-x 1 root tty 27232 Mar 29 22:34 /usr/bin/wall
-rwxr-sr-x 1 root shadow 22744 Nov 20  2014 /usr/bin/expiry
-rwsr-sr-x 1 root shadow 35408 Aug  9  2014 /sbin/unix_chkpwd
-rwsr-xr-x 1 root root 40000 Mar 29 22:34 /bin/mount
-rwsr-xr-x 1 root root 40168 Nov 20  2014 /bin/su
-rwsr-xr-x 1 root root 70576 Oct 28  2014 /bin/ping
-rwsr-xr-x 1 root root 27416 Mar 29 22:34 /bin/umount
-rwsr-xr-x 1 root root 61392 Oct 28  2014 /bin/ping6
```

После этого можно «обезвредить» эти бинарные файлы командой chmod a-s, отменяющей установку бита suid. Например, можно создать менее опасный образ Debian с помощью следующего файла Dockerfile:

```
FROM debian:wheezy
RUN find / -perm +6000 -type f -exec chmod a-s {} \; || true    ❶
```

❶ Часть команды `|| true` позволяет игнорировать любые ошибки при выполнении команды find.

После создания образа можно попробовать запустить его:

```
$ docker build -t defanged-debian .
...
Successfully built 526744cf1bc1
docker run --rm defanged-debian \
  find / -perm +6000 -type f -exec ls -ld {} \; 2> /dev/null | wc -l
0
$
```

Но более вероятно, что в используемом вами Dockerfile будут применяться исполняемые файлы с установленными битами setuid/setgid, а не ваше приложение. Поэтому всегда можно выполнить описанное выше действие ближе к концу файла, после всех вызовов команд, но перед изменением пользователя (исключение бинарных файлов с установленным битом setuid не имеет смысла, если приложение запускается с привилегиями суперпользователя root).

Ограничение использования оперативной памяти

Ограничение использования оперативной памяти защищает от DoS-атак и от приложений с утечками памяти, которые постепенно съедают всю доступную память на хосте (такие приложения могут автоматически перезапускаться, чтобы обеспечить требуемый уровень сервиса).

Флаги `-m` и `--memory-swap` в команде `docker run` ограничивают для контейнера объем оперативной памяти и свопа, который он может использовать. Аргумент `--memory-swap` устанавливает общий объем доступной памяти, то есть объем оперативной памяти плюс объем свопа, а не один только объем свопа, что приводит

к небольшой путанице. По умолчанию ограничения памяти не устанавливаются. Если флаг -m используется без флага --memory-swap, то значение для --memory-swap принимается как удвоенное значение аргумента -m. Это будет более понятно на следующем примере. Воспользуемся образом amouat/stress, в который включена Unix-утилита stress (http://people.seas.harvard.edu/~apw/stress/) для тестирования ситуаций, когда все ресурсы поглощаются одним процессом. В нашем случае определим захват заданного объема памяти:

```
$ docker run -m 128m --memory-swap 128m amouat/stress \
     stress --vm 1 --vm-bytes 127m -t 5s  ❶
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: info: [1] successful run completed in 5s
$ docker run -m 128m --memory-swap 128m amouat/stress \
     stress --vm 1 --vm-bytes 130m -t 5s  ❷
stress: FAIL: [1] (416) <-- worker 6 got signal 9
stress: WARN: [1] (418) now reaping child worker processes
stress: FAIL: [1] (422) kill error: No such process
stress: FAIL: [1] (452) failed run completed in 0s
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
$ docker run -m 128m amouat/stress \
     stress --vm 1 --vm-bytes 255m -t 5s  ❸
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: info: [1] successful run completed in 5s
```

❶ Эти аргументы позволяют утилите stress запустить один процесс, который захватывает 127 Мб оперативной памяти и завершается по тайм-ауту по истечении пяти секунд.

❷ В этот раз процесс пытается захватить 130 Мб памяти, и эта попытка завершается критической ошибкой, так как объем доступной памяти ограничен 128 Мб.

❸ Попытка захвата 255 Мб памяти, но здесь по умолчанию для флага --memory-swap определяется размер 256 Мб, поэтому команда завершается успешно.

Ограничение загрузки процессора

Если атакующий может захватить один контейнер или группу контейнеров, чтобы начать использование всех процессоров на данном хосте, то у него есть возможность лишить этого ресурса все прочие контейнеры хоста, то есть осуществить DoS-атаку.

В Docker совместное использование процессоров определяется относительным весовым коэффициентом со значением по умолчанию 1024, таким образом, по умолчанию все контейнеры получают одинаковое процессорное время.

Действие механизма распределения процессорного времени лучше всего пояснить на конкретном примере. Запустим четыре контейнера из образа amouat/stress, примененного в примере предыдущего раздела, но в этот раз контейнеры будут пытаться захватить как можно больше процессорного времени, а не памяти.

```
$ docker run -d --name load1 -c 2048 amouat/stress
912a37982de1d8d3c4d38ed495b3c24a7910f9613a55a42667d6d28e1da71fe5
```

```
$ docker run -d --name load2 amouat/stress
df69312a0c959041948857fca27b56539566fb5c7cda33139326f16485948bc8
$ docker run -d --name load3 -c 512 amouat/stress
c2675318fefafa3e9bfc891fa303a16e72caf221ec23a4c222c2b889ea82d6e2
$ docker run -d --name load4 -c 512 amouat/stress
5c6e199423b59ae481d41268c867c705f25a5375d627ab7b59c5fbfbcfc1d0e0
$ docker stats $(docker inspect -f {{.Name}} $(docker ps -q))
CONTAINER           CPU %    ...
/load1              392.13%
/load2              200.56%
/load3              97.75%
/load4              99.36%
```

В этом примере контейнеру `load1` назначается весовой коэффициент 2048, для контейнера `load2` сохраняется значение по умолчанию 1024, остальные два контейнера получают значение весового коэффициента 512. На моем компьютере процессор имеет 8 ядер, следовательно, распределяются 800% данного ресурса, таким образом, `load1` получает в свое распоряжение приблизительно половину, `load2` – четверть, а `load3` и `load4` по одной восьмой. Если бы работал только один контейнер, то он захватил бы столько ресурсов, сколько захотел.

Смысл относительного весового коэффициента заключается в том, чтобы не допустить захвата ресурса одним контейнером и лишения этого ресурса всех прочих контейнеров при параметрах, определенных по умолчанию. Тем не менее можно формировать группы контейнеров, которым будет отдаваться предпочтение при распределении процессорного ресурса, и в этом случае можно включать в эту группу контейнеры с пониженным коэффициентом по умолчанию, чтобы обеспечить справедливое распределение ресурса. При планировании совместного использования процессорного ресурса следует тщательно продумать значение весового коэффициента по умолчанию, чтобы контейнеры без явного указания этого коэффициента не ущемлялись другими контейнерами с высоким весовым коэффициентом.

Другим вариантом планирования совместного использования процессорного ресурса является применение абсолютно справедливого планировщика CFS (Completely Fair Scheduler) с помощью флагов `--cpu-period` и `--cpu-quota`. По этой методике для контейнеров устанавливается квота выделяемого процессорного ресурса (время, определяемое в микросекундах), которым они пользуются в течение заданного интервала времени. Если контейнер превышает квоту в рамках своего рабочего интервала, то он должен ждать следующего, для того чтобы продолжить свое выполнение. Например:

```
$ docker run -d --cpu-period=50000 --cpu-quota=25000 myimage
```

Этому контейнеру разрешено использовать половину процессорного ресурса каждые 50 мс, предполагая наличие одного процессорного устройства в системе. Более подробно о планировщике CFS можно узнать из документации по ядру Linux (https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt).

Ограничение возможности перезапуска

Если контейнер постоянно останавливается и перезапускается, то он расходует большое количество системного времени и ресурсов, что может привести к отказу в обслуживании (DoS). Ситуация легко исправляется с помощью установки стратегии перезапуска `on-failure` вместо стратегии `always`. Например:

```
$ docker run -d --restart=on-failure:10 my-flaky-image
…
```

Docker получает указание перезапускать данный контейнер не более 10 раз. Текущее значение счетчика можно узнать из параметра `.RestartCount` в метаданных, возвращаемых командой `docker inspect`:

```
$ docker inspect -f "{{ .RestartCount }}" $(docker ps -lq)
0
```

В Docker применяется экспоненциальное увеличение задержки при перезапуске контейнеров (контейнер будет ждать перезапуска 100 мс, потом 200 мс, потом 400 мс и т. д. при всех последующих перезапусках). Эта мера выглядит достаточно эффективной для предотвращения DoS-атак, при которых атакующий пытается воспользоваться возможностью многократного перезапуска контейнеров.

Ограничения файловых систем

Строгое запрещение записи в файл предотвращает некоторые типы атак и существенно снижает возможности взломщиков. Они не могут записать свой скрипт в файл и запустить его вместо вашего приложения или изменить важные данные или файлы конфигурации.

Начиная с Docker версии 1.5, в команду `docker run` можно передавать флаг `--read-only`, полностью защищающий от записи файловую систему внутри контейнера:

```
$ docker run --read-only debian touch x
touch: cannot touch 'x': Read-only file system
```

Нечто похожее можно сделать и для томов, если добавить ключ `:ro` в конец аргумента тома:

```
$ docker run -v $(pwd):/pwd:ro debian touch /pwd/x
touch: cannot touch '/pwd/x': Read-only file system
```

Большинству приложений необходима запись в файлы, они не могут работать в среде с полной защитой от записи. В таких случаях можно определить каталоги и файлы, для которых необходима операция записи из приложения, и использовать отдельные тома для монтирования только этих файлов.

Применение такого подхода дает большие преимущества при контрольных проверках: если я уверен в том, что файловая система контейнера в точности совпадает с файловой системой образа, из которого был создан этот контейнер, я могу выполнить единственную офлайновую контрольную проверку данного образа вместо многочисленных контрольных проверок каждого отдельного контейнера.

Ограничение использования механизма Capabilities

В ядре Linux определены наборы привилегий, называемые Capabilities, которые могут назначаться процессам с целью ограничения их доступа к системным функциям. Эти параметры охватывают широкий диапазон функций, от изменения системного времени до открытия сетевых сокетов. Раньше процесс или получал полностью все привилегии суперпользователя root, или оставался чисто пользовательским, без каких-либо промежуточных состояний. Такое жесткое разграничение было особенно неудобным при использовании приложений, подобных ping, которому требуются привилегии root только для того, чтобы открыть простейший (так называемый «сырой» (raw)) сетевой сокет. Это означало, что небольшая ошибка в утилите ping могла бы позволить атакующему получить все права root в системе. С созданием механизма Capabilities появилась возможность создания версии ping, обладающей только теми привилегиями, которые необходимы для создания сетевого сокета, но не всеми привилегиями root. Теперь атакующий при использовании уязвимостей в отдельной утилите получал гораздо меньше свободы действий.

По умолчанию контейнеры Docker запускаются с некоторым Capabilities[1]. Например, контейнеру обычно запрещено использование таких устройств, как графический процессор (GPU) или звуковая карта, а также добавление модулей ядра. Чтобы предоставить контейнеру расширенные привилегии, нужно запустить его с аргументом `--privileges` в команде `docker run`.

С точки зрения обеспечения безопасности необходимо максимально ограничить количество Capabilities, разрешенных для контейнеров. Управлять Capabilities, доступными для конкретного контейнера, можно с помощью аргументов `--cap-add` и `--cap-drop`. Например, если нужно изменить системное время (не выполняйте этот пример, если не хотите нарушить работу своей системы):

```
$ docker run debian date -s "10 FEB 1981 10:00:00"
Tue Feb 10 10:00:00 UTC 1981
date: cannot set date: Operation not permitted
$ docker run --cap-add SYS_TIME debian date -s "10 FEB 1981 10:00:00"
Tue Feb 10 10:00:00 UTC 1981
$ date
Tue Feb 10 10:00:03 GMT 1981
```

В этом примере изменение даты и времени невозможно до тех пор, пока не добавлена привилегия `SYS_TIME` для данного контейнера. Поскольку системное время является характеристикой ядра вне каких-либо пространств имен, установка времени внутри контейнера приводит к соответствующему изменению времени для всего хоста и всех прочих контейнеров[2].

---

[1] Включены параметры `CHOWN`, `DAC_OVERRIDE`, `FSETID`, `FOWNER`, `MKNOD`, `NET_RAW`, `SETGID`, `SETUID`, `SETFCAP`, `NET_BIND_SERVICE`, `SYS_CHROOT`, `KILL` и `AUDIT_WRITE`. Среди исключенных следует отметить `SYS_TIME`, `NET_ADMIN`, `SYS_MODULE`, `SYS_NICE` и `SYS_ADMIN` (но это не полный список). Полную информацию о механизме Capabilities можно получить из руководства `man capabilities`.

[2] Если вы выполните этот пример, то работа вашей системы будет нарушена до тех пор, пока не будет установлено правильное время. Для возврата корректного системного времени попробуйте выполнить команду `sudo ntpdate-debian`.

При более осторожном подходе следует отменить все привилегии, а затем добавить только те, которые действительно необходимы:

```
$ docker run --cap-drop all debian chown 100 /tmp
chown: changing ownership of '/tmp': Operation not permitted
$ docker run --cap-drop all --cap-add CHOWN debian chown 100 /tmp
```

Такой способ существенно укрепляет защиту системы: после взлома контейнера атакующий сильно ограничен в выборе доступных ему системных вызовов (обращающихся непосредственно к ядру). Но при этом остаются нерешенными следующие проблемы:

- как узнать, какие привилегии можно исключить без ущерба для функциональности? Самый простой подход – метод проб и ошибок, но что, если случайно будет исключен параметр capabilities, который требуется в приложении крайне редко? Определение требуемых привилегий упрощается, если имеется в наличии полный комплект тестов, которым можно проверить контейнер. Следует рассмотреть возможность использования методики микросервисов, при которой уменьшается объем исходного кода и количество изменяемых элементов в каждом контейнере;
- параметры capabilities не сгруппированы и не классифицированы так, как нам бы хотелось. Например, параметр SYS_ADMIN определяет большой объем функциональности, и создается впечатление, что разработчики ядра использовали его по умолчанию, когда не нашли (или даже не потрудились искать) лучшую альтернативу. В результате обычный бинарный файл может оказаться втянутым в конфликт между правами администратора и обычного пользователя, но именно для устранения подобной ситуации и были введены параметры capabilities .

Ограничение ресурсов (ulimits)

В ядре Linux определены ограничения ресурсов, которые могут применяться к процессам, например ограничение на количество создаваемых процессов-потомков и на количество дескрипторов открытых файлов. Эти ограничения можно использовать и для контейнеров Docker с помощью флага `--ulimit` в команде `docker run` или посредством установки ограничений по умолчанию при передаче аргумента `--default-ulimit` при запуске демона Docker. Аргумент включает два значения: мягкое ограничение и жесткое ограничение, разделенные двоеточием. Воздействие этого аргумента зависит от заданного ограничения. Если указано только одно значение, то оно используется и для мягкого, и для жесткого ограничений.

Полный набор возможных значений и их подробное описание приведено в руководстве `man setrlimit` (но следует отметить, что ограничение `as` не может использоваться для контейнеров). Для нас особый интерес представляют следующие значения:

- *cpu* – ограничение использования процессорного времени определяется в секундах. Применяется мягкое ограничение (после чего контейнеру по-

сылается сигнал SIGXCPU), с последующей передачей сигнала SIGKILL при достижении значения жесткого ограничения. Для примера, как и в нескольких предыдущих разделах, снова воспользуемся утилитой stress для попытки максимального использования процессорного времени:

```
$ time docker run --ulimit cpu=12:14 amouat/stress stress --cpu 1
stress: FAIL: [1] (416) <-- worker 5 got signal 24
stress: WARN: [1] (418) now reaping child worker processes
stress: FAIL: [1] (422) kill error: No such process
stress: FAIL: [1] (452) failed run completed in 12s
stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd

real  0m12.765s
user  0m0.247s
sys 0m0.014s
```

Аргумент ulimit позволяет принудительно завершить работу контейнера после 12-секундного интервала использования процессорного времени.
Это может оказаться полезным для ограничения использования ресурса процессора контейнерами, запущенными другими процессами (например, при выполнении вычислений от имени пользователей). В таких условиях этот способ ограничения процессорного времени может стать достаточно эффективным средством против DoS-атак;

- *nofile* – максимальное количество дескрипторов файлов[1], которые могут быть открыты в контейнере. И это ограничение можно использовать для защиты от DoS-атак, поскольку для атакующего ограничиваются операции чтения и записи в контейнере или на томах. (Отметим, что для nofile необходимо устанавливать значение на единицу больше, чем действительно требуемое максимальное количество дескрипторов.) Например:

```
$ docker run --ulimit nofile=5 debian cat /etc/hostname
b874469fe42b
$ docker run --ulimit nofile=4 debian cat /etc/hostname
Timestamp: 2015-05-29 17:02:46.956279781 +0000 UTC
Code: System error

Message: Failed to open /dev/null - open /mnt/sda1/var/lib/docker/aufs...
```

Здесь операционная система требует нескольких дескрипторов для открытых файлов, хотя для утилиты cat нужен только один дескриптор. Трудно заранее определить, сколько потребуется дескрипторов файлов для конкретного приложения, но установка разумного их количества с небольшим запасом обеспечивает некоторую степень защиты от DoS-атак, по сравнению со значением по умолчанию 1048576;

---

[1] Дескриптор файла (file descriptor) – это указатель на запись в таблице, содержащей информацию об открытых файлах в системе. Запись создается при доступе к файлу, в ней фиксируются режим доступа (чтение, запись и т. д.) и прочие параметры.

- *nproc* – максимальное количество процессов, которое может быть создано пользователем данного контейнера. Это полезное ограничение, поскольку оно может защитить систему от форк-бомб и некоторых других типов атак. К сожалению, ограничение `nproc` не устанавливается для каждого отдельного контейнера, а накладывается на все процессы пользователя данного контейнера. Например:

```
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
92b162b1bb91af8413104792607b47507071c52a2e3128f0c6c7659bfbb84511
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
158f98af66c8eb53702e985c8c6e95bf9925401c3901c082a11889182bc843cb
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
6444e3b5f97803c02b62eae601fbb1dd5f1349031e0251613b9ff80871555664
FATA[0000] Error response from daemon: Cannot start container 6444e3b5f9780...
[8] System error: resource temporarily unavailable
$ docker run --user 500 -d debian sleep 100
f740ab7e0516f931f09b634c64e95b97d64dae5c883b0a349358c5995806e503
```

  Третий контейнер запустить невозможно, так как уже имеются два процесса, принадлежащих пользователю с UID 500. Но если просто убрать аргумент `--ulimit`, то этот пользователь получает возможность создавать любое количество процессов. Несмотря на этот существенный недостаток, ограничение `nproc` остается весьма полезным в тех случаях, когда один пользователь является владельцем ограниченного количества контейнеров.

Кроме того, следует отметить, что ограничение `nproc` невозможно установить для суперпользователя root.

Использование защищенного ядра

Кроме мер по поддержанию операционной системы хоста в актуальном состоянии с применением всех последних исправлений, может потребоваться использование специально модифицированного, защищенного ядра, с изменениями от grsecurity (https://grsecurity.net/) и PaX (https://pax.grsecurity.net/). PaX предоставляет дополнительную защиту от атак, направленных на выполняющиеся в текущий момент программы с целью изменения оперативной памяти (например, атаки, связанные с переполнением буфера). Защита реализована посредством маркировки программного кода в оперативной памяти как неизменяемого, а данных – как невыполняемых. В дополнение к этому области памяти выделяются случайным образом, чтобы затруднить атаки, пытающиеся изменить адресацию кода существующих процедур (например, системных вызовов в библиотеках общего пользования). Средство grsecurity предназначено для совместного использования с PaX, оно добавляет корректировки, позволяющие организовать управление доступом на основе ролей (role-based access control – RBAC), оперативные проверки и многие другие функциональные возможности.

Чтобы воспользоваться PaX и/или grsecurity, вероятнее всего, потребуется ввести все соответствующие изменения и самостоятельно скомпилировать моди-

фицированное ядро. Это не так трудно, как кажется на первый взгляд, и тем, кто делает это впервые, поможет множество сетевых информационных ресурсов (от WikiBooks (https://www.wikibooks.org/) до InsanityBit (http://www.insanitybit.com/)).

Использование этих дополнительных средств защиты может привести к тому, что некоторые приложения перестанут работать. PaX конфликтует со всеми программами, которые генерируют код во время выполнения. Кроме того, с дополнительными проверками и действиями по обеспечению безопасности связаны небольшие накладные расходы. И последнее: если вы используете готовое защищенное ядро, то необходимо убедиться в том, что оно в полной мере поддерживает ту версию Docker, с которой вы намерены работать.

Модули безопасности Linux

В ядре Linux определен интерфейс модулей безопасности Linux Security Module (LSM), который реализуется различными модулями, устанавливающими конкретную стратегию защиты. Во время написания книги существовало несколько реализаций, в том числе AppArmor, SELinux, Smack и TOMOYO Linux. Эти модули защиты можно использовать для создания еще одного уровня контроля прав доступа процессов и пользователей, в дополнение к стандартным средствам управления доступом на уровне файлов.

Вместе с Docker обычно используются модули SELinux (в основном в дистрибутивах, основанных на Red Hat) и AppArmor (как правило, в дистрибутивах на основе Ubuntu и Debian). В следующих разделах подробно рассматриваются оба этих модуля.

SELinux

SELinux, или Security Enhanced Linux, – это модуль, разработанный Агентством национальной безопасности США как реализация методики, названной мандатным контролем доступа (Mandatory Access Control – MAC), как противоположность принятой в Unix модели дискретного контроля доступа (Discretionary Access Control – DAC). Если говорить упрощенно, то между системой управления доступом SELinux и стандартной системой управления доступом Unix существуют два главных различия:

- средства контроля доступа модуля SELinux основаны на типах (types), по существу являющихся метками, присваиваемыми процессам и объектам (файлам, сокетам и т. п.). Если стратегия SELinux запрещает процессу типа А доступ к объекту типа Б, то доступ будет запрещен полностью, независимо от прав доступа на файл, установленных для данного объекта, и от привилегий доступа, назначенных данному пользователю. Проверки SELinux выполняются после обычных системных процедур проверки прав доступа к файлу;
- возможно применение нескольких уровней защиты, в некоторой степени напоминающих модель, используемую в государственном управлении: конфиденциальный (confidential), секретный (secret) и совершенно секретный (top

secret) уровни доступа. Процессы, принадлежащие к более низким уровням, не могут читать файлов, записанных процессами более высоких уровней, вне зависимости от расположения файлов в файловой системе и установленных прав доступа к этим файлам. Таким образом, процесс уровня «совершенно секретно» может записать файл в каталог /tmp с установкой прав доступа chmod 777, но процесс уровня «конфиденциальный» не сможет получить доступа к этому файлу. В SELinux это называется многоуровневой защитой (Multi-level Security – MLS), которая тесно связана с концепцией защиты на основе набора категорий (Multi-Category Security – MCS). В соответствии с концепцией MCS процессам и объектам присваиваются определенные категории, и доступ к ресурсам запрещается, если процесс или объект не принадлежит к правильной категории. В отличие от многоуровневой защиты, категории не перекрываются и не представляют собой иерархическую структуру. Защита на основе категорий может использоваться для ограничения доступа к ресурсам определенного подмножества некоторого типа (например, используя уникальную категорию, можно так ограничить доступ к ресурсу, чтобы его мог использовать только один процесс).

SELinux устанавливается по умолчанию в дистрибутивах Red Hat, но его несложно установить и в большинстве других дистрибутивов. Проверить, работает ли модуль SELinux, можно с помощью команды `sestatus`. Если команда существует, то она сообщит, разрешена или запрещена работа модуля SELinux и какой режим установлен для него – рекомендующий (permissive) или строгий (enforcing). В рекомендующем режиме SELinux фиксирует в системном журнале все нарушения правил управления доступом, но самого доступа не запрещает.

По умолчанию стратегия SELinux для Docker формируется таким образом, чтобы обеспечить защиту хоста от контейнеров, а также защиту контейнеров друг от друга. По умолчанию контейнерам назначается тип процесса `svirt_lxc_net_t`, а файлам, доступным для контейнеров, назначается тип `svirt_sandbox_file_t`. Стратегия определяет правила, в соответствии с которыми контейнерам разрешается только читать и выполнять файлы из каталога */usr* на хосте, но при этом полностью запрещена запись каких-либо файлов в файловой системе хоста. Каждому контейнеру также присваивается уникальный номер категории по модели MCS, чтобы исключить возможность доступа к файлам или ресурсам, с которыми работают (на запись) другие контейнеры в случае аварийного завершения их работы.

Включение SELinux

При использовании дистрибутива на основе Red Hat SELinux устанавливается по умолчанию. Проверить, разрешена ли его работа и установлен ли строгий режим, можно при помощи выполнения в командной строке команды `sestatus`. Чтобы разрешить работу SELinux и установить строгий режим, необходимо отредактировать файл конфигурации /etc/selinux/config, добавив в него строку `SELINUX=enforsing`.

Кроме того, необходимо проверить, разрешена ли поддержка SELinux в демоне Docker. Демон должен быть запущен с флагом `--selinux-enabled`. В противном случае этот флаг нужно добавить в файл /etc/sysconfig/docker.

Для использования SELinux обязательным является драйвер файловой системы devicemapper. Во время написания книги велась работа по обеспечению совместного использования SELinux вместе с драйверами Overlay и BTRFS, но полная совместимость пока не достигнута.
Для установки SELinux в других дистрибутивах изучите соответствующую документацию. Обратите внимание на то, что SELinux должен пометить все файлы в файловой системе, для чего потребуется некоторое время. Не следует легкомысленно относиться к установке SELinux.

Разрешение работы SELinux дает немедленный и весьма заметный эффект при использовании контейнеров с томами. При установленном и разрешенном модуле SELinux операции чтения и записи на томах по умолчанию запрещены:

```
$ sestatus | grep mode
Current mode:    enforcing
$ mkdir data
$ echo "hello" > data/file
$ docker run -v $(pwd)/data:/data debian cat /data/file
cat: /data/file: Permission denied
```

Причину этого можно выяснить, изучив контекст обеспечения безопасности данного каталога:

```
$ ls --scontext data
unconfined_u:object_r:user_home_t:s0 file
```

Метка для данных не совпадает с меткой для контейнеров. Это можно исправить, определив метку контейнера для требуемых данных с помощью утилиты `chcon`, при этом система немедленно оповещается о том, что указанные файлы будут использоваться контейнерами с такой же меткой:

```
$ chcon -Rt svirt_sandbox_file_t data
$ docker run -v $(pwd)/data:/data debian cat /data/file
hello
$ docker run -v $(pwd)/data:/data debian sh -c 'echo "bye" >> /data/file'
$ cat data/file
hello
bye
$ ls --scontext data
unconfined_u:object_r:svirt_sandbox_file_t:s0 file
```

Отметим, что если выполнить команду `chcon` только для файла, но не для содержащего его каталога, то разрешается лишь читать этот файл, а операции записи в него будут запрещены.

В версии 1.7 и последующих Docker автоматически изменяет метки томов для использования в контейнерах, если при монтировании тома задан суффикс `:Z` или `:z`. Том с меткой `:z` могут использовать все контейнеры (это необходимо для контейнеров данных, которые совместно используют тома вместе с несколькими дру-

гими контейнерами), а метка :Z обозначает том, который разрешено использовать только данному контейнеру. Например:

```
$ mkdir new_data
$ echo "hello" > new_data/file
$ docker run -v $(pwd)/new_data:/new_data debian cat /new_data/file
cat: /new_data/file: Permission denied
$ docker run -v $(pwd)/new_data:/new_data:Z debian cat /new_data/file
hello
```

Также можно воспользоваться флагом --security-opt для изменения метки контейнера или для запрещения присваивания метки данному контейнеру:

```
$ touch newfile
$ docker run -v $(pwd)/newfile:/file --security-opt label:disable \
debian sh -c 'echo "hello" > /file'
$ cat newfile
hello
```

Интересен вариант использования меток SELinux для присваивания особой метки контейнеру с целью назначения конкретной стратегии обеспечения безопасности. Например, можно создать политику для контейнера Nginx, которая разрешает передачу данных только через порты 80 и 443.

Помните о невозможности выполнять команды SELinux внутри контейнеров. Внутри контейнеров SELinux отключен, чтобы предотвратить попытки приложений и пользователей воспользоваться командами установки стратегий SELinux, которые могут нарушить или заблокировать работу SELinux на хосте.

Помощь в разработке политик SELinux могут оказать разнообразные инструментальные средства и статьи. Например, полезна утилита audit2allow, которая может переключать сообщения в системных журналах с рекомендующего режима на стратегии, позволяющие запускать приложения в строгом режиме без нарушения их функциональности.

Будущее SELinux выглядит многообещающим: множество флагов и реализаций по умолчанию добавляется в Docker, и это значительно упрощает развертывание конфигураций, защищенных модулем SELinux. Функциональность MCS должна обеспечить создание «секретных» и «совершенно секретных» контейнеров для обработки важных данных с помощью одного простого флага. К сожалению, практический опыт применения пользователями модуля SELinux пока невелик – новичков отпугивает часто выводимое сообщение SELinux «Доступ запрещен» («Permission Denied»), они считают, что система испорчена, не понимают, что происходит, и не знают, как исправить ситуацию. Разработчики предпочитают отключать SELinux, и снова возвращается проблема различия среды разработки и среды эксплуатации, одна из основных проблем, которую стремится решить Docker. Если необходима дополнительная защита, предоставляемая модулем SELinux, то вы должны стойко переносить все тяготы и неудобства текущего положения вещей и ждать улучшения ситуации.

AppArmor

Преимуществом и в то же время недостатком модуля AppArmor является то, что он намного проще модуля SELinux. Этот модуль должен просто работать, не мешая пользователю, но он не способен предоставить тот уровень детализации степеней защиты, который предлагает SELinux. Работа AppArmor состоит в применении к процессам профилей, ограничивающих привилегии на уровне системных параметров ядра Linux и прав доступа к файлам.

Если вы используете хост под управлением Ubuntu, то вероятнее всего, что AppArmor уже установлен и работает. Это можно проверить командой `sudo apparmor_status`. Docker автоматически применяет профиль AppArmor к каждому запускаемому контейнеру. По умолчанию профиль предоставляет некоторый уровень защиты от контейнеров-мошенников, пытающихся получить доступ к разнообразным системным ресурсам. Обычно профиль размещается в подкаталоге */etc/apparmor/docker*. Во время написания книги изменять профиль по умолчанию было запрещено, поскольку демон Docker перезаписывает его во время перезагрузки.

Если AppArmor мешает запуску контейнера, то этот модуль можно отключить, передавая аргумент `--security-opt="apparmor:unconfined"` в команду `docker run`. Другой профиль для контейнера определяется с помощью аргумента `--security-opt="apparmor:PROFILE"` в команде `docker run`, где `PROFILE` – имя файла, содержащего профиль защиты, предварительно загруженный модулем AppArmor.

Проведение контрольных проверок

Проведение регулярных контрольных проверок и обзоров контейнеров и образов – хороший способ убедиться в том, что система сохраняет работоспособность и актуальность, а кроме того, это дополнительная проверка на отсутствие уязвимостей в подсистеме защиты. Контрольная проверка системы, основанной на контейнерах, должна включать проверку актуальности образов для всех запускаемых контейнеров, а также актуальность и безопасность ПО, используемого в этих образах. Любое различие между контейнером и соответствующим ему образом должно быть зафиксировано и обосновано. Кроме того, контрольные проверки должны охватывать области, не относящиеся напрямую к контейнерным системам, такие как проверка доступа к системным журналам, права доступа к файлам и обеспечение целостности данных. Если процедуры контрольных проверок в достаточной степени автоматизированы, то их можно проводить регулярно, чтобы как можно быстрее выявлять все возникающие проблемы.

Вместо фиксации в журналах событий и исследования записей журналов для каждого контейнера в отдельности можно выполнить контрольную проверку образа, из которого создан данный контейнер, затем воспользоваться командой `docker diff` для обнаружения любых отклонений от образа. Проверка будет даже более надежной, если используется защищенная от записи файловая система (см. раздел «Ограничения файловых систем» выше), поскольку дает уверенность в том, что в проверяемом контейнере ничего не изменяется.

По меньшей мере, следует проверить актуальность версий используемых программ и наличие в них исправлений, связанных с обеспечением безопасности. Такая проверка должна проводиться для каждого образа, все файлы которого проверяются на предмет изменений с помощью команды `docker diff`. Если используются тома, то необходимо проверить все соответствующие каталоги.

Объем работ, связанных с контрольными проверками, можно существенно сократить, если использовать минималистичные образы, содержащие только те файлы и библиотеки, которые действительно необходимы и важны для данного приложения.

Система хоста также требует контрольных проверок, проводимых таким образом, как если бы это был обычный компьютер или виртуальная машина. Убедиться в том, что в ядро внесены все необходимые корректировки, – это самая важная проверка в системе, в которой многие контейнеры совместно используют функциональность этого ядра.

В настоящее время доступны некоторые инструментальные средства для проведения контрольных проверок контейнерных систем, и в самое ближайшее время можно ожидать пополнения этого набора. Компания Docker представляет инструмент Docker Bench for Security (https://dockerbench.com/), который проверяет ПО на совместимость в соответствии с набором рекомендаций из документа Docker Benchmark, созданного организацией Center for Internet Security (CIS) (https://benchmarks.cisecurity.org/). Средство выполнения контрольных проверок Lynis (https://cisofy.com/lynis/) с открытым исходным кодом предлагает несколько тестов, проверяющих работу контейнеров Docker.

Реакция на нестандартные ситуации

В случае возникновения нестандартной ситуации вы можете воспользоваться некоторыми функциональными возможностями Docker, чтобы быстро отреагировать на возникшую проблему и определить ее причину. Команда `docker commit` используется для оперативного получения мгновенного снимка поврежденной системы, а команды `docker diff` и `docker logs` помогут обнаружить изменения, внесенные взломщиком.

При изучении нарушений в поведении контейнера главным вопросом, требующим немедленного ответа, становится следующий: «Действительно ли произошел взлом контейнера?» (то есть смог ли атакующий получить прямой доступ к хосту?). Если подтверждена высокая вероятность взлома, то для хоста необходимо выполнить «дезинфицирующие» процедуры, после чего все контейнеры должны быть перезапущены из «чистых» образов (даже при отсутствии явных признаков атаки). Если вы уверены в том, что атака изолирована в конкретном контейнере, то можете просто остановить работу этого контейнера и заменить его. (*Никогда* не возвращайте пораженный контейнер в сервис, даже если он содержит данные или изменения, которых нет в базовом образе, – такому контейнеру уже нельзя доверять ни при каких условиях.)

Эффективным способом предотвращения атак может стать любое ограничение функциональности контейнера, например отключение всех системных параметров ядра или использование защищенной от записи файловой системы.

При возникновении нестандартной ситуации и/или выявления факта атаки какого-либо типа поврежденный образ следует подвергнуть всестороннему анализу с целью точного определения причин и области воздействия данной атаки.

Разработка эффективной стратегии обеспечения безопасности с методиками оперативного реагирования на нестандартные ситуации более подробно описана в документе CERT «Steps for Recovering from a UNIX or NT System Compromise» (https://www.cert.org/historical/tech_tips/win-UNIX-system-compromise.cfm) и в материалах сайта ServerFault (https://serverfault.com/questions/218005/how-do-i-deal-with-a-compromised-server).

Функциональные возможности будущих версий

Некоторые функциональные возможности Docker, связанные с обеспечением безопасности, пока находятся в разработке. Поскольку этим функциям в компании Docker уделяется особое внимание, можно надеяться, что они будут реализованы в ближайшее время, может быть, даже к моменту выхода из печати данной книги. Среди этих функциональных возможностей в первую очередь отметим следующие:

- *seccomp* – функция ядра Linux seccomp (или *secure computing mode*) используется для ограничения в процессе возможностей выполнения системных вызовов. Чаще всего seccomp применяется в веб-браузерах, в том числе в Chrome и Firefox, для размещения подключаемых модулей в так называемой «песочнице» (sandbox). Объединив seccomp с Docker, можно обеспечить для контейнеров блокировку определенного набора системных вызовов. Предполагается, что внедренный в Docker механизм seccomp по умолчанию будет блокировать 32-битовые системные вызовы, устаревшие сетевые функции, а также различные системные функции, которые обычно не требуются контейнерам. Кроме того, любые другие системные вызовы можно будет явно запрещать или разрешать во время выполнения. Например, следующая команда разрешает контейнеру выполнить системный вызов clock_adjtime для синхронизации системного времени по протоколу Network Time Protocol:

```
$ docker run -d --security-opt seccomp:allow:clock_adjtime ntpd
```

- *пространство имен пользователей* – ранее уже обсуждались вопросы решения проблемы, связанной с пространством имен пользователей, в основном относящейся к работе с привилегиями суперпользователя root. Вскоре ожидается обеспечение поддержки преобразования суперпользователя root внутри контейнера в обычного непривилегированного пользователя на хосте.

В дополнение к этим средствам я хотел бы увидеть полностью укомплектованный набор различных инструментальных средств обеспечения безопасности, специализированных для Docker, возможно, в форме профиля безопасности для

контейнеров. В настоящее время существует большое количество инструментов и методик с перекрывающимися функциональными возможностями (например, доступ к файлам может регулироваться с помощью модуля SELinux, запрещения или разрешения системных параметров ядра или применения флага `--read-only`).

Резюме

В этой главе мы увидели, как много вопросов необходимо решить для обеспечения безопасности системы. Главная рекомендация – соблюдать принципы глубокой защиты и предоставления минимальных привилегий. При таком подходе атакующий, даже получив управление одним компонентом системы, не сможет получить полного доступа к системе в целом. Взломщику придется преодолеть следующие линии защиты, прежде чем он получит возможность нанести существенный ущерб системе или получить доступ к секретным данным.

Группы контейнеров, принадлежащие различным пользователям или обрабатывающие особо важные данные, должны запускаться на отдельных виртуальных машинах, изолированных от контейнеров, принадлежащих другим пользователям или обеспечивающих работу открытых общедоступных интерфейсов. Порты, предъявляемые контейнерами, должны быть скрыты за шлюзами, особенно если они открываются для внешней сети, но и во внутренних сетях следует ограничивать доступ, чтобы изолировать опасные контейнеры. Ресурсы и функциональность, доступные контейнерам, также необходимо ограничивать только тем объемом, который действительно требуется для выполнения поставленных задач, устанавливая ограничения на используемую оперативную память, на доступ к файлам, на применение системных параметров ядра. Усилить защиту можно на уровне ядра, применяя специально модифицированные ядра и модули обеспечения безопасности, такие как AppArmor и SELinux.

Кроме того, атаки можно обнаруживать на ранней стадии, если воспользоваться средствами контроля и отслеживания. Инструменты контрольных проверок особенно полезны для систем, основанных на контейнерах, поскольку контейнеры легко сравнивать с образами, из которых они были созданы, для выявления подозрительных изменений. В свою очередь, образы должны проверяться в офлайн-режиме для полной уверенности в том, что они используют актуальные и безопасные версии ПО. Подозрительные контейнеры с неопределенным состоянием следует немедленно заменять новыми версиями.

Контейнеры являются полезным средством с точки зрения обеспечения безопасности, поскольку предоставляют дополнительный уровень изоляции и управления. Система, правильно использующая контейнеры, будет более безопасной и защищенной, чем аналогичная система без применения контейнеров.

Предметный указатель

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**.

Эдриен Моуэт

Использование Docker

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Научный редактор *Маркелов А. А.*
Перевод *Снастин А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 33,1875. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

«Использование Docker» — глубокий, всеобъемлющий обзор программной среды Docker и экосистемы контейнеров. Практическая направленность книги в сочетании с большим количеством примеров упрощает применение теоретических концепций и методик в реальных проектах.»
— Пини Резник,
главный инженер компании Container Solutions

Контейнеры Docker предоставляют простые быстрые и надёжные методы разработки, распространения и запуска программного обеспечения, особенно в динамических и распределённых средах. Из книги вы узнаете, почему контейнеры так важны, какие преимущества вы получите от применения Docker и как сделать Docker частью процесса разработки. Вы последовательно пройдете по всем этапам, необходимым для создания, тестирования и развёртывания любого веб-приложения, использующего Docker. Также вы изучите обширный материал — начиная от основ, необходимых для запуска десятка контейнеров, и заканчивая описанием сопровождения крупной системы с множеством хостов в сетевой среде со сложным режимом планирования.
Издание предназначено разработчикам, инженерам по эксплуатации и системным администраторам.

В книге рассматриваются следующие темы:

- начало работы с Docker — создание и развёртывание простого веб-приложения;
- использование методик непрерывного развёртывания для продвижения вашего приложения к активному промышленному использованию несколько раз в день;
- изучение различных возможностей и методик для регистрации в системных журналах и наблюдения за многочисленными контейнерами;
- исследование сетевой среды и сетевых сервисов: как контейнеры находят друг друга и каким образом можно установить соединение между ними;
- распределение и организация кластеров контейнеров с целью балансировки нагрузки, масштабирования, устранения критических сбоев и планирования;
- обеспечение безопасности системы, следуя принципам «глубокой или многоуровневой защиты» и минимальных привилегий;
- применение контейнеров для построения архитектуры микросервисов.

Эдриен Моуэт *является руководителем отдела научных исследований в компании Container Solutions. Он занимается многими программными проектами — от небольших веб-приложений до крупномасштабного программного обеспечения для анализа больших объёмов данных.*

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа:
“Альянс-книга”
тел.(499)782-38-89
books@alians-kniga.ru

O’REILLY®

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-426-7

9 785970 604267 >