Бесплатная электронная книга

учусь Docker

Free unaffiliated eBook created from
Stack Overflow contributors.

#docker

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: docker

It is an unofficial and free Docker ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Docker.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Docker

замечания

Docker - это проект с открытым исходным кодом, который автоматизирует развертывание приложений внутри программных контейнеров. Эти контейнеры приложений аналогичны легким виртуальным машинам, так как они могут выполняться изолированно друг от друга и с хостом.

Docker требует функций, присутствующих в последних ядрах Linux, для правильной работы, поэтому на компьютерах Mac OSX и Windows виртуальная машина, работающая под Linux, требуется для правильной работы докера. В настоящее время основным методом установки и настройки этой виртуальной машины является Docker Toolbox, который использует VirtualBox внутренне, но есть планы интегрировать эту функциональность в сам докер, используя собственные функции виртуализации операционной системы. На Linux-сервере докеры запускаются изначально на самом хосте.

Версии

Версия	Дата выхода
17.05.0	2017-05-04
17.04.0	2017-04-05
17.03.0	2017-03-01
1.13.1	2016-02-08
1.12.0	2016-07-28
1.11.2	2016-04-13
1.10.3	2016-02-04
1.9.1	2015-11-03
1.8.3	2015-08-11
1.7.1	2015-06-16
1.6.2	2015-04-07
1.5.0	2015-02-10

Examples

Установка Docker в Mac OS X

Требования: OS X 10.8 «Горный лев» или более новый, необходимый для запуска Docker.

Хотя двоичный файл docker может запускаться изначально на Mac OS X, для сборки и размещения контейнеров вам нужно запустить виртуальную машину Linux на коробке.

1.12.0

Начиная с версии 1.12 вам не нужно устанавливать отдельную виртуальную машину, поскольку Docker может использовать встроенную функциональность `Hypervisor.framework` OSX для запуска небольшой Linux-машины, которая будет выступать в качестве бэкэнд.

Чтобы установить докер, выполните следующие действия:

1. Перейти к Docker для Mac
2. Загрузите и запустите программу установки.
3. Продолжайте установку с помощью параметров по умолчанию и введите учетные данные своей учетной записи по запросу.

Проверьте здесь для получения дополнительной информации по установке.

1.11.2

До версии 1.11 лучшим способом запуска этой виртуальной машины Linux является установка Docker Toolbox, которая устанавливает Docker, VirtualBox и гостевую машину Linux.

Чтобы установить панель инструментов докеров, выполните следующие действия:

1. Перейти к панели инструментов Docker
2. Нажмите ссылку для Mac и запустите программу установки.
3. Продолжайте установку с помощью параметров по умолчанию и введите учетные данные своей учетной записи по запросу.

Это установит двоичные файлы Docker в `/usr/local/bin` и обновит любую существующую установку Virtual Box. Проверьте здесь для получения дополнительной информации по установке.

Для проверки установки:

1.12.0

1. Запустите `Docker.app` из папки «Приложения» и убедитесь, что он запущен. Затем откройте терминал.

1.11.2

1. Откройте `Docker Quickstart Terminal` , который откроет терминал и подготовит его для использования для команд Docker.

2. Как только терминал открыт,

```
$ docker run hello-world
```

3. Если все хорошо, тогда это должно напечатать приветственное сообщение, подтверждающее успешную установку.

Установка Docker в Windows

Требования: 64-разрядная версия Windows 7 или выше на машине, которая поддерживает технологию виртуализации оборудования и включена.

Хотя двоичный файл docker может запускаться изначально на Windows, для сборки и размещения контейнеров вам нужно запустить виртуальную машину Linux на коробке.

1.12.0

Начиная с версии 1.12 вам не нужно устанавливать отдельную виртуальную машину, поскольку Docker может использовать встроенную функциональность Hyper-V для Windows, чтобы запустить небольшую машину Linux, чтобы действовать как бэкэнд.

Чтобы установить докер, выполните следующие действия:

1. Перейти к Docker для Windows
2. Загрузите и запустите программу установки.
3. Продолжайте установку с помощью параметров по умолчанию и введите учетные данные своей учетной записи по запросу.

Проверьте здесь для получения дополнительной информации по установке.

1.11.2

До версии 1.11 лучшим способом запуска этой виртуальной машины Linux является установка Docker Toolbox, которая устанавливает Docker, VirtualBox и гостевую машину Linux.

Чтобы установить панель инструментов докеров, выполните следующие действия:

1. Перейти к панели инструментов Docker
2. Нажмите ссылку для Windows и запустите программу установки.
3. Продолжайте установку с помощью параметров по умолчанию и введите учетные данные своей учетной записи по запросу.

Это установит двоичные файлы Docker в Program Files и обновит любую существующую установку Virtual Box. Проверьте здесь для получения дополнительной информации по установке.

Для проверки установки:

1.12.0

1. Запустите `Docker` из меню «Пуск», если он еще не запущен, и убедитесь, что он запущен. Затем поднимите любой терминал (либо `cmd` либо PowerShell)

1.11.2

1. На рабочем столе найдите значок панели инструментов Docker. Щелкните значок, чтобы запустить терминал Docker Toolbox.
2. Как только терминал открыт,

   ```
   docker run hello-world
   ```

3. Если все хорошо, тогда это должно напечатать приветственное сообщение, подтверждающее успешную установку.

Установка докеров на Ubuntu Linux

Docker поддерживается в следующих *64-битных* версиях Ubuntu Linux:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Несколько примечаний:

> Следующие инструкции включают установку только с использованием пакетов **Docker** , что обеспечивает получение последней официальной версии **Docker** . Если вам нужно установить только пакеты, `Ubuntu-managed` , ознакомьтесь с документацией Ubuntu (по очевидным причинам не рекомендуется по-другому).
>
> Ubuntu Utopic 14.10 и 15.04 существуют в репозитории APT Docker, но уже не поддерживаются официально из-за известных проблем безопасности.

Предпосылки

- Docker работает только на 64-битной установке Linux.
- Для Docker требуется ядро Linux версии 3.10 или новее (кроме `Ubuntu Precise 12.04` , для которого требуется версия 3.13 или выше). Ядрам старше 3.10 не хватает

некоторых функций, необходимых для запуска контейнеров Docker и содержат известные ошибки, которые вызывают потерю данных и часто паникуют при определенных условиях. Проверьте текущую версию ядра с помощью команды `uname -r` . Проверьте этот пост, если вам нужно обновить ядро `Ubuntu Precise (12.04 LTS)` , прокручивая его дальше. Обратитесь к этому сообщению WikiHow, чтобы получить последнюю версию для других установок Ubuntu.

Обновление источников APT

Это необходимо сделать для доступа к пакетам из репозитория Docker.

1. Войдите в свой компьютер как пользователь с привилегиями `sudo` или `root` .
2. Откройте окно терминала.
3. Обновите информацию о пакете, убедитесь, что APT работает с https-методом и установлены сертификаты CA.

```
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

4. Добавить официальный ключ GPG Docker:

   ```
   $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
   ```

   Убедитесь, что ключевой отпечаток **9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88** .

   ```
   $ sudo apt-key fingerprint 0EBFCD88
   ```

```
      pub   4096R/0EBFCD88 2017-02-22
            Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
      uid                  Docker Release (CE deb) <docker@docker.com>
      sub   4096R/F273FCD8 2017-02-22
```

5. Найдите запись в таблице ниже, которая соответствует вашей версии Ubuntu. Это определяет, где APT будет искать пакеты Docker. Когда это возможно, запустите долгосрочную версию (LTS) Ubuntu.

Версия Ubuntu	вместилище
Точный 12,04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-precise main`
Trusty 14.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-trusty main`
Уили 15.10	`deb https://apt.dockerproject.org/repo ubuntu-wily main`

Версия Ubuntu	вместилище
Xenial 16.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-xenial main`

Примечание. Docker не предоставляет пакеты для всех архитектур. Бинарные артефакты создаются ночью, и вы можете скачать их с `https://master.dockerproject.org` . Чтобы установить докеры в многоадресной системе, добавьте в запись `[arch=...]` . Подробнее см. В Debian Multiarch wiki .

6. Выполните следующую команду, заменив запись для вашей операционной системы на placeholder `<REPO>` .

   $ echo "" | sudo tee /etc/apt/sources.list.d/docker.list

7. Обновите индекс пакета `APT` , выполнив `sudo apt-get update` .

8. Убедитесь, что `APT` вытягивается из правого репозитория.

Когда вы запускаете следующую команду, возвращается запись для каждой версии Docker, которая доступна для вас. Каждая запись должна иметь URL `https://apt.dockerproject.org/repo/` . Установленная в настоящее время версия помечена знаком `***` См. Вывод нижеприведенного примера.

```
$ apt-cache policy docker-engine

  docker-engine:
    Installed: 1.12.2-0~trusty
    Candidate: 1.12.2-0~trusty
    Version table:
   *** 1.12.2-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
          100 /var/lib/dpkg/status
       1.12.1-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
       1.12.0-0~trusty 0
          500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

Теперь, когда вы запускаете `apt-get upgrade` , `APT` вытаскивает из нового репозитория.

Предварительные требования по версии Ubuntu

Для Ubuntu Trusty (14.04), Wily (15.10) и Xenial (16.04) установите пакеты `linux-image-extra-*` kernel, которые позволяют использовать драйвер хранилища `aufs` .

Чтобы установить пакеты `linux-image-extra-*` :

1. Откройте терминал на хосте Ubuntu.

2. Обновите менеджер пакетов командой `sudo apt-get update` .

3. Установите рекомендуемые пакеты.

```
$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
```

4. Перейти к установке Docker

Для Ubuntu Precise (12.04 LTS) Docker требует версию ядра 3.13. Если ваша версия ядра старше 3.13, вы должны ее обновить. Обратитесь к этой таблице, чтобы узнать, какие пакеты необходимы для вашей среды:

пакет	Описание
`linux-image-generic-lts-trusty`	Общий образ ядра Linux. Это ядро имеет встроенный `AUFS` . Это необходимо для запуска Docker.
`linux-headers-generic-lts-trusty`	Позволяет создавать пакеты, такие как `VirtualBox guest additions ZFS` и `VirtualBox guest additions` которые зависят от них. Если вы не установили заголовки для вашего существующего ядра, вы можете пропустить эти заголовки для `trusty` ядра. Если вы не уверены, вы должны включить этот пакет для обеспечения безопасности.
`xserver-xorg-lts-trusty`	Необязательно в неграфических средах без Unity / Xorg. **Требуется** при запуске Docker на машине с графической средой.
`ligbl1-mesa-glx-lts-trusty`	Чтобы узнать больше о причинах этих пакетов, прочитайте инструкции по установке для backported ядер, в частности, LTS Enablement Stack . См. Примечание 5 под каждой версией.

Чтобы обновить ядро и установить дополнительные пакеты, выполните следующие действия:

1. Откройте терминал на хосте Ubuntu.
2. Обновите менеджер пакетов командой `sudo apt-get update` .
3. Установите как необходимые, так и дополнительные пакеты.

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

4. Повторите этот шаг для других пакетов, которые необходимо установить.
5. Перезагрузите хост, чтобы использовать обновленное ядро, используя команду `sudo reboot` .
6. После перезагрузки перейдите и установите Docker.

Установите последнюю версию

Убедитесь, что вы удовлетворяете необходимым требованиям, только затем следуйте приведенным ниже инструкциям.

> **Примечание.** Для производственных систем рекомендуется установить определенную версию, чтобы вы случайно не обновили Docker. Вы должны тщательно планировать модернизацию производственных систем.

1. Войдите в свою установку Ubuntu как пользователь с привилегиями `sudo` . (Возможно, работает `sudo -su`).
2. Обновите индекс пакета APT, выполнив `sudo apt-get update` .
3. Установите Docker Community Edition с помощью команды `sudo apt-get install docker-ce` .
4. Запустите демон `docker` с помощью команды `sudo service docker start` .
5. Убедитесь, что `docker` установлен правильно, запустив изображение hello-world.

```
$ sudo docker run hello-world
```

Эта команда загружает тестовое изображение и запускает его в контейнере. Когда контейнер запускается, он печатает информационное сообщение и завершает работу.

Управление Docker как пользователем без полномочий root

Если вы не хотите использовать `sudo` при использовании команды docker, создайте группу Unix под названием `docker` и добавьте к ней пользователей. Когда демон `docker` запускается, он становится владельцем сокета Unix, который читается / записывается группой докеров.

Чтобы создать группу `docker` и добавить пользователя:

1. Войдите в Ubuntu как пользователь с привилегиями `sudo` .
2. Создайте `docker` группу с помощью команды `sudo groupadd docker` .
3. Добавьте пользователя в группу `docker` .

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

4. Выйдите из системы и войдите в систему, чтобы ваше членство в группе было переоценено.
5. Убедитесь, что вы можете выполнять команды `docker` без разрешения `sudo` .

```
$ docker run hello-world
```

Если это не удается, вы увидите сообщение об ошибке:

```
 Cannot connect to the Docker daemon. Is 'docker daemon' running on this host?
```

Проверьте, установлена ли `DOCKER_HOST` среды `DOCKER_HOST` для вашей оболочки.

```
$ env | grep DOCKER_HOST
```

Если он установлен, указанная выше команда вернет результат. Если это так, отключите его.

```
$ unset DOCKER_HOST
```

Возможно, вам придется отредактировать свою среду в файлах, таких как `~/.bashrc` или `~/.profile` чтобы предотвратить `DOCKER_HOST` переменной `DOCKER_HOST` .

Установка Docker на Ubuntu

Требования: Docker может быть установлен на любом Linux с ядром, по крайней мере, версии 3.10. Docker поддерживается в следующих 64-битных версиях Ubuntu Linux:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Простая установка

Примечание. Установка Docker из репозитория Ubuntu по умолчанию будет устанавливать старую версию Docker.

Чтобы установить последнюю версию Docker с помощью репозитория Docker, используйте `curl` для захвата и запуска скрипта установки, предоставленного Docker:

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

Кроме того, `wget` можно использовать для установки Docker:

```
$ wget -qO- https://get.docker.com/ | sh
```

Теперь будет установлен Docker.

Ручная установка

Если, однако, запуск сценария установки не является вариантом, следующие инструкции могут быть использованы для ручной установки последней версии Docker из официального

репозитория.

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

Добавьте ключ GPG:

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
  --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

Затем откройте файл `/etc/apt/sources.list.d/docker.list` в вашем любимом редакторе. Если файл не существует, создайте его. Удалите все существующие записи. Затем, в зависимости от вашей версии, добавьте следующую строку:

- Ubuntu Precise 12.04 (LTS):

  ```
  deb https://apt.dockerproject.org/repo ubuntu-precise main
  ```

- Ubuntu Trusty 14.04 (LTS)

  ```
  deb https://apt.dockerproject.org/repo ubuntu-trusty main
  ```

- Ubuntu Wily 15.10

  ```
  deb https://apt.dockerproject.org/repo ubuntu-wily main
  ```

- Ubuntu Xenial 16.04 (LTS)

  ```
  deb https://apt.dockerproject.org/repo ubuntu-xenial main
  ```

Сохраните файл и выйдите, а затем обновите свой индекс пакета, удалите все установленные версии Docker и убедитесь, что `apt` вытягивается из правильного репо:

```
$ sudo apt-get update
$ sudo apt-get purge lxc-docker
$ sudo apt-cache policy docker-engine
```

В зависимости от вашей версии Ubuntu могут потребоваться некоторые предварительные условия:

- Ubuntu Xenial 16.04 (LTS), Ubuntu Wily 15.10, Ubuntu Trusty 14.04 (LTS)

  ```
  sudo apt-get update && sudo apt-get install linux-image-extra-$(uname -r)
  ```

- Ubuntu Precise 12.04 (LTS)

  Для этой версии Ubuntu требуется версия ядра 3.13. Возможно, вам потребуется установить дополнительные пакеты в зависимости от вашей среды:

  ```
  linux-image-generic-lts-trusty
  ```

  Общий образ ядра Linux. Это ядро имеет встроенный AUFS. Это необходимо для

запуска Docker.

`linux-headers-generic-lts-trusty`

Позволяет создавать пакеты, такие как гостевые дополнения ZFS и VirtualBox, которые зависят от них. Если вы не установили заголовки для вашего существующего ядра, вы можете пропустить эти заголовки для `trusty` ядра. Если вы не уверены, вы должны включить этот пакет для обеспечения безопасности.

`xserver-xorg-lts-trusty`

`libgl1-mesa-glx-lts-trusty`

Эти два пакета являются необязательными в неграфических средах без Unity / Xorg. Требуется при запуске Docker на машине с графической средой.

Чтобы узнать больше о причинах этих пакетов, ознакомьтесь с инструкциями по установке для backported ядер, в частности, LTS Enablement Stack - см. Примечание 5 к каждой версии.

Установите необходимые пакеты и перезагрузите хост:

```
$ sudo apt-get install linux-image-generic-lts-trusty

$ sudo reboot
```

Наконец, обновите индекс `apt` package и установите Docker:

```
$ sudo apt-get update
$ sudo apt-get install docker-engine
```

Запуск демона:

```
$ sudo service docker start
```

Теперь убедитесь, что докер работает правильно, запустив тестовое изображение:

```
$ sudo docker run hello-world
```

Эта команда должна напечатать приветственное сообщение, подтверждающее успешную установку.

Создайте контейнер для докеров в облаке Google

Вы можете использовать докер, не используя демона докеров (движок), используя облачных провайдеров. В этом примере у вас должен быть `gcloud` (Google Cloud util), который подключен к вашей учетной записи

```
docker-machine create --driver google --google-project `your-project-name` google-machine-type
```

```
f1-large fm02
```

В этом примере будет создан новый экземпляр на консоли Google Cloud. Использование машинного времени `f1-large`

Установите Docker на Ubuntu

Docker поддерживается в следующих *64-битных* версиях Ubuntu Linux:

- Ubuntu Xenial 16.04 (LTS)
- Ubuntu Wily 15.10
- Ubuntu Trusty 14.04 (LTS)
- Ubuntu Precise 12.04 (LTS)

Несколько примечаний:

> Следующие инструкции включают установку только с использованием пакетов **Docker** , что обеспечивает получение последней официальной версии **Docker** . Если вам нужно установить только пакеты, `Ubuntu-managed` , ознакомьтесь с документацией Ubuntu (по очевидным причинам не рекомендуется по-другому).
>
> Ubuntu Utopic 14.10 и 15.04 существуют в репозитории APT Docker, но уже не поддерживаются официально из-за известных проблем безопасности.

Предпосылки

- Docker работает только на 64-битной установке Linux.
- Для Docker требуется ядро Linux версии 3.10 или новее (кроме `Ubuntu Precise 12.04` , для которого требуется версия 3.13 или выше). Ядрам старше 3.10 не хватает некоторых функций, необходимых для запуска контейнеров Docker и содержат известные ошибки, которые вызывают потерю данных и часто паникуют при определенных условиях. Проверьте текущую версию ядра с помощью команды `uname -r` . Проверьте этот пост, если вам нужно обновить ядро `Ubuntu Precise (12.04 LTS)` , прокручивая его дальше. Обратитесь к этому сообщению WikiHow, чтобы получить последнюю версию для других установок Ubuntu.

Обновление источников APT

Это необходимо сделать для доступа к пакетам из репозитория Docker.

1. Войдите в свой компьютер как пользователь с привилегиями `sudo` или `root` .
2. Откройте окно терминала.
3. Обновите информацию о пакете, убедитесь, что APT работает с https-методом и установлены сертификаты CA.

```
$ sudo apt-get update
```

```
$ sudo apt-get install apt-transport-https ca-certificates
```

4. Добавьте новый ключ `GPG` . Эти команды загружают ключ с ID `58118E89F3A912897C070ADBF76221572C52609D` с `58118E89F3A912897C070ADBF76221572C52609D` ключей `hkp://ha.pool.sks-keyservers.net:80` и добавляет его в `adv keychain` . Для получения дополнительной информации см. Вывод `man apt-key` .

   ```
   $ sudo apt-key adv \
          --keyserver hkp://ha.pool.sks-keyservers.net:80 \
          --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
   ```

5. Найдите запись в таблице ниже, которая соответствует вашей версии Ubuntu. Это определяет, где APT будет искать пакеты Docker. Когда это возможно, запустите долгосрочную версию (LTS) Ubuntu.

Версия Ubuntu	вместилище
Точный 12,04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-precise main`
Trusty 14.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-trusty main`
Уили 15.10	`deb https://apt.dockerproject.org/repo ubuntu-wily main`
Xenial 16.04 (LTS)	`deb https://apt.dockerproject.org/repo ubuntu-xenial main`

   Примечание. Docker не предоставляет пакеты для всех архитектур. Бинарные артефакты создаются ночью, и вы можете скачать их с `https://master.dockerproject.org` . Чтобы установить докеры в многоадресной системе, добавьте в запись `[arch=...]` . Подробнее см. В Debian Multiarch wiki .

6. Выполните следующую команду, заменив запись для вашей операционной системы на placeholder `<REPO>` .

   $ echo "" | sudo tee /etc/apt/sources.list.d/docker.list

7. Обновите индекс пакета `APT` , выполнив `sudo apt-get update` .

8. Убедитесь, что `APT` вытягивается из правого репозитория.

Когда вы запускаете следующую команду, возвращается запись для каждой версии Docker, которая доступна для вас. Каждая запись должна иметь URL `https://apt.dockerproject.org/repo/` . Установленная в настоящее время версия помечена знаком `***` См. Вывод нижеприведенного примера.

```
$ apt-cache policy docker-engine

  docker-engine:
```

```
   Installed: 1.12.2-0~trusty
   Candidate: 1.12.2-0~trusty
   Version table:
  *** 1.12.2-0~trusty 0
         500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
         100 /var/lib/dpkg/status
      1.12.1-0~trusty 0
         500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
      1.12.0-0~trusty 0
         500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

Теперь, когда вы запускаете `apt-get upgrade` , `APT` вытаскивает из нового репозитория.

Предварительные требования по версии Ubuntu

Для Ubuntu Trusty (14.04), Wily (15.10) и Xenial (16.04) установите пакеты `linux-image-extra-*` kernel, которые позволяют использовать драйвер хранилища `aufs` .

Чтобы установить пакеты `linux-image-extra-*` :

1. Откройте терминал на хосте Ubuntu.
2. Обновите менеджер пакетов командой `sudo apt-get update` .
3. Установите рекомендуемые пакеты.

```
$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
```

4. Перейти к установке Docker

Для Ubuntu Precise (12.04 LTS) Docker требует версию ядра 3.13. Если ваша версия ядра старше 3.13, вы должны ее обновить. Обратитесь к этой таблице, чтобы узнать, какие пакеты необходимы для вашей среды:

пакет	Описание
`linux-image-generic-lts-trusty`	Общий образ ядра Linux. Это ядро имеет встроенный `AUFS` . Это необходимо для запуска Docker.
`linux-headers-generic-lts-trusty`	Позволяет создавать пакеты, такие как `VirtualBox guest additions ZFS` и `VirtualBox guest additions` которые зависят от них. Если вы не установили заголовки для вашего существующего ядра, вы можете пропустить эти заголовки для `trusty` ядра. Если вы не уверены, вы должны включить этот пакет для обеспечения безопасности.
`xserver-xorg-lts-trusty`	Необязательно в неграфических средах без Unity / Xorg. **Требуется** при запуске Docker на машине с графической средой.
`libgl1-mesa-glx-lts-`	Чтобы узнать больше о причинах этих пакетов, прочитайте инструкции

пакет	Описание
`trusty`	по установке для backported ядер, в частности, LTS Enablement Stack . См. Примечание 5 под каждой версией.

Чтобы обновить ядро и установить дополнительные пакеты, выполните следующие действия:

1. Откройте терминал на хосте Ubuntu.
2. Обновите менеджер пакетов командой `sudo apt-get update` .
3. Установите как необходимые, так и дополнительные пакеты.

```
$ sudo apt-get install linux-image-generic-lts-trusty
```

4. Повторите этот шаг для других пакетов, которые необходимо установить.
5. Перезагрузите хост, чтобы использовать обновленное ядро, используя команду `sudo reboot` .
6. После перезагрузки перейдите и установите Docker.

Установите последнюю версию

Убедитесь, что вы удовлетворяете необходимым требованиям, только затем следуйте приведенным ниже инструкциям.

> **Примечание.** Для производственных систем рекомендуется установить определенную версию, чтобы вы случайно не обновили Docker. Вы должны тщательно планировать модернизацию производственных систем.

1. Войдите в свою установку Ubuntu как пользователь с привилегиями `sudo` . (Возможно, работает `sudo -su`).
2. Обновите индекс пакета APT, выполнив `sudo apt-get update` .
3. Установите Docker с помощью команды `sudo apt-get install docker-engine` .
4. Запустите демон `docker` с помощью команды `sudo service docker start` .
5. Убедитесь, что `docker` установлен правильно, запустив изображение hello-world.

```
$ sudo docker run hello-world
```

Эта команда загружает тестовое изображение и запускает его в контейнере. Когда контейнер запускается, он печатает информационное сообщение и завершает работу.

Управление Docker как пользователем без полномочий root

Если вы не хотите использовать `sudo` при использовании команды docker, создайте группу Unix под названием `docker` и добавьте к ней пользователей. Когда демон `docker` запускается, он становится владельцем сокета Unix, который читается / записывается группой докеров.

Чтобы создать группу `docker` и добавить пользователя:

1. Войдите в Ubuntu как пользователь с привилегиями `sudo` .
2. Создайте `docker` группу с помощью команды `sudo groupadd docker` .
3. Добавьте пользователя в группу `docker` .

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

4. Выйдите из системы и войдите в систему, чтобы ваше членство в группе было переоценено.
5. Убедитесь, что вы можете выполнять команды `docker` без разрешения `sudo` .

```
$ docker run hello-world
```

Если это не удается, вы увидите сообщение об ошибке:

```
Cannot connect to the Docker daemon. Is 'docker daemon' running on this host?
```

Проверьте, установлена ли `DOCKER_HOST` среды `DOCKER_HOST` для вашей оболочки.

```
$ env | grep DOCKER_HOST
```

Если он установлен, указанная выше команда вернет результат. Если это так, отключите его.

```
$ unset DOCKER_HOST
```

Возможно, вам придется отредактировать свою среду в файлах, таких как `~/.bashrc` или `~/.profile` чтобы предотвратить `DOCKER_HOST` переменной `DOCKER_HOST` .

Установка Docker-ce OR Docker-ee на CentOS

Докер объявил следующие выпуски:

-Docker-ee (Enterprise Edition) вместе с Docker-ce (Community Edition) и Docker (коммерческая поддержка)

Этот документ поможет вам с этапами установки выпуска Docker-ee и Docker-ce в CentOS

Установка Docker-ce

Ниже приведены шаги по установке версии docker-ce

1. Установите yum-utils, который предоставляет утилиту yum-config-manager:

```
$ sudo yum install -y yum-utils
```

2. Используйте следующую команду для настройки стабильного репозитория:

```
$ sudo yum-config-manager \
 --add-repo \
 https://download.docker.com/linux/centos/docker-ce.repo
```

3. Необязательно: включить репозиторий. Этот репозиторий включен в файл docker.repo выше, но по умолчанию отключен. Вы можете включить его вместе с стабильным хранилищем.

```
 $ sudo yum-config-manager --enable docker-ce-edge
```

- Вы можете отключить пограничный репозиторий, запустив команду `yum-config-manager` с флагом `--disable` . Чтобы снова включить его, используйте флаг `--enable` . Следующая команда отключает репозиторий.

```
 $ sudo yum-config-manager --disable docker-ce-edge
```

4. Обновите индекс пакета yum.

```
 $ sudo yum makecache fast
```

5. Установите docker-ce, используя следующую команду:

```
 $ sudo yum install docker-ce-17.03.0.ce
```

6. Подтвердите отпечаток Docker-ce

```
060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35
```

Если вы хотите установить другую версию docker-ce, вы можете использовать следующую команду:

```
$ sudo yum install docker-ce-VERSION
```

Укажите номер `VERSION`

7. Если все пойдет хорошо, docker-ce теперь установлен в вашей системе, используйте следующую команду для запуска:

```
$ sudo systemctl start docker
```

8. Проверьте установку докеров:

```
$ sudo docker run hello-world
```

вы должны получить следующее сообщение:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

-Docker-ee (Enterprise Edition)

Для Enterprise Edition (EE) потребуется зарегистрироваться, чтобы получить <DOCKER-EE-URL>.

1. Чтобы зарегистрироваться, перейдите на страницу https://cloud.docker.com/ . Введите свои данные и подтвердите свой идентификатор электронной почты. После подтверждения вам будет предоставлен <DOCKER-EE-URL>, который вы можете увидеть в своей панели после нажатия на настройку.

2. Удалите все существующие репозитории Docker из `/etc/yum.repos.d/`

3. Сохраните URL-адрес репозитория Docker EE в переменной yum в `/etc/yum/vars/` . Замените <DOCKER-EE-URL> URL-адресом, указанным на первом шаге.

```
$ sudo sh -c 'echo "<DOCKER-EE-URL>" > /etc/yum/vars/dockerurl'
```

4. Установите yum-utils, который предоставляет утилиту yum-config-manager:

```
$ sudo yum install -y yum-utils
```

5. Используйте следующую команду, чтобы добавить стабильный репозиторий:

```
$ sudo yum-config-manager \
--add-repo \
<DOCKER-EE-URL>/docker-ee.repo
```

6. Обновите индекс пакета yum.

```
$ sudo yum makecache fast
```

7. Установить docker-ee

```
sudo yum install docker-ee
```

8. Вы можете запустить docker-ee, используя следующую команду:

```
$ sudo systemctl start docker
```

Прочитайте Начало работы с Docker онлайн: https://riptutorial.com/ru/docker/topic/658/начало-работы-с-docker

глава 2: API-интерфейс Docker Engine

Вступление

API, который позволяет вам контролировать все аспекты Docker из ваших собственных приложений, создавать инструменты для управления и мониторинга приложений, работающих на Docker, и даже использовать его для создания приложений на самом Docker.

Examples

Включить удаленный доступ к Docker API в Linux

Измените `DOCKER_OPTS /etc/init/docker.conf` и обновите переменную `DOCKER_OPTS` следующим образом:

```
DOCKER_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'
```

Перезапустить Docker deamon

```
service docker restart
```

Проверьте, работает ли Remote API

```
curl -X GET http://localhost:4243/images/json
```

Включить удаленный доступ к API-интерфейсу Docker на Linux, работающем systemd

Linux, работающий systemd, как Ubuntu 16.04, добавление `-H tcp://0.0.0.0:2375` в `/etc/default/docker` не имеет никакого эффекта, к которому он привык.

Вместо этого создайте файл с именем `/etc/systemd/system/docker-tcp.socket` чтобы сделать докеры доступными для TCP-сокета на порту 4243:

```
[Unit]
Description=Docker Socket for the API
[Socket]
ListenStream=4243
Service=docker.service
[Install]
WantedBy=sockets.target
```

Затем включите новый сокет:

```
systemctl enable docker-tcp.socket
systemctl enable docker.socket
systemctl stop docker
systemctl start docker-tcp.socket
systemctl start docker
```

Теперь проверьте, работает ли Remote API:

```
curl -X GET http://localhost:4243/images/json
```

Включить удаленный доступ с помощью TLS на Systemd

Скопируйте файл блока установщика пакета в / etc, где изменения не будут перезаписаны при обновлении:

```
cp /lib/systemd/system/docker.service /etc/systemd/system/docker.service
```

Обновите /etc/systemd/system/docker.service с вашими параметрами в ExecStart:

```
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376 \
  --tlsverify --tlscacert=/etc/docker/certs/ca.pem \
  --tlskey=/etc/docker/certs/key.pem \
  --tlscert=/etc/docker/certs/cert.pem
```

Обратите внимание, что `dockerd` - это имя демона 1.12, прежде чем он был `docker daemon` `dockerd` . Также обратите внимание, что 2376 - стандартный стандартный порт TLS, 2375 - стандартный незашифрованный порт. См. Эту страницу о шагах по созданию собственного собственного сертификата CA, сертификата и ключа TLS.

После внесения изменений в файлы unitd, запустите следующую команду, чтобы перезагрузить конфигурацию systemd:

```
systemctl daemon-reload
```

Затем запустите следующий перезапуск докера:

```
systemctl restart docker
```

Плохая идея пропустить шифрование TLS при экспорте порта Docker, поскольку любой, у кого есть сетевой доступ к этому порту, имеет полный доступ root на хост.

Потяжка изображения с индикаторами выполнения, написанная на Go

Вот пример вытягивания изображения с использованием `Docker Engine API Go` и `Docker Engine API` и тех же индикаторов выполнения, что и показанные при запуске `docker pull your_image_name` в `CLI` . Для целей индикаторов выполнения используются некоторые коды

ANSI .

```
package yourpackage

import (
    "context"
    "encoding/json"
    "fmt"
    "io"
    "strings"

    "github.com/docker/docker/api/types"
    "github.com/docker/docker/client"
)

// Struct representing events returned from image pulling
type pullEvent struct {
    ID             string `json:"id"`
    Status         string `json:"status"`
    Error          string `json:"error,omitempty"`
    Progress       string `json:"progress,omitempty"`
    ProgressDetail struct {
        Current int `json:"current"`
        Total   int `json:"total"`
    } `json:"progressDetail"`
}

// Actual image pulling function
func PullImage(dockerImageName string) bool {
    client, err := client.NewEnvClient()

    if err != nil {
        panic(err)
    }

    resp, err := client.ImagePull(context.Background(), dockerImageName,
types.ImagePullOptions{})

    if err != nil {
        panic(err)
    }

    cursor := Cursor{}
    layers := make([]string, 0)
    oldIndex := len(layers)

    var event *pullEvent
    decoder := json.NewDecoder(resp)

    fmt.Printf("\n")
    cursor.hide()

    for {
        if err := decoder.Decode(&event); err != nil {
            if err == io.EOF {
                break
            }

            panic(err)
        }
```

```
        imageID := event.ID

        // Check if the line is one of the final two ones
        if strings.HasPrefix(event.Status, "Digest:") || strings.HasPrefix(event.Status,
"Status:") {
            fmt.Printf("%s\n", event.Status)
            continue
        }

        // Check if ID has already passed once
        index := 0
        for i, v := range layers {
            if v == imageID {
                index = i + 1
                break
            }
        }

        // Move the cursor
        if index > 0 {
            diff := index - oldIndex

            if diff > 1 {
                down := diff - 1
                cursor.moveDown(down)
            } else if diff < 1 {
                up := diff*(-1) + 1
                cursor.moveUp(up)
            }

            oldIndex = index
        } else {
            layers = append(layers, event.ID)
            diff := len(layers) - oldIndex

            if diff > 1 {
                cursor.moveDown(diff) // Return to the last row
            }

            oldIndex = len(layers)
        }

        cursor.clearLine()

        if event.Status == "Pull complete" {
            fmt.Printf("%s: %s\n", event.ID, event.Status)
        } else {
            fmt.Printf("%s: %s %s\n", event.ID, event.Status, event.Progress)
        }

    }

    cursor.show()

    if strings.Contains(event.Status, fmt.Sprintf("Downloaded newer image for %s",
dockerImageName)) {
        return true
    }

    return false
}
```

Для лучшей читаемости действия курсора с кодами ANSI перемещаются в отдельную структуру, которая выглядит следующим образом:

```
package yourpackage

import "fmt"

// Cursor structure that implements some methods
// for manipulating command line's cursor
type Cursor struct{}

func (cursor *Cursor) hide() {
    fmt.Printf("\033[?25l")
}

func (cursor *Cursor) show() {
    fmt.Printf("\033[?25h")
}

func (cursor *Cursor) moveUp(rows int) {
    fmt.Printf("\033[%dF", rows)
}

func (cursor *Cursor) moveDown(rows int) {
    fmt.Printf("\033[%dE", rows)
}

func (cursor *Cursor) clearLine() {
    fmt.Printf("\033[2K")
}
```

После этого в вашем основном пакете вы можете вызвать функцию `PullImage` передающую имя изображения, которое вы хотите вытащить. Конечно, перед тем, как позвонить, вы должны войти в реестр Docker, где находится изображение.

Выполнение запроса cURL с передачей некоторой сложной структуры

При использовании `cURL` для некоторых запросов к `Docker API` может быть немного сложно передать некоторые сложные структуры. Скажем, получение списка изображений позволяет использовать фильтры в качестве параметра запроса, которые должны быть `JSON` представлением `map[string][]string` (о картах в `Go` вы можете найти здесь).
Вот как это сделать:

```
curl --unix-socket /var/run/docker.sock \
    -XGET "http:/v1.29/images/json" \
    -G \
    --data-urlencode 'filters={"reference":{"yourpreciousregistry.com/path/to/image": true},
"dangling":{"true": true}}'
```

Здесь флаг `-G` используется для указания того, что данные в параметре `--data-urlencode` будут использоваться в запросе `HTTP GET` вместо запроса `POST` который в противном случае использовался бы. Данные будут добавлены к URL-адресу с помощью `?` разделитель.

Прочитайте API-интерфейс Docker Engine онлайн:
https://riptutorial.com/ru/docker/topic/3935/api-интерфейс-docker-engine

глава 3: docker проверяет получение различных полей для ключа: значение и элементы списка

Examples

различные примеры проверки докеров

Я нахожу, что примеры в `docker inspect` документацию, кажутся волшебными, но не объясняют многое.

Проверка докеров важна, потому что это чистый способ извлечения информации из `docker inspect -f ... container_id` контейнера `docker inspect -f ... container_id`

(или весь запущенный контейнер)

```
docker inspect -f ... $(docker ps -q)
```

избегая некоторых ненадежных

```
docker command | grep or awk | tr or cut
```

Когда вы запускаете `docker inspect` вы можете легко получить значения с «верхнего уровня», используя базовый синтаксис для контейнера, использующего htop (от https://hub.docker.com/r/jess/htop/) с pid ae1

```
docker inspect -f '{{.Created}}' ae1
```

может показать

```
2016-07-14T17:44:14.159094456Z
```

или же

```
docker inspect -f '{{.Path}}' ae1
```

может показать

```
htop
```

Теперь, если я извлечу часть моего `docker inspect`

я вижу

```
"State": { "Status": "running", "Running": true, "Paused": false, "Restarting": false,
"OOMKilled": false, "Dead": false, "Pid": 4525, "ExitCode": 0, "Error": "", "StartedAt": "2016-
07-14T17:44:14.406286293Z", "FinishedAt": "0001-01-01T00:00:00Z"
```

Таким образом, я получаю

словарь, поскольку он имеет `{ ...}` и много ключей: значения

Таким образом, команда

```
docker inspect -f '{{.State}}' ae1
```

вернет список, например

```
{running true false false false false 4525 0 2016-07-14T17:44:14.406286293Z 0001-01-
01T00:00:00Z}
```

Я могу получить значение State.Pid легко

```
docker inspect -f '{{ .State.Pid }}' ae1
```

я получил

```
4525
```

Иногда проверка докеров дает список, начиная с `[` и заканчивается `]`

другой пример, с другим контейнером

```
docker inspect -f '{{ .Config.Env }}' 7a7
```

дает

```
[DISPLAY=:0 PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin LANG=fr_FR.UTF-8
LANGUAGE=fr_FR:en LC_ALL=fr_FR.UTF-8 DEBIAN_FRONTEND=noninteractive HOME=/home/gg WINEARCH=win32
WINEPREFIX=/home/gg/.wine_captvty]
```

Чтобы получить первый элемент списка, добавим индекс перед обязательным полем и 0 (как первый элемент) после, так что

```
docker inspect -f '{{ index ( .Config.Env) 0 }}' 7a7
```

дает

```
DISPLAY=:0
```

Мы получаем следующий элемент с 1 вместо 0, используя тот же синтаксис

```
docker inspect -f '{{ index ( .Config.Env) 1 }}' 7a7
```

дает

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Мы можем получить количество элементов этого списка

```
docker inspect -f '{{ len .Config.Env }}' 7a7
```

дает

```
9
```

и мы можем получить последний элемент списка, синтаксис непросто

```
docker inspect -f "{{ index .Config.Cmd $[$(docker inspect -format '{{ len .Config.Cmd }}'
$CID)-1]}}" 7a7
```

Прочитайте docker проверяет получение различных полей для ключа: значение и элементы списка онлайн: https://riptutorial.com/ru/docker/topic/6470/docker-проверяет-получение-различных-полей-для-ключа--значение-и-элементы-списка

глава 4: Docker фиксирует все запущенные контейнеры

Examples

Docker фиксирует все запущенные контейнеры

```
sudo docker stats $(sudo docker inspect -f "{{ .Name }}" $(sudo docker ps -q))
```

Показывает активное использование ЦП всех запущенных контейнеров.

Прочитайте Docker фиксирует все запущенные контейнеры онлайн:
https://riptutorial.com/ru/docker/topic/5863/docker-фиксирует-все-запущенные-контейнеры

глава 5: Dockerfiles

Вступление

Dockerfiles - это файлы, используемые для программной сборки изображений Docker. Они позволяют быстро и воспроизводить изображение Docker, и поэтому они полезны для совместной работы. Dockerfiles содержит инструкции по созданию изображения Docker. Каждая команда записывается в одну строку и указывается в форме `<INSTRUCTION><argument(s)>` . Dockerfiles используется для построения Docker изображений с помощью `docker build` команды.

замечания

Докер-файлы имеют форму:

```
# This is a comment
INSTRUCTION arguments
```

- Комментарии начинаются с `#`
- Инструкции только в верхнем регистре
- Первая инструкция файла Docker должна быть `FROM` чтобы указать базовое изображение

При создании файла Docker клиент Docker отправляет «сценарий сборки» демону Docker. Контекст сборки включает все файлы и папку в том же каталоге, что и файл Docker. Операции `COPY` и `ADD` могут использовать только файлы из этого контекста.

Некоторые файлы Docker могут начинаться с:

```
# escape=`
```

Это используется для указания парсеру Docker использовать `` ` `` как escape-символ вместо `\` . Это в основном полезно для файлов Windows Docker.

Examples

HelloWorld Dockerfile

Минимальный файл Dockerfile выглядит так:

```
FROM alpine
```

```
CMD ["echo", "Hello StackOverflow!"]
```

Это даст указание Docker создать изображение на основе Alpine (`FROM`), минимальное распределение для контейнеров и запустить определенную команду (`CMD`) при выполнении полученного изображения.

Создайте и запустите:

```
docker build -t hello .
docker run --rm hello
```

Это приведет к выводу:

```
Hello StackOverflow!
```

Копирование файлов

Чтобы скопировать файлы из контекста сборки в образ Docker, используйте инструкцию `COPY` :

```
COPY localfile.txt containerfile.txt
```

Если имя файла содержит пробелы, используйте альтернативный синтаксис:

```
COPY ["local file", "container file"]
```

Команда `COPY` поддерживает подстановочные знаки. Его можно использовать, например, для копирования всех изображений в каталог `images/` :

```
COPY *.jpg images/
```

Примечание: в этом примере `images/` могут отсутствовать. В этом случае Docker создаст его автоматически.

Показ порта

Чтобы объявить открытые порты из файла Dockerfile, используйте инструкцию `EXPOSE` :

```
EXPOSE 8080 8082
```

Параметры открытых портов можно переопределить из командной строки Docker, но это хорошая практика, чтобы явно установить их в файле Docker, поскольку он помогает понять, что делает приложение.

Лучшие докеры

Групповые общие операции

Docker создает изображения как набор слоев. Каждый уровень может добавлять только данные, даже если эти данные говорят о том, что файл был удален. Каждая инструкция создает новый слой. Например:

```
RUN apt-get -qq update
RUN apt-get -qq install some-package
```

Имеет пару недостатков:

- Он создаст два слоя, создавая более крупное изображение.
- Использование `apt-get update` только в заявлении `RUN` вызывает проблемы с кешированием, и впоследствии инструкции `apt-get install` могут **завершиться неудачно** . Предположим, что вы позже модифицируете `apt-get install` путем добавления дополнительных пакетов, затем докер интерпретирует начальные и измененные инструкции как идентичные и повторно использует кеш из предыдущих шагов. В результате команда `apt-get update` **не** выполняется, поскольку ее кешированная версия используется во время сборки.

Вместо этого используйте:

```
RUN apt-get -qq update && \
    apt-get -qq install some-package
```

так как это создает только один слой.

Упомяните сопровождающего

Обычно это вторая строка файла Docker. Он рассказывает, кто несет ответственность и сможет помочь.

```
LABEL maintainer John Doe <john.doe@example.com>
```

Если вы пропустите его, он не сломает ваше изображение. Но это тоже не поможет вашим пользователям.

Будьте краткими

Держите файл Dockerfile коротким. Если необходима сложная настройка, попробуйте использовать выделенный сценарий или настроить базовые изображения.

Инструкция пользователя

```
USER daemon
```

Инструкция `USER` устанавливает имя пользователя или UID для использования при запуске изображения и любых инструкций `RUN` , `CMD` и `ENTRYPOINT` которые следуют за ним в `Dockerfile` .

Инструкция WORKDIR

```
WORKDIR /path/to/workdir
```

Инструкция `WORKDIR` устанавливает рабочий каталог для любых инструкций `RUN` , `CMD` , `ENTRYPOINT` , `COPY` и `ADD` которые следуют за ним в файле Docker. Если `WORKDIR` не существует, он будет создан, даже если он не используется в любой последующей инструкции `Dockerfile` .

Его можно использовать несколько раз в одном `Dockerfile` . Если предоставлен относительный путь, он будет относиться к пути предыдущей инструкции `WORKDIR` . Например:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

Результатом окончательной команды `pwd` в этом `Dockerfile` будет `/a/b/c` .

Инструкция `WORKDIR` может разрешать переменные среды, предварительно установленные с помощью `ENV` . Вы можете использовать только переменные среды, явно установленные в `Dockerfile` . Например:

```
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

Результатом окончательной команды `pwd` в этом файле Docker будет `/path/$DIRNAME`

Инструкция VOLUME

```
VOLUME ["/data"]
```

Инструкция `VOLUME` создает точку монтирования с указанным именем и отмечает, что она содержит внешние тома с локального хоста или других контейнеров. Значение может быть массивом JSON, `VOLUME ["/var/log/"]` или простой строкой с несколькими аргументами, такими как `VOLUME /var/log` или `VOLUME /var/log /var/db` . Для получения дополнительной информации / примеров и инструкций по установке через клиент Docker см. Документацию об общих каталогах через тома.

Команда `docker run` инициализирует вновь созданный том любыми данными, которые существуют в указанном местоположении в базовом изображении. Например, рассмотрим

следующий фрагмент Dockerfile:

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

Этот файл Dockerfile приводит к созданию образа, который вызывает запуск докеров, для создания новой точки монтирования в / myvol и копирования файла приветствия во вновь созданный том.

Примечание. Если какие-либо шаги сборки изменяют данные в томе после того, как они были объявлены, эти изменения будут отброшены.

Примечание. Список анализируется как массив JSON, что означает, что вы должны использовать двойные кавычки (") вокруг слов, а не одиночных кавычек (').

Инструкция COPY

`COPY` имеет две формы:

```
COPY <src>... <dest>
COPY ["<src>",... "<dest>"] (this form is required for paths containing whitespace)
```

Инструкция `COPY` копирует новые файлы или каталоги из `<src>` и добавляет их в файловую систему контейнера по пути `<dest>` .

Можно указать несколько ресурсов `<src>` но они должны относиться к исходному каталогу, который строится (контекст сборки).

Каждый `<src>` может содержать подстановочные знаки, и сопоставление будет выполняться с помощью правил Go's `filepath.Match` . Например:

```
COPY hom* /mydir/        # adds all files starting with "hom"
COPY hom?.txt /mydir/    # ? is replaced with any single character, e.g., "home.txt"
```

`<dest>` - это абсолютный путь или путь относительно `WORKDIR` , в который будет скопирован источник в контейнере назначения.

```
COPY test relativeDir/   # adds "test" to `WORKDIR`/relativeDir/
COPY test /absoluteDir/  # adds "test" to /absoluteDir/
```

Все новые файлы и каталоги создаются с UID и GID из 0.

Примечание. Если вы создаете с помощью stdin (`docker build - < somefile`), контекст сборки не существует, поэтому `COPY` нельзя использовать.

`COPY` подчиняется следующим правилам:

- Путь `<src>` должен находиться внутри контекста сборки; вы не можете `COPY` ../something / something, потому что первым шагом сборки docker является отправка каталога контекста (и подкаталогов) демона докеров.
- Если `<src>` является каталогом, копируется все содержимое каталога, включая метаданные файловой системы. Примечание. Сама директория не копируется, а только ее содержимое.
- Если `<src>` - это любой другой тип файла, он копируется отдельно вместе с его метаданными. В этом случае, если `<dest>` заканчивается конечной косой чертой /, он будет считаться каталогом, а содержимое `<src>` будет записано в `<dest>/base(<src>)` .
- Если указано несколько ресурсов `<src>` , либо напрямую, либо из-за использования подстановочного знака, то `<dest>` должен быть каталогом, и он должен заканчиваться косой чертой `/` .
- Если `<dest>` не заканчивается конечной косой чертой, он будет считаться обычным файлом, а содержимое `<src>` будет записано в `<dest>` .
- Если `<dest>` не существует, он создается вместе со всеми отсутствующими каталогами на своем пути.

Инструкция ENV и ARG

ENV

```
ENV <key> <value>
ENV <key>=<value> ...
```

Команда `ENV` устанавливает переменную среды `<key>` в значение. Это значение будет находиться в среде всех команд «потомки» Dockerfile и также может быть заменено inline во многих.

Инструкция `ENV` имеет две формы. Первая форма, `ENV <key> <value>` , установит единственную переменную в значение. Вся строка после первого пространства будет рассматриваться как `<value>` - включая символы, такие как пробелы и кавычки.

Вторая форма, `ENV <key>=<value> ...` , позволяет одновременно устанавливать несколько переменных. Обратите внимание, что вторая форма использует знак равенства (=) в синтаксисе, а первая форма - нет. Подобно анализу командной строки, кавычки и обратные слэши могут использоваться для включения пробелов в значениях.

Например:

```
ENV myName="John Doe" myDog=Rex\ The\ Dog \
    myCat=fluffy
```

а также

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

даст те же чистые результаты в конечном контейнере, но первая форма предпочтительнее, поскольку она создает один уровень кеша.

Переменные среды, заданные с использованием `ENV` , сохраняются, когда контейнер запускается из полученного изображения. Вы можете просмотреть значения с помощью проверки `docker run --env <key>=<value>` и изменить их с помощью `docker run --env <key>=<value>` .

ARG

Если вы не хотите настаивать на настройке, вместо этого используйте `ARG` . `ARG` будет устанавливать среды только во время сборки. Например, установка

```
ENV DEBIAN_FRONTEND noninteractive
```

могут запутать пользователей `apt-get` на образ Debian, когда они вводят контейнер в интерактивном контексте через `docker exec -it the-container bash` .

Вместо этого используйте:

```
ARG DEBIAN_FRONTEND noninteractive
```

Вы можете альтернативно также установить значение для одной команды только с помощью:

```
RUN <key>=<value> <command>
```

ЭКСПОЗИЦИЯ

```
EXPOSE <port> [<port>...]
```

Команда `EXPOSE` информирует Docker о том, что контейнер прослушивает указанные сетевые порты во время выполнения. `EXPOSE` НЕ делает порты контейнера доступными для хоста. Для этого вы должны использовать флаг `-p` для публикации диапазона портов или флага `-P` для публикации всех открытых портов. Эти флаги используются в `docker run [OPTIONS] IMAGE [COMMAND][ARG...]` чтобы открыть порт для хоста. Вы можете открыть один

номер порта и опубликовать его извне под другим номером.

```
docker run -p 2500:80 <image name>
```

Эта команда создаст контейнер с именем <image> и привяжет порт контейнера 80 к порту 2500 хост-машины.

Чтобы настроить перенаправление портов в главной системе, см. Использование флага `-P` . Сетевая функция Docker поддерживает создание сетей без необходимости раскрывать порты в сети, подробную информацию см. В обзоре этой функции).

Инструкция LABEL

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

Команда `LABEL` добавляет метаданные к изображению. `LABEL` - пара ключ-значение. Чтобы включить пробелы в значение `LABEL` , используйте кавычки и обратную косую черту, как в случае синтаксического анализа командной строки. Несколько примеров использования:

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

Изображение может иметь более одной метки. Чтобы указать несколько меток, Docker рекомендует комбинировать метки в одной инструкции `LABEL` где это возможно. Каждая инструкция `LABEL` создает новый слой, который может привести к неэффективному изображению, если вы используете много меток. В этом примере создается один слой изображения.

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

Вышеуказанное также может быть записано как:

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```

Ярлыки являются добавочными, включая `LABEL` s в изображениях `FROM` . Если Docker обнаруживает метку / ключ, который уже существует, новое значение переопределяет любые предыдущие метки с идентичными ключами.

Чтобы просмотреть метки изображений, используйте команду проверки докеров.

```
"Labels": {
    "com.example.vendor": "ACME Incorporated"
```

```
    "com.example.label-with-value": "foo",
    "version": "1.0",
    "description": "This text illustrates that label-values can span multiple lines.",
    "multi.label1": "value1",
    "multi.label2": "value2",
    "other": "value3"
},
```

Инструкция CMD

Инструкция `CMD` имеет три формы:

```
CMD ["executable","param1","param2"] (exec form, this is the preferred form)
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
CMD command param1 param2 (shell form)
```

В `Dockerfile` может быть только одна команда `CMD` . Если вы перечислите несколько `CMD` тогда вступит в силу только последний `CMD` .

Основной целью `CMD` является предоставление значений по умолчанию для исполняющего контейнера. Эти значения по умолчанию могут включать исполняемый файл, или они могут опустить исполняемый файл, и в этом случае вы должны указать инструкцию `ENTRYPOINT` .

Примечание. Если `CMD` используется для предоставления аргументов по умолчанию для инструкции `ENTRYPOINT` инструкции `CMD` и `ENTRYPOINT` следует указывать в формате массива JSON.

Примечание. Форма exec анализируется как массив JSON, что означает, что вы должны использовать двойные кавычки («) вокруг слов, а не одиночных кавычек (').

Примечание. В отличие от формы оболочки форма exec не вызывает командную оболочку. Это означает, что нормальной обработки оболочки не происходит. Например, `CMD [ "echo", "$HOME" ]` не будет делать замену переменных в `$HOME` . Если вы хотите обработать оболочку, то либо используйте форму оболочки, либо выполните оболочку напрямую, например: `CMD [ "sh", "-c", "echo $HOME" ]` .

При использовании в форматах оболочки или exec команда `CMD` устанавливает команду, которая должна выполняться при запуске изображения.

Если вы используете форму оболочки `CMD` , тогда команда будет выполняться в `/bin/sh -c` :

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

Если вы хотите запустить команду без оболочки, вы должны выразить команду как массив JSON и предоставить полный путь к исполняемому файлу. Эта форма массива является предпочтительным форматом `CMD` . Любые дополнительные параметры должны быть индивидуально выражены как строки в массиве:

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```

Если вы хотите, чтобы ваш контейнер запускал один и тот же исполняемый файл каждый раз, вам следует рассмотреть возможность использования `ENTRYPOINT` в сочетании с `CMD` . См. `ENTRYPOINT` .

Если пользователь указывает аргументы для запуска docker, они будут переопределять значение по умолчанию, указанное в `CMD` .

Примечание: не путайте `RUN` с `CMD` . `RUN` фактически выполняет команду во время построения изображения и фиксирует результат; `CMD` ничего не выполняет во время сборки, но указывает намеченную команду для изображения.

Инструкция MAINTAINER

```
MAINTAINER <name>
```

Инструкция `MAINTAINER` позволяет вам установить поле Author созданных изображений.

НЕ ИСПОЛЬЗУЙТЕ ДИРЕКТИВУ ГЛАВНОГО ОБОРУДОВАНИЯ

Согласно официальной документации Docker, инструкция `MAINTAINER` устарела. Вместо этого следует использовать инструкцию `LABEL` для определения автора созданных изображений. Команда `LABEL` более гибкая, позволяет устанавливать метаданные и может быть легко просмотрена с помощью `docker inspect` команды.

```
LABEL maintainer="someone@something.com"
```

ОТ Инструкция

```
FROM <image>
```

Или же

```
FROM <image>:<tag>
```

Или же

```
FROM <image>@<digest>
```

Инструкция `FROM` устанавливает базовое изображение для последующих инструкций. Таким образом, действительный файл Dockerfile должен иметь `FROM` качестве первой инструкции. Изображение может быть любым допустимым изображением - его особенно легко начать, потянув изображение из публичных репозиториев.

`FROM` должна быть первой командой без комментария в файле Docker.

`FROM` может появляться несколько раз в пределах одного файла Docker для создания нескольких изображений. Просто запишите последний вывод ID изображения с помощью фиксации перед каждой новой командой `FROM` .

Значения тегов или дайджеста являются необязательными. Если вы опустите любой из них, строитель принимает по умолчанию последний. Строитель возвращает ошибку, если она не может соответствовать значению тега.

Инструкция RUN

`RUN` имеет 2 формы:

```
RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on
Linux or cmd /S /C on Windows)
RUN ["executable", "param1", "param2"] (exec form)
```

Команда `RUN` будет выполнять любые команды в новом слое поверх текущего изображения и фиксировать результаты. Результирующее зафиксированное изображение будет использоваться для следующего шага в `Dockerfile` .

Инструкции Layout `RUN` и генерирующие коммиты соответствуют основным понятиям Docker, где коммиты дешевы, и контейнеры могут быть созданы из любой точки в истории изображения, подобно контролю источника.

Форма exec позволяет избежать перебора строк оболочки и команд `RUN` с использованием базового изображения, которое не содержит указанный исполняемый файл оболочки.

По умолчанию оболочка для формы оболочки может быть изменена с помощью команды `SHELL` .

В форме оболочки вы можете использовать `\` (обратную косую черту), чтобы продолжить одну инструкцию `RUN` на следующую строку. Например, рассмотрим эти две строки:

```
RUN /bin/bash -c 'source $HOME/.bashrc ;\
echo $HOME'
```

Вместе они эквивалентны этой единственной строке:

```
RUN /bin/bash -c 'source $HOME/.bashrc ; echo $HOME'
```

Примечание. Чтобы использовать другую оболочку, отличную от '/ bin / sh', используйте форму exec, проходящую в нужной оболочке. Например, `RUN ["/bin/bash", "-c", "echo hello"]`

Примечание. Форма exec анализируется как массив JSON, что означает, что вы должны

использовать двойные кавычки (") вокруг слов, а не одиночных кавычек (').

Примечание. В отличие от формы оболочки форма exec не вызывает командную оболочку. Это означает, что нормальной обработки оболочки не происходит. Например, `RUN [ "echo", "$HOME" ]` не будет делать замену переменных в `$HOME` . Если вы хотите обработать оболочку, то либо используйте форму оболочки, либо выполните оболочку напрямую, например: `RUN [ "sh", "-c", "echo $HOME" ]` .

Примечание. В форме JSON необходимо избегать обратных косых черт. Это особенно актуально для Windows, где обратная косая черта - разделитель путей. Следующая строка в противном случае была бы обработана как форма оболочки из-за недействительности JSON и непредвиденного сбоя: `RUN ["c:\windows\system32\tasklist.exe"]`

Правильный синтаксис для этого примера: `RUN ["c:\\windows\\system32\\tasklist.exe"]`

Кэш для команд `RUN` автоматически не отменяется во время следующей сборки. Кэш для команды, такой как `RUN apt-get dist-upgrade -y` будет использоваться повторно во время следующей сборки. Кэш для команд `RUN` может быть недействительным с использованием флага -no-cache, например docker build --no-cache.

Дополнительную информацию см. В Руководстве по лучшей практике Dockerfile.

Кэш для команд `RUN` может быть аннулирован инструкциями `ADD` . Подробнее см. Ниже.

Инструкция ONBUILD

```
ONBUILD [INSTRUCTION]
```

Инструкция `ONBUILD` добавляет к изображению триггерную инструкцию, которая будет выполнена позднее, когда изображение используется в качестве основы для другой сборки. Триггер будет выполняться в контексте нисходящей сборки, как если бы он был вставлен сразу после инструкции `FROM` в нисходящем файле Docker.

Любая инструкция сборки может быть зарегистрирована как триггер.

Это полезно, если вы создаете образ, который будет использоваться в качестве базы для создания других изображений, например среды создания приложения или демона, который может быть настроен с учетом конфигурации пользователя.

Например, если ваш образ является многоразовым конструктором приложений Python, для его использования в конкретном каталоге потребуется исходный код приложения, и после этого может потребоваться вызывать скрипт сборки. Вы не можете просто вызывать `ADD` и `RUN` сейчас, потому что у вас еще нет доступа к исходному коду приложения, и для каждой сборки приложения он будет отличаться. Вы можете просто предоставить разработчикам приложений шаблонный файл Docker для копирования-вставки в свое приложение, но это неэффективно, подвержено ошибкам и сложно обновить, поскольку оно смешивается с

кодом приложения.

Решение состоит в том, чтобы использовать `ONBUILD` для регистрации предварительных инструкций для запуска позже, на следующем этапе сборки.

Вот как это работает:

Когда он встречает инструкцию `ONBUILD` , строитель добавляет триггер к метаданным создаваемого образа. Эта инструкция не влияет на текущую сборку.

В конце сборки список всех триггеров хранится в манифесте изображения под ключ OnBuild. Их можно проверить с помощью команды `docker inspect` . Позже изображение может быть использовано в качестве основы для новой сборки, используя инструкцию `FROM` . В процессе обработки команды `FROM` строитель нисходящего потока ищет триггеры `ONBUILD` и выполняет их в том же порядке, в котором они были зарегистрированы. Если какой-либо из триггеров терпит неудачу, команда `FROM` прерывается, что, в свою очередь, приводит к сбою сборки. Если все триггеры преуспевают, команда `FROM` завершается, и сборка продолжается, как обычно.

После запуска триггеры очищаются от окончательного изображения. Другими словами, они не унаследованы сборниками «grand-children».

Например, вы можете добавить что-то вроде этого:

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

Предупреждение: `ONBUILD` использование `ONBUILD ONBUILD` с использованием `ONBUILD ONBUILD` запрещен.

Предупреждение: инструкция `ONBUILD` может не запускать инструкции `FROM` или `MAINTAINER` .

Инструкция STOPSIGNAL

```
STOPSIGNAL signal
```

Инструкция `STOPSIGNAL` устанавливает сигнал системного вызова, который будет отправлен в контейнер для выхода. Этот сигнал может быть допустимым числом без знака, которое соответствует позиции в таблице syscall ядра, например 9, или имени сигнала в формате SIGNAME, например SIGKILL.

Инструкция HEALTHCHECK

Инструкция `HEALTHCHECK` имеет две формы:

```
HEALTHCHECK [OPTIONS] CMD command (check container health by running a command inside the
container)
HEALTHCHECK NONE (disable any healthcheck inherited from the base image)
```

Команда `HEALTHCHECK` сообщает Docker, как тестировать контейнер, чтобы проверить, что он все еще работает. Это может обнаружить такие случаи, как веб-сервер, который застрял в бесконечном цикле и не может обрабатывать новые соединения, даже несмотря на то, что процесс сервера все еще работает.

Когда в контейнере указан медицинский осмотр, у него есть состояние здоровья в дополнение к его нормальному состоянию. Первоначально этот статус запускается. Всякий раз, когда проходит проверка здоровья, он становится здоровым (независимо от того, в каком состоянии он был ранее). После определенного количества последовательных сбоев он становится нездоровым.

Параметры, которые могут отображаться перед `CMD` :

```
--interval=DURATION (default: 30s)
--timeout=DURATION (default: 30s)
--retries=N (default: 3)
```

Сначала проверка работоспособности начнется через несколько секунд после запуска контейнера, а затем снова через несколько секунд после завершения каждой предыдущей проверки.

Если один прогон проверки занимает больше времени, чем тайм-аут, то проверка считается неудачной.

Это требует повторных попыток проверки работоспособности контейнера, который считается нездоровым.

В `HEALTHCHECK` может быть только одна инструкция `Dockerfile` . Если вы перечислите более одного, тогда `HEALTHCHECK` силу только последний `HEALTHCHECK` .

Команда после ключевого слова `CMD` может быть либо командой оболочки (например, `HEALTHCHECK CMD /bin/check-running`), либо массивом exec (как и в других командах Dockerfile, см., Например, `ENTRYPOINT` для деталей).

Статус выхода команды указывает состояние работоспособности контейнера. Возможные значения:

- `0: success` - контейнер здоров и готов к использованию
- `1: unhealthy` - контейнер работает неправильно
- `2: starting` - контейнер еще не готов к использованию, но работает правильно

Если зонд возвращает 2 («запуск»), когда контейнер уже вышел из состояния «запуска», тогда он рассматривается как «нездоровый».

Например, чтобы проверить каждые пять минут или так, чтобы веб-сервер мог обслуживать главную страницу сайта в течение трех секунд:

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

Чтобы помочь отлаживать сбойные зонды, любой выходной текст (кодированный UTF-8), который команда записывает на stdout или stderr, будет сохранен в состоянии работоспособности и может быть запрошен с `docker inspect` . Такой вывод должен быть коротким (в настоящий момент хранятся только первые 4096 байтов).

Когда состояние работоспособности контейнера изменяется, событие `health_status` генерируется с новым статусом.

Функция `HEALTHCHECK` была добавлена в Docker 1.12.

Инструкция SHELL

```
SHELL ["executable", "parameters"]
```

Команда `SHELL` позволяет использовать оболочку по умолчанию, используемую для оболочки команд команд для переопределения. Стандартная оболочка в Linux - это `["/bin/sh", "-c"]` , а в Windows - `["cmd", "/S", "/C"]` . Инструкция `SHELL` должна быть записана в форме JSON в файле Docker.

Инструкция `SHELL` особенно полезна в Windows, где есть две обычно используемые и совершенно разные родные оболочки: cmd и powershell, а также альтернативные оболочки, включая sh.

Инструкция `SHELL` может появляться несколько раз. Каждая команда `SHELL` отменяет все предыдущие инструкции `SHELL` и влияет на все последующие инструкции. Например:

```
FROM windowsservercore

# Executed as cmd /S /C echo default
RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
SHELL ["cmd", "/S"", "/C"]
RUN echo hello
```

Инструкция `SHELL` может быть затронута следующими инструкциями, когда их форма

оболочки используется в файле Docker: `RUN` , `CMD` и `ENTRYPOINT` .

Следующий пример - это общий шаблон, найденный в Windows, который можно упростить с помощью инструкции `SHELL` :

```
...
RUN powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
...
```

Команда, вызываемая докером, будет:

```
cmd /S /C powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
```

Это неэффективно по двум причинам. Во-первых, вызывается ненужный командный процессор cmd.exe (aka shell). Во-вторых, для каждой инструкции `RUN` в форме оболочки требуется дополнительная команда powershell-command, префиксная команда.

Чтобы сделать это более эффективным, можно использовать один из двух механизмов. Один из них - использовать форму JSON команды `RUN` такую как:

```
...
RUN ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\foo.txt\""]
...
```

Хотя форма JSON недвусмысленна и не использует ненужный cmd.exe, для этого требуется более многословие посредством двойного кавычки и экранирования. Альтернативным механизмом является использование инструкции `SHELL` и формы оболочки, что делает более естественным синтаксис для пользователей Windows, особенно в сочетании с директивой анализа парсера:

```
# escape=`

FROM windowsservercore
SHELL ["powershell","-command"]
RUN New-Item -ItemType Directory C:\Example
ADD Execute-MyCmdlet.ps1 c:\example\
RUN c:\example\Execute-MyCmdlet -sample 'hello world'
```

В результате чего:

```
PS E:\docker\build\shell> docker build -t shell .
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM windowsservercore
 ---> 5bc36a335344
Step 2 : SHELL powershell -command
 ---> Running in 87d7a64c9751
 ---> 4327358436c1
Removing intermediate container 87d7a64c9751
Step 3 : RUN New-Item -ItemType Directory C:\Example
 ---> Running in 3e6ba16b8df9
```

```
Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         6/2/2016   2:59 PM                Example


 ---> 1f1dfdcec085
Removing intermediate container 3e6ba16b8df9
Step 4 : ADD Execute-MyCmdlet.ps1 c:\example\
 ---> 6770b4c17f29
Removing intermediate container b139e34291dc
Step 5 : RUN c:\example\Execute-MyCmdlet -sample 'hello world'
 ---> Running in abdcf50dfd1f
Hello from Execute-MyCmdlet.ps1 - passed hello world
 ---> ba0e25255fda
Removing intermediate container abdcf50dfd1f
Successfully built ba0e25255fda
PS E:\docker\build\shell>
```

Инструкция `SHELL` также может использоваться для изменения способа работы оболочки. Например, используя `SHELL cmd /S /C /V:ON|OFF` в Windows, можно было бы изменить семантику расширения переменных среды с задержкой.

Инструкция `SHELL` может также использоваться в Linux, если требуется чередование оболочки zsh, csh, tcsh и других.

Функция `SHELL` была добавлена в Docker 1.12.

Установка пакетов Debian / Ubuntu

Запустите установку в команде с одним запуском, чтобы объединить обновление и установить. Если позже вы добавите больше пакетов, это снова запустит обновление и установит все необходимые пакеты. Если обновление запускается отдельно, оно будет кэшироваться и пакеты могут завершиться неудачей. Установка интерфейсов в неинтерактивный и передача -y для установки необходима для сценариев установки. Очистка и очистка в конце установки минимизирует размер слоя.

```
FROM debian

RUN apt-get update \
 && DEBIAN_FRONTEND=noninteractive apt-get install -y \
    git \
    openssh-client \
    sudo \
    vim \
    wget \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*
```

Прочитайте Dockerfiles онлайн: https://riptutorial.com/ru/docker/topic/3161/dockerfiles

глава 6: Iptables with Docker

Вступление

В этом разделе рассказывается о том, как ограничить доступ к контейнерам докеров из внешнего мира с помощью iptables.

Для нетерпеливых людей вы можете проверить примеры. Для остальных, пожалуйста, прочитайте раздел примечаний, чтобы понять, как создавать новые правила.

Синтаксис

- iptables -I DOCKER [RULE ...] [ACCEPT | DROP] // Чтобы добавить правило в верхнюю часть таблицы DOCKER
- iptables -D DOCKER [RULE ...] [ACCEPT | DROP] // Чтобы удалить правило из таблицы DOCKER
- ipset restore </etc/ipfriends.conf // Чтобы перенастроить ваш ipps- *адрес ipset*

параметры

параметры	подробности
ext_if	Ваш внешний интерфейс на хосте Docker.
XXX.XXX.XXX.XXX	Необходимо указать конкретный IP-адрес, на который должен быть предоставлен доступ к контейнерам Docker.
YYY.YYY.YYY.YYY	Еще один IP-адрес, на который должен быть предоставлен доступ к контейнерам Docker.
ipfriends	Имя ipset, определяющее IP-адреса, позволило получить доступ к вашим контейнерам Docker.

замечания

Эта проблема

Настройка правил iptables для контейнеров Docker немного сложна. Сначала вы думаете, что «классические» правила брандмауэра должны делать трюк.

Например, предположим, что вы сконфигурировали контейнер nginx-proxy + несколько контейнеров-служб для предоставления через HTTPS некоторых персональных веб-сервисов. Затем такое правило должно предоставлять доступ к вашим веб-службам только для IP XXX.XXX.XXX.XXX.

```
$ iptables -A INPUT -i eth0 -p tcp -s XXX.XXX.XXX.XXX -j ACCEPT
$ iptables -P INPUT DROP
```

Это не сработает, ваши контейнеры все еще доступны для всех.

Действительно, контейнеры Docker не являются хост-услугами. Они полагаются на виртуальную сеть вашего хоста, а хост выступает в качестве шлюза для этой сети. Что касается шлюзов, маршрутизируемый трафик не обрабатывается таблицей INPUT, а таблицей FORWARD, что делает правило выше, чем неэффективным.

Но это еще не все. Фактически, демон Docker создает множество правил iptables, когда он начинает делать свою магию в отношении подключения к сети контейнеров. В частности, создается таблица DOCKER для обработки правил, касающихся контейнеров, путем пересылки трафика из таблицы FORWARD в эту новую таблицу.

```
$ iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination
DOCKER-ISOLATION  all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             ctstate RELATED,ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere             ctstate RELATED,ESTABLISHED
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain DOCKER (2 references)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             172.18.0.4           tcp dpt:https
ACCEPT     tcp  --  anywhere             172.18.0.4           tcp dpt:http

Chain DOCKER-ISOLATION (1 references)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere
DROP       all  --  anywhere             anywhere
RETURN     all  --  anywhere             anywhere
```

Решение

Если вы проверяете официальную документацию (https://docs.docker.com/v1.5/articles/networking/) , то предоставляется первое решение ограничить доступ контейнера Docker к одному конкретному IP- адресу .

```
$ iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
```

Действительно, добавление правила в верхней части таблицы DOCKER - хорошая идея. Это не мешает правилам, автоматически настроенным Docker, и это просто. Но два основных недостатка:

- Во-первых, что, если вам нужно получить доступ с двух IP вместо одного? Здесь может быть принят только один IP-адрес src, другие будут исключены, чтобы предотвратить это.
- Во-вторых, что делать, если вашему докере нужен доступ в Интернет? При этом ни один запрос не будет выполнен, так как только сервер 8.8.8.8 может ответить на них.
- Наконец, что делать, если вы хотите добавить другие логики? Например, предоставить доступ любому пользователю к вашему веб-серверу, работающему по протоколу HTTP, но ограничивать все остальное конкретным IP-адресом.

Для первого наблюдения мы можем использовать *ipset* . Вместо того, чтобы разрешать один IP в приведенном выше правиле, мы разрешаем всем IP-адресам из предопределенного ipset. В качестве бонуса ipset можно обновить без необходимости переопределять правило iptable.

```
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
```

Для второго наблюдения это каноническая проблема для брандмауэров: если вам разрешено связаться с сервером через брандмауэр, тогда брандмауэр должен разрешить серверу отвечать на ваш запрос. Это можно сделать, разрешив пакеты, связанные с установленным соединением. Для докерной логики он дает:

```
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Последнее наблюдение фокусируется на одной точке: правила iptables необходимы. В самом деле, дополнительная логика для ПРИНИМАНИЯ некоторых подключений (в том числе связанных с ESTABLISHED соединениями) должна быть помещена в верхнюю часть таблицы DOCKER до правила DROP, которое отрицает все остальные соединения, не соответствующие ipset.

Поскольку мы используем опцию -I iptable, которая вводит правила в верхней части таблицы, предыдущие правила iptables должны быть вставлены в обратном порядке:

```
// Drop rule for non matching IPs
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
// Then Accept rules for established connections
```

```
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 3rd custom accept rule
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 2nd custom accept rule
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 1st custom accept rule
```

Учитывая все это, вы можете теперь проверить примеры, иллюстрирующие эту конфигурацию.

Examples

Ограничить доступ к контейнерам Docker к набору IP-адресов

Во-первых, при необходимости установите *ipset* . Пожалуйста, обратитесь к вашему дистрибутиву, чтобы узнать, как это сделать. Например, вот команда для Debian-подобных дистрибутивов.

```
$ apt-get update
$ apt-get install ipset
```

Затем создайте файл конфигурации, чтобы определить ipset, содержащий IP-адреса, для которых вы хотите открыть доступ к своим контейнерам Docker.

```
$ vi /etc/ipfriends.conf
# Recreate the ipset if needed, and flush all entries
create -exist ipfriends hash:ip family inet hashsize 1024 maxelem 65536
flush
# Give access to specific ips
add ipfriends XXX.XXX.XXX.XXX
add ipfriends YYY.YYY.YYY.YYY
```

Загрузите этот ipset.

```
$ ipset restore < /etc/ipfriends.conf
```

Убедитесь, что ваш демон Docker запущен: при вводе следующей команды не следует выводить никаких ошибок.

```
$ docker ps
```

Вы готовы вставить свои правила iptables. Вы **должны** уважать заказ.

```
// All requests of src ips not matching the ones from ipset ipfriends will be dropped.
$ iptables -I DOCKER -i ext_if -m set ! --match-set ipfriends src -j DROP
// Except for requests coming from a connection already established.
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Если вы хотите создать новые правила, вам нужно будет удалить все пользовательские правила, которые вы добавили, прежде чем вставлять новые.

```
$ iptables -D DOCKER -i ext_if -m set ! --match-set ipfriends src -j DROP
$ iptables -D DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
```

Настройка доступа к ограничениям при запуске Docker-демона

Работа в процессе

Некоторые пользовательские правила iptables

Работа в процессе

Прочитайте Iptables with Docker онлайн: https://riptutorial.com/ru/docker/topic/9201/iptables-with-docker

глава 7: бегущий консул в докере 1.12 рой

Examples

Запуск консула в докер 1.12 рой

Это зависит от официального изображения докеры-консула для запуска консула в кластерном режиме в докеровском рое с новым режимом роя в Docker 1.12. Этот пример основан на http://qnib.org/2016/08/11/consul-service/ . Вкратце идея состоит в том, чтобы использовать две службы докеров, которые говорят друг с другом. Это решает проблему, из-за которой вы не можете знать ips отдельных контейнеров консула спереди и позволяет вам полагаться на dns docker swarm.

Это предполагает, что у вас уже есть рабочий кластер 1.12 с кластерами с минимум тремя узлами.

Вы можете настроить драйвер журнала на своих демонах докеров, чтобы вы могли видеть, что происходит. Я использовал драйвер syslog для этого: установите параметр `--log-driver=syslog` на dockerd.

Сначала создайте оверлейную сеть для консула:

```
docker network create consul-net -d overlay
```

Теперь загрузите кластер только с одним узлом (по умолчанию --replicas равно 1):

```
docker service create --name consul-seed \
  -p 8301:8300 \
  --network consul-net \
  -e 'CONSUL_BIND_INTERFACE=eth0' \
  consul agent -server -bootstrap-expect=3  -retry-join=consul-seed:8301 -retry-join=consul-cluster:8300
```

Теперь у вас должен быть кластер из 1 узла. Теперь поднимите вторую службу:

```
docker service create --name consul-cluster \
  -p 8300:8300 \
  --network consul-net \
  --replicas 3 \
  -e 'CONSUL_BIND_INTERFACE=eth0' \
  consul agent -server -retry-join=consul-seed:8301 -retry-join=consul-cluster:8300
```

Теперь у вас должен быть кластер с четырьмя узлами. Вы можете проверить это, запустив любой из контейнеров докеров:

```
docker exec <containerid> consul members
```

Прочитайте бегущий консул в докере 1.12 рой онлайн:
https://riptutorial.com/ru/docker/topic/6437/бегущий-консул-в-докере-1-12-рой

глава 8: безопасность

Вступление

Чтобы обновлять наши изображения для патчей безопасности, нам нужно знать, с какого базового образа мы зависим

Examples

Как найти, с какого изображения происходит наше изображение

В качестве примера давайте посмотрим на контейнер Wordpress

Файл Docker начинается с FROM php: 5.6-apache

поэтому мы переходим к файлу Docker выше https://github.com/docker-library/php/blob/master/5.6/apache/Dockerfile

и мы находим FROM debian: jessie Итак, это означает, что мы устанавливаем патч безопасности для Debian jessie, нам нужно снова создать наш образ.

Прочитайте безопасность онлайн: https://riptutorial.com/ru/docker/topic/8077/безопасность

глава 9: Докер в Докере

Examples

Контейнер Jenkins CI с использованием Docker

В этой главе описывается, как настроить контейнер Docker с Jenkins внутри, который способен отправлять команды Docker на установку Docker (Docker Daemon) хоста. Эффективно использование Docker в Docker. Чтобы достичь этого, нам нужно создать пользовательское изображение Docker Image, основанное на произвольной версии официального Jenkins Docker Image. Файл Dockerfile (Инструкция по созданию образа) выглядит следующим образом:

```
FROM jenkins

USER root

RUN cd /usr/local/bin && \
curl https://master.dockerproject.org/linux/amd64/docker > docker  && \
chmod +x docker  && \
groupadd -g 999 docker && \
usermod -a -G docker jenkins

USER Jenkins
```

Этот Dockerfile создает изображение, содержащее двоичные файлы клиента Docker, этот клиент используется для связи с Docker Daemon. В этом случае Деккер-демон Хозяина. Оператор `RUN` в этом файле также создает UNIX-группу с UID 999 и добавляет к ней пользователя Jenkins. Почему именно это необходимо, описано в следующей главе. С помощью этого изображения мы можем запустить сервер Jenkins, который может использовать команды Docker, но если мы просто запустим это изображение, клиент Docker, который мы установили внутри изображения, не сможет связаться с Docker Daemon Host. Эти два компонента взаимодействуют через UNIX Socket `/var/run/docker.sock` . В Unix это файл, как и все остальное, поэтому мы можем легко установить его внутри контейнера Jenkins. Это делается с помощью команды `docker run -v /var/run/docker.sock:/var/run/docker.sock --name jenkins MY_CUSTOM_IMAGE_NAME` . Но этот смонтированный файл принадлежит `docker:root` и из-за этого Dockerfile создает эту группу с хорошо известным UID и добавляет к ней пользователя Jenkins. Теперь Jenkins Container действительно способен работать и использовать Docker. В процессе выполнения команда запуска также должна содержать `-v jenkins_home:/var/jenkins_home` для резервного копирования каталога Jenkins_home и, конечно же, сопоставления портов для доступа к серверу по сети.

Прочитайте Докер в Докере онлайн: https://riptutorial.com/ru/docker/topic/8012/докер-в-докере

глава 10: Докерная машина

Вступление

Удаленное управление несколькими хост-компьютерами докеров.

замечания

`docker-machine` управляет удаленными хостами, работающими с Docker.

Инструмент командной строки `docker-machine` управляет жизненным циклом полной машины с использованием драйверов конкретного поставщика. Его можно использовать для выбора «активной» машины. После выбора можно использовать активную машину, как если бы это был локальный Docker Engine.

Examples

Получить текущую информацию о среде Docker Machine

Все это команды оболочки.

`docker-machine env` чтобы получить текущую конфигурацию докеров-машин по умолчанию

`eval $(docker-machine env)` чтобы получить текущую конфигурацию докер-машины и установить текущую среду оболочки для использования этой док-машины.

Если ваша оболочка настроена на использование прокси-сервера, вы можете указать опцию -no-proxy, чтобы обойти прокси-сервер при подключении к вашей докер-машине: `eval $(docker-machine env --no-proxy)`

Если у вас несколько докеров-машин, вы можете указать имя машины в качестве аргумента: `eval $(docker-machine env --no-proxy machinename)`

SSH в докерную машину

Все это команды оболочки

- Если вам нужно войти в рабочую док-машину напрямую, вы можете сделать это:

`docker-machine ssh` в ssh в стандартную докер-машину

`docker-machine ssh machinename` для ssh в нестандартную докер-машину

- Если вы просто хотите запустить одну команду, вы можете сделать это. Чтобы

запустить `uptime` на докере-машине по умолчанию, чтобы узнать, как долго он работает, запустите `docker-machine ssh default uptime`

Создать докерную машину

Использование `docker-machine` - лучший способ установки Docker на машину. Он автоматически применит лучшие доступные параметры безопасности, включая создание уникальной пары SSL-сертификатов для взаимной аутентификации и SSH-ключей.

Чтобы создать локальный компьютер с помощью Virtualbox:

```
docker-machine create --driver virtualbox docker-host-1
```

Чтобы установить Docker на существующую машину, используйте `generic` драйвер:

```
docker-machine -D create -d generic --generic-ip-address 1.2.3.4 docker-host-2
```

Опция `--driver` сообщает docker, как создать машину. Список поддерживаемых драйверов см. В следующих разделах:

- официально поддерживается
- третья вечеринка

Список докеров

Листинг docker-machines вернет состояние, адрес и версию Docker для каждой докерной машины.

```
 docker-machine ls
```

Выведет что-то вроде:

```
NAME             ACTIVE   DRIVER    STATE     URL                         SWARM   DOCKER
ERRORS
docker-machine-1 -        ovh       Running   tcp://1.2.3.4:2376                  v1.11.2
docker-machine-2 -        generic   Running   tcp://1.2.3.5:2376                  v1.11.2
```

Чтобы просмотреть список работающих машин:

```
docker-machine ls --filter state=running
```

Чтобы просмотреть машины с ошибками:

```
docker-machine ls --filter state=
```

Чтобы перечислить машины, имена которых начинаются с «side-project-», используйте фильтр Голанга:

```
docker-machine ls --filter name="^side-project-"
```

Чтобы получить только список URL-адресов машины:

```
docker-machine ls --format '{{ .URL }}'
```

См. Https://docs.docker.com/machine/reference/ls/ для полной ссылки на команду.

Обновление докерной машины

Модернизация докерной машины подразумевает простои и может потребовать строгания. Чтобы обновить докер-машину, запустите:

```
docker-machine upgrade docker-machine-name
```

У этой команды нет параметров

Получить IP-адрес устройства для докеров

Чтобы получить IP-адрес устройства докеров, вы можете сделать это с помощью этой команды:

```
docker-machine ip machine-name
```

Прочитайте Докерная машина онлайн: https://riptutorial.com/ru/docker/topic/1349/докерная-машина

глава 11: Докерная сеть

Examples

Как найти контейнер контейнера ip

Вам нужно узнать IP-адрес контейнера, запущенного на хосте, чтобы вы могли, например, подключиться к веб-серверу, запущенному в нем.

`docker-machine` - это то, что используется в MacOSX и Windows.

Во-первых, перечислите свои машины:

```
$ docker-machine ls

NAME      ACTIVE   DRIVER       STATE     URL                         SWARM
default   *        virtualbox   Running   tcp://192.168.99.100:2376
```

Затем выберите одну из машин (по умолчанию она называется по умолчанию) и:

```
$ docker-machine ip default

192.168.99.100
```

Создание сети Docker

```
docker network create app-backend
```

Эта команда создаст простую `appBackend` сеть под названием `appBackend` . По умолчанию в эту сеть нет контейнеров.

Листинг сетей

```
docker network ls
```

Эта команда перечисляет все сети, созданные на локальном хосте Docker. Он включает в себя сеть моста `bridge` по умолчанию, сеть хост- `host` и нулевую `null` сеть. Все контейнеры по умолчанию привязаны к мостовой сети `bridge` по умолчанию.

Добавить контейнер в сеть

```
docker network connect app-backend myAwesomeApp-1
```

Эта команда присоединяет контейнер `myAwesomeApp-1` к сети `app-backend` . Когда вы

добавляете контейнер в определенную пользователем сеть, встроенный DNS-преобразователь (который не является полнофункциональным DNS-сервером и не экспортируется) позволяет каждому контейнеру в сети разрешать друг другу контейнер в той же сети. Этот простой DNS-ресивер недоступен в `bridge` сети `bridge` по умолчанию.

Отсоединить контейнер от сети

```
docker network disconnect app-backend myAwesomeApp-1
```

Эта команда отделяет контейнер `myAwesomeApp-1` от сети `app-backend` . Контейнер больше не сможет связываться с другими контейнерами в сети, из которой он был отключен, и не использовать встроенный DNS-реверсор для поиска других контейнеров в сети, из которой он был отсоединен.

Удаление сети Docker

```
docker network rm app-backend
```

Эта команда удаляет пользовательскую сеть `app-backend` с хостом Docker. Все контейнеры в сети, которые иначе не подключены через другую сеть, потеряют связь с другими контейнерами. Невозможно удалить мостовую сеть `bridge` по умолчанию, сеть `host` хоста или `null` нулевую сеть.

Осмотрите сеть Docker

```
docker network inspect app-backend
```

Эта команда выведет информацию о сети `app-backend` .

Результат этой команды должен выглядеть так:

```
[
    {
        "Name": "foo",
        "Id": "a0349d78c8fd7c16f5940bdbaf1adec8d8399b8309b2e8a969bd4e3226a6fc58",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Internal": false,
```

```
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Прочитайте Докерная сеть онлайн: https://riptutorial.com/ru/docker/topic/3221/докерная-сеть

глава 12: Докеры - сетевые режимы (мост, hots, сопоставленный контейнер и ни один).

Вступление

Начиная

Bridge Mode Это значение по умолчанию и привязано к мосту docker0. Поместите контейнер в полностью отдельное пространство имен.

Хост-режим Когда контейнер - это всего лишь процесс, запущенный на хосте, мы присоединяем контейнер к сетевому сетевому адаптеру.

Mapped Container Mode Этот режим по существу отображает новый контейнер в существующий сетевой стек контейнеров. Он также называется «контейнер в режиме контейнера».

Нет. Он сообщает, что докер ставит контейнер в свой собственный сетевой стек без конфигурации

Examples

Режим моста, режим хоста и режим отображения контейнера

Режим моста

```
$ docker run -d --name my_app -p 10000:80 image_name
```

Обратите внимание, что нам не нужно указывать **--net = bridge,** потому что это рабочий режим по умолчанию для докеров. Это позволяет запускать несколько контейнеров для работы на одном и том же хосте без какого-либо назначения динамического порта. Таким образом, режим **BRIDGE** позволяет избежать столкновения портов, и это безопасно, поскольку каждый контейнер имеет собственное пространство имен частной сети.

Режим хоста

```
$ docker run -d --name my_app -net=host image_name
```

Поскольку он использует пространство имен хост-сети, нет необходимости в специальной конфигурации, но может привести к проблеме безопасности.

Режим отображения контейнера

Этот режим по существу отображает новый контейнер в существующий сетевой стек контейнеров. Это означает, что сетевые ресурсы, такие как IP-адрес и сопоставления портов первого контейнера, будут совместно использоваться вторым контейнером. Это также называется режимом «контейнер в контейнере». Предположим, у вас есть две задачи: web_container_1 и web_container_2, и мы запустим web_container_2 в отображаемом режиме контейнера. Давайте сначала загрузим web_container_1 и запустим его в отдельный режим со следующей командой,

```
$ docker run -d --name web1 -p 80:80 USERNAME/web_container_1
```

Как только он загрузится, давайте посмотрим и убедитесь, что он работает. Здесь мы просто сопоставили порт в контейнер, который работает в режиме моста по умолчанию. Теперь давайте запустим второй контейнер в режиме отображения контейнера. Мы сделаем это с помощью этой команды.

```
$ docker run -d --name web2 --net=container:web1 USERNAME/web_container_2
```

Теперь, если вы просто получите информацию о интерфейсе на обоих контурах, вы получите ту же конфигурацию сети. Это фактически включает режим HOST, который отображает точную информацию о хосте. Первый contaienr работал в режиме моста по умолчанию, а второй контейнер запускался в режиме отображения контейнера. Мы можем получить очень похожие результаты, запустив первый контейнер в режиме хозяина и второй контейнер в отображаемом режиме контейнера.

Прочитайте Докеры - сетевые режимы (мост, hots, сопоставленный контейнер и ни один). онлайн: https://riptutorial.com/ru/docker/topic/9643/докеры---сетевые-режимы--мост--hots--сопоставленный-контейнер-и-ни-один--

глава 13: Запуск контейнеров

Синтаксис

- docker run [ОПЦИИ] ИЗОБРАЖЕНИЕ [КОМАНДА] [ARG ...]

Examples

Запуск контейнера

```
docker run hello-world
```

Это позволит получить последнее изображение приветствия из Docker Hub (если у вас его еще нет), создать новый контейнер и запустить его. Вы должны увидеть сообщение о том, что ваша установка работает правильно.

Выполнение другой команды в контейнере

```
docker run docker/whalesay cowsay 'Hello, StackExchange!'
```

Эта команда сообщает Docker о создании контейнера из образа `docker/whalesay` и запускает команду `cowsay 'Hello, StackExchange!'` в этом. Он должен напечатать фотографию кита, говорящего « `Hello, StackExchange!` к вашему терминалу.

Если входная точка на изображении по умолчанию, вы можете запустить любую команду, доступную на изображении:

```
docker run docker/whalesay ls /
```

Если он был изменен во время сборки изображения, вам нужно отменить его обратно по умолчанию

```
docker run --entrypoint=/bin/bash docker/whalesay -c ls /
```

Автоматически удалять контейнер после его запуска

Обычно контейнер Docker сохраняется после его выхода. Это позволяет снова запустить контейнер, проверить его файловую систему и т. Д. Однако иногда вы хотите запустить контейнер и удалить его сразу же после его выхода. Например, чтобы выполнить команду или показать файл из файловой системы. Docker предоставляет `--rm` командной строки `--rm` для этой цели:

```
docker run --rm ubuntu cat /etc/hosts
```

Это создаст контейнер из образа «ubuntu», покажет содержимое файла **/ etc / hosts,** а затем удалит контейнер сразу после его выхода. Это помогает предотвратить очистку контейнеров после того, как вы закончите экспериментировать.

> **Примечание.** Флаг `--rm` не работает в сочетании с флагом `-d` (`--detach`) в докере <1.13.0.

Когда `--rm` флаг `--rm` , Docker также удаляет тома, связанные с контейнером, когда контейнер удален. Это похоже на запуск `docker rm -v my-container` . ***Удаляются только тома, которые указаны без имени*** .

Например, при `docker run -it --rm -v /etc -v logs:/var/log centos /bin/produce_some_logs` , объем `/etc` будет удален, но объем `/var/log` не будет. Объемы, унаследованные через --volumes-from, будут удалены с той же логикой - если исходный том был указан с именем, он не будет удален.

Указание имени

По умолчанию контейнерам, созданным с помощью `docker run` , присваивается случайное имя, например `small_roentgen` или `modest_dubinsky` . Эти имена не особенно полезны при определении цели контейнера. Можно указать имя для контейнера, передав параметр командной строки `--name` :

```
docker run --name my-ubuntu ubuntu:14.04
```

Имена должны быть уникальными; если вы передадите имя, которое уже использует другой контейнер, Docker напечатает ошибку и новый контейнер не будет создан.

Указание имени будет полезно при обращении к контейнеру в сети Docker. Это работает как для фоновых, так и для передних контейнеров Docker.

Контейнеры в сети моста по умолчанию **должны** быть связаны для связи по имени.

Связывание порта контейнера с хостом

```
docker run -p "8080:8080" myApp
docker run -p "192.168.1.12:80:80" nginx
docker run -P myApp
```

Чтобы использовать порты на хосте, были выставлены в изображении (через директиву `EXPOSE` Dockerfile или `--expose` командной строки `--expose` для `docker run`), эти порты должны быть привязаны к хосту с помощью команды `-p` или `-P` лайн. Использование `-p` требует `-p` конкретного порта (и дополнительного интерфейса хоста). Использование опции

командной строки верхнего регистра `-P` заставит Docker связывать *все* открытые порты в изображении контейнера с хостом.

Политика перезагрузки контейнера (запуск контейнера при загрузке)

```
docker run --restart=always -d <container>
```

По умолчанию Docker не перезапускает контейнеры, когда демон Docker перезапускается, например, после перезагрузки системы хоста. Docker предоставляет политику перезагрузки для ваших контейнеров, предоставляя `--restart` командной строки `--restart` . Поставка `--restart=always` всегда вызывает перезапуск контейнера после перезапуска Docker-демона. **Однако,** когда этот контейнер вручную остановлен (например, с помощью `docker stop <container>`), политика перезагрузки не будет применяться к контейнеру.

Можно указать несколько опций для опции `--restart` на основе требования (`--restart=[policy]`). Эти параметры влияют на то, как контейнер запускается при загрузке.

политика	Результат
нет	Значение по **умолчанию** . Не будет перезагружать контейнер автоматически, когда контейнер остановлен.
на провал [: макс-повторов]	Перезагрузитесь, только если контейнер выходит с ошибкой (`non-zero exit status`). Чтобы не перезапускать его неограниченно (в случае какой-либо проблемы), можно ограничить количество повторных попыток повторения попыток демона Docker.
всегда	Всегда перезапускайте контейнер независимо от состояния выхода. Когда вы укажете `always` , демон Docker попытается перезапустить контейнер на неопределенный срок. Контейнер также всегда запускается при запуске демона, независимо от текущего состояния контейнера.
если не-остановлен	Всегда перезагружайте контейнер независимо от его статуса выхода, но не запускайте его при запуске демона, если контейнер ранее был остановлен.

Запуск контейнера в фоновом режиме

Чтобы контейнер работал в фоновом режиме, `-d` командной строки -d во время запуска контейнера:

```
docker run -d busybox top
```

Опция `-d` запускает контейнер в отдельном режиме. Это также эквивалентно `-d=true` .

Контейнер в отдельном режиме не может быть удален автоматически при его остановке, это означает, что нельзя использовать параметр -rm в сочетании с опцией -d.

Назначение тома контейнеру

Том Docker - это файл или каталог, который сохраняется за пределами срока действия контейнера. Можно монтировать файл или каталог хоста в контейнер в виде тома (в обход UnionFS).

Добавьте том с `-v` командной строки `-v` :

```
docker run -d -v "/data" awesome/app bootstrap.sh
```

Это создаст том и подключит его к пути `/data` внутри контейнера.

- Примечание. Вы можете использовать флаг `--rm` для автоматического удаления тома при удалении контейнера.

Установка каталогов хоста

Чтобы монтировать файл или каталог хоста в контейнер:

```
docker run -d -v "/home/foo/data:/data" awesome/app bootstrap.sh
```

- **При указании каталога хоста должен быть указан абсолютный путь.**

Это приведет к установке каталога хоста `/home/foo/data` в `/data` внутри контейнера. Этот том «bind-installed host directory» - это то же самое, что и Linux `mount --bind` и поэтому временно монтирует каталог хоста по указанному пути контейнера в течение всего срока службы контейнера. Изменения объема из хоста или контейнера немедленно отражаются в другом, поскольку они являются одним и тем же местом назначения на диске.

Пример UNIX: установка относительной папки

```
docker run -d -v $(pwd)/data:/data awesome/app bootstrap.sh
```

Объемы именования

Том может быть назван путем подачи строки вместо пути к каталогу хоста, докер будет создавать тома с использованием этого имени.

```
docker run -d -v "my-volume:/data" awesome/app bootstrap.sh
```

После создания именованного тома тома затем можно будет использовать совместно с

другими контейнерами, используя это имя.

Настройка переменных среды

```
$ docker run -e "ENV_VAR=foo" ubuntu /bin/bash
```

Оба `-e` и `--env` могут использоваться для определения переменных среды внутри контейнера. Можно указать множество переменных среды, используя текстовый файл:

```
$ docker run --env-file ./env.list ubuntu /bin/bash
```

Пример файла переменной окружения:

```
# This is a comment
TEST_HOST=10.10.0.127
```

Флаг `--env-file` принимает имя файла в качестве аргумента и ожидает, что каждая строка будет в формате `VARIABLE=VALUE` , имитируя аргумент, переданный в `--env` . Строки комментариев должны иметь только префикс `#` .

Независимо от порядка этих трех флагов сначала обрабатывается `--env-file` , а затем `-e` / `--env` flags. Таким образом, любые переменные среды, поставляемые отдельно с `-e` или `--env` будут переопределять переменные, `--env-var` текстовый файл `--env-var` .

Указание имени хоста

По умолчанию контейнерам, созданным при запуске docker, присваивается произвольное имя хоста. Вы можете предоставить контейнеру другое имя хоста, передав флаг -hostname:

```
docker run --hostname redbox -d ubuntu:14.04
```

Запуск контейнера в интерактивном режиме

Чтобы запустить контейнер в интерактивном режиме, перейдите в опции `-it` :

```
$ docker run -it ubuntu:14.04 bash
root@8ef2356d919a:/# echo hi
hi
root@8ef2356d919a:/#
```

`-i` держит STDIN открытым, а `-t` выделяет псевдотематику.

Запуск контейнера с ограничениями памяти / свопа

Установите ограничение памяти и отключите ограничение подкачки

```
docker run -it -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

Установите как ограничение памяти, так и своп. В этом случае контейнер может использовать 300M памяти и 700M swap.

```
docker run -it -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash
```

Получение оболочки в работающий (отсоединенный) контейнер

Вход в рабочий контейнер

Пользователь может ввести запущенный контейнер в новую интерактивную оболочку bash с командой `exec` .

Скажем, что контейнер называется `jovial_morse` тогда вы можете получить интерактивную оболочку bash pseudo-TTY, запустив:

```
docker exec -it jovial_morse bash
```

Вход в рабочий контейнер с конкретным пользователем

Если вы хотите ввести контейнер в качестве конкретного пользователя, вы можете установить его с параметром `-u` или `--user` . Имя пользователя должно существовать в контейнере.

> `-u, --user` Имя пользователя или UID (формат: `<name|uid>[:<group|gid>]`)

Эта команда войдет в `jovial_morse` с пользователем `dockeruser`

```
docker exec -it -u dockeruser jovial_morse bash
```

Войдите в запущенный контейнер как root

Если вы хотите войти в систему под именем root, просто используйте параметр **`-u root`** . Корневой пользователь всегда существует.

```
docker exec -it -u root jovial_morse bash
```

Вход в изображение

Вы также можете войти в образ с помощью команды `run` , но для этого требуется имя изображения вместо имени контейнера.

```
docker run -it dockerimage bash
```

Вход в промежуточное изображение (отладка)

Вы также можете войти в промежуточное изображение, которое создается во время сборки Dockerfile.

Вывод `docker build .`

```
$ docker build .
Uploading context 10240 bytes
Step 1 : FROM busybox
Pulling repository busybox
 ---> e9aa60c60128MB/2.284 MB (100%) endpoint: https://cdn-registry-1.docker.io/v1/
Step 2 : RUN ls -lh /
 ---> Running in 9c9e81692ae9
total 24
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 bin
drwxr-xr-x    5 root     root        4.0K Oct 19 00:19 dev
drwxr-xr-x    2 root     root        4.0K Oct 19 00:19 etc
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 lib
lrwxrwxrwx    1 root     root           3 Mar 12  2013 lib64 -> lib
dr-xr-xr-x  116 root     root           0 Nov 15 23:34 proc
lrwxrwxrwx    1 root     root           3 Mar 12  2013 sbin -> bin
dr-xr-xr-x   13 root     root           0 Nov 15 23:34 sys
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 tmp
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 usr
 ---> b35f4035db3f
Step 3 : CMD echo Hello world
 ---> Running in 02071fceb21b
 ---> f52f38b7823e
```

Обратите внимание, что `---> Running in 02071fceb21b` выход, вы можете войти в эти изображения:

```
docker run -it 02071fceb21b bash
```

Передача stdin в контейнер

В таких случаях, как восстановление дампа базы данных или иное желание отправить некоторую информацию через канал с хоста, вы можете использовать флаг `-i` в качестве

аргумента для `docker run` `docker exec` или `docker exec` .

Например, если вы хотите поставить контейнеризованному клиенту mariadb дамп базы данных, который у вас есть на хосте, в локальном файле `dump.sql` вы можете выполнить следующую команду:

```
docker exec -i mariadb bash -c 'mariadb "-p$MARIADB_PASSWORD" ' < dump.sql
```

В общем,

```
docker exec -i container command < file.stdin
```

Или же

```
docker exec -i container command <<EOF
inline-document-from-host-shell-HEREDOC-syntax
EOF
```

Отсоединение от контейнера

При подключении к работающему контейнеру с назначенным pty (`docker run -it ...`), вы можете нажать `Control P` - `Control Q`, чтобы отсоединиться.

Переопределение директивы точки входа изображения

```
docker run --name="test-app" --entrypoint="/bin/bash" example-app
```

Эта команда будет переопределять директиву `ENTRYPOINT` образа `example-app` при создании `test-app` контейнера. Директива `CMD` изображения останется неизменной, если не указано иное:

```
docker run --name="test-app" --entrypoint="/bin/bash" example-app /app/test.sh
```

В приведенном выше примере как `ENTRYPOINT` и `CMD` изображения были переопределены. Этот процесс контейнера становится `/bin/bash /app/test.sh` .

Добавить запись узла в контейнер

```
docker run --add-host="app-backend:10.15.1.24" awesome-app
```

Эта команда добавляет запись в файл `/etc/hosts` контейнера, который следует за форматом `--add-host <name>:<address>` . В этом примере имя `app-backend` будет `10.15.1.24` до `10.15.1.24` . Это особенно полезно для связывания разрозненных компонентов приложения вместе программным путем.

Запретить остановке контейнера, если команды не запущены

Контейнер остановится, если на переднем плане не выполняется команда. Использование опции `-t` предотвратит остановку контейнера даже при отсоединении с опцией `-d` .

```
docker run -t -d debian bash
```

Остановка контейнера

```
docker stop mynginx
```

Кроме того, идентификатор контейнера также можно использовать для остановки контейнера вместо его имени.

Это остановит запуск контейнера, посылая сигнал SIGTERM, а затем сигнал SIGKILL, если это необходимо.

Кроме того, команда kill может быть использована для немедленной отправки SIGKILL или любого другого заданного сигнала с использованием опции `-s` .

```
docker kill mynginx
```

Указанный сигнал:

```
docker kill -s SIGINT mynginx
```

Остановка контейнера не удаляет его. Используйте `docker ps -a` чтобы увидеть ваш остановленный контейнер.

Выполните еще одну команду в запущенном контейнере

При необходимости вы можете сообщить Docker о выполнении дополнительных команд в уже запущенном контейнере с помощью команды `exec` . Вам нужен идентификатор контейнера, который вы можете получить с помощью `docker ps` .

```
docker exec 294fbc4c24b3 echo "Hello World"
```

Вы можете приложить интерактивную оболочку, если вы используете опцию `-it` .

```
docker exec -it 294fbc4c24b3 bash
```

Запуск приложений GUI в контейнере Linux

По умолчанию контейнер Docker не сможет *запускать* приложение GUI.

До этого гнездо X11 должно быть отправлено первым в контейнер, поэтому его можно

использовать напрямую. Переменная среды *DISPLAY также* должна быть переадресована:

```
docker run -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=unix$DISPLAY <image-name>
```

Сначала это не сработает, поскольку мы не установили разрешения для хоста X-сервера:

```
cannot connect to X server unix:0
```

Самый быстрый (но не самый безопасный) способ - разрешить доступ напрямую:

```
xhost +local:root
```

После окончания работы с контейнером мы можем вернуться в исходное состояние с:

```
xhost -local:root
```

Другой (более безопасный) способ - подготовить Dockerfile, который будет создавать новое изображение, которое будет использовать наши учетные данные для доступа к X-серверу:

```
FROM <iamge-name>
MAINTAINER <you>

# Arguments picked from the command line!
ARG user
ARG uid
ARG gid

#Add new user with our credentials
ENV USERNAME ${user}
RUN useradd -m $USERNAME && \
        echo "$USERNAME:$USERNAME" | chpasswd && \
        usermod --shell /bin/bash $USERNAME && \
        usermod  --uid ${uid} $USERNAME && \
        groupmod --gid ${gid} $USERNAME

USER ${user}

WORKDIR /home/${user}
```

Когда вы вызываете `docker build` из командной строки, мы должны передавать переменные *ARG,* которые появляются в файле Docker:

```
docker build --build-arg user=$USER --build-arg uid=$(id -u) --build-arg gid=$(id -g) -t <new-image-with-X11-enabled-name> -f <Dockerfile-for-X11> .
```

Теперь перед созданием нового контейнера нам нужно создать файл xauth с правами доступа:

```
xauth nlist $DISPLAY | sed -e 's/^..../ffff/' | xauth -f /tmp/.docker.xauth nmerge -
```

Этот файл должен быть установлен в контейнер при его создании / запуске:

```
docker run -e DISPLAY=unix$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -v
/tmp/.docker.xauth:/tmp/.docker.xauth:rw -e XAUTHORITY=/tmp/.docker.xauth
```

Прочитайте Запуск контейнеров онлайн: https://riptutorial.com/ru/docker/topic/679/запуск-контейнеров

глава 14: Запуск приложения Simple Node.js

Examples

Запуск приложения Basic Node.js внутри контейнера

Пример, который я собираюсь обсудить, предполагает, что у вас есть установка Docker, которая работает в вашей системе, и основное понимание того, как работать с Node.js. Если вам известно о том, как вы должны работать с Docker, должно быть очевидно, что в вашей системе не нужно устанавливать инфраструктуру Node.js, скорее мы будем использовать самую `latest` версию изображения `node` доступную из Docker. Следовательно, при необходимости вы можете загрузить изображение заранее с помощью `docker pull node` выгрузки команды `docker pull node` . (Команда автоматически `pulls` последнюю версию изображения `node` из докера).

1. Продолжайте создавать каталог, в котором будут находиться все ваши рабочие файлы приложений. Создайте файл `package.json` в этом каталоге, который описывает ваше приложение, а также зависимости. Ваш файл `package.json` должен выглядеть примерно так:

```
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "First Last <first.last@example.com>",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.13.3"
  }
}
```

2. Если нам нужно работать с Node.js, мы обычно создаем файл `server` который определяет веб-приложение. В этом случае мы используем среду `Express.js` (версия `4.13.3` далее). Основной файл `server.js` будет выглядеть примерно так:

```
var express = require('express');
var PORT = 8080;
var app = express();
app.get('/', function (req, res) {
  res.send('Hello world\n');
});

app.listen(PORT);
```

```
console.log('Running on http://localhost:' + PORT);
```

3. Для тех, кто знаком с Docker, вы столкнулись с `Dockerfile` . `Dockerfile` - это текстовый файл, содержащий все команды, необходимые для создания пользовательского образа, специально разработанного для вашего приложения.

Создайте пустой текстовый файл с именем `Dockerfile` в текущем каталоге. Способ создания одного из них прост в Windows. В Linux вы можете выполнить `touch Dockerfile` в каталоге, содержащем все файлы, необходимые для вашего приложения. Откройте Dockerfile с любым текстовым редактором и добавьте следующие строки:

```
FROM node:latest
RUN mkdir -p /usr/src/my_first_app
WORKDIR /usr/src/my_first_app
COPY package.json /usr/src/my_first_app/
RUN npm install
COPY . /usr/src/my_first_app
EXPOSE 8080
```

- `FROM node:latest` указывает демона Docker, с какого изображения мы хотим построить. В этом случае мы используем `latest` версию официального `node` изображения Docker, доступную в Docker Hub .
- Внутри этого изображения мы перейдем к созданию рабочего каталога, который содержит все необходимые файлы, и мы даем указание демонам установить этот каталог как желаемый рабочий каталог для нашего приложения. Для этого добавим

  ```
  RUN mkdir -p /usr/src/my_first_app
  WORKDIR /usr/src/my_first_app
  ```

- Затем мы переходим к установке зависимостей приложений, сначала перемещая файл `package.json` (который указывает информацию о приложении, включая зависимости) в рабочий каталог `/usr/src/my_first_app` на изображении. Мы это делаем

  ```
  COPY package.json /usr/src/my_first_app/
  RUN npm install
  ```

- Затем мы `COPY . /usr/src/my_first_app` чтобы добавить все файлы приложений и исходный код в рабочий каталог на изображении.
- Затем мы используем директиву `EXPOSE` чтобы дать указание демонам отобразить порт `8080` отображаемого контейнера (через сопоставление между контейнером и узлом), так как приложение привязывается к порту `8080` .
- На последнем шаге мы инструктируем демона запустить командный `node server.js` внутри изображения, выполнив основную команду `npm start` . Для этого мы используем директиву `CMD` , которая принимает команды как аргументы.

```
CMD [ "npm", "start" ]
```

4. Затем мы создаем файл `.dockerignore` в том же каталоге, что и `Dockerfile` чтобы наша копия `node_modules` и журналов, используемых нашей системой установки Node.js, не копировалась на образ Docker. Файл `.dockerignore` должен иметь следующий контент:

```
node_modules
npm-debug.log
```

5. # Создайте свой образ

Перейдите в каталог, содержащий `Dockerfile` и выполните следующую команду для создания образа Docker. Флаг `-t` позволяет пометить ваше изображение, чтобы его легче было найти позже, используя команду изображений докеров:

```
$ docker build -t <your username>/node-web-app .
```

Теперь ваше изображение будет указано в Docker. Просмотр изображений с помощью команды:

```
$ docker images

REPOSITORY                        TAG         ID              CREATED
node                              latest      539c0211cd76    10 minutes ago
<your username>/node-web-app      latest      d64d3505b0d2    1 minute ago
```

6. # Запуск изображения

Теперь мы можем запустить созданный образ, используя содержимое приложения, базовое изображение `node` и файл `Dockerfile` . Теперь мы перейдем к созданию нового созданного образа `<your username>/node-web-app` . Предоставление ключа `-d` для команды `docker run` запускает контейнер в отдельном режиме, так что контейнер работает в фоновом режиме. Флаг `-p` перенаправляет открытый порт в частный порт внутри контейнера. Запустите изображение, которое вы ранее создали с помощью этой команды:

```
$ docker run -p 49160:8080 -d <your username>/node-web-app
```

7. Распечатайте вывод своего приложения, запустив `docker ps` на вашем терминале. Результат должен выглядеть примерно так.

```
 CONTAINER ID        IMAGE                          COMMAND             CREATED
STATUS              PORTS                      NAMES
 7b701693b294      <your username>/node-web-app    "npm start"         20 minutes ago
```

```
Up 48 seconds       0.0.0.0:49160->8080/tcp   loving_goldstine
```

Получить выход приложения, введя `docker logs <CONTAINER ID>` . В этом случае это `docker logs 7b701693b294` .

Выход: `Running on http://localhost:8080`

8. Из вывода `docker ps` получается полученное сопоставление портов `0.0.0.0:49160->8080/tcp` . Следовательно, Docker сопоставил порт `8080` внутри контейнера с портом 49160 на главной машине. В браузере теперь мы можем ввести `localhost:49160` .

Мы также можем вызвать наше приложение с помощью `curl` :

```
$ curl -i localhost:49160

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
Date: Sun, 08 Jan 2017 14:00:12 GMT
Connection: keep-alive

Hello world
```

Прочитайте Запуск приложения Simple Node.js онлайн:
https://riptutorial.com/ru/docker/topic/8754/запуск-приложения-simple-node-js

глава 15: Запуск услуг

Examples

Создание более продвинутого сервиса

В следующем примере мы создадим сервис с *визуализатором* имени. Мы укажем пользовательский ярлык и перенастроим внутренний порт службы с 8080 по 9090. Кроме того, мы свяжем монтирование внешнего каталога хоста в службе.

```
docker service create \
        --name=visualizer \
        --label com.my.custom.label=visualizer \
        --publish=9090:8080 \
        --mount type=bind,source=/var/run/docker.sock,target=/var/run/docker.sock \
        manomarks/visualizer:latest
```

Создание простого сервиса

Этот простой пример создаст приветственный веб-сервис, который будет прослушивать порт 80.

```
docker service create \
    --publish 80:80 \
    tutum/hello-world
```

Удаление услуги

Этот простой пример удалит службу с именем «визуализатор»:

```
docker service rm visualizer
```

Масштабирование службы

Этот пример будет масштабировать службу до 4 экземпляров:

```
docker service scale visualizer=4
```

В режиме Docker Swarm мы не останавливаем службу. Мы масштабируем его до нуля:

```
docker service scale visualizer=0
```

Прочитайте Запуск услуг онлайн: https://riptutorial.com/ru/docker/topic/8802/запуск-услуг

глава 16: Как настроить три узла Mongo Replica с использованием изображения Docker и с помощью шеф-повара

Вступление

В этой документации описывается, как создать набор реплик с тремя узлами Mongo с использованием Docker Image и автоматически подготовлен с использованием шеф-повара.

Examples

Шаг сборки

шаги:

1. Создайте базовый файл Base 64 для аутентификации узлов Mongo. Поместите этот файл в chef data_bags
2. Пойдите в супермаркет шеф-повара и загрузите кулинарную книгу докеров. Создайте пользовательскую поваренную книгу (например, custom_mongo) и добавьте в «cookie» метаданные вашей cookbook файл cookie «docker», «~> 2.0»
3. Создайте атрибуты и рецепт в своей пользовательской кулинарной книге
4. Инициализация Mongo для создания кластера Rep Set

Шаг 1. Создание файла ключа

создайте data_bag, называемый mongo-keyfile, и элемент, называемый keyfile. Это будет в каталоге data_bags в шеф-поваре. Содержимое элемента будет следующим:

```
openssl rand -base64 756 > <path-to-keyfile>
```

содержимое элемента ключа

```
{
  "id": "keyfile",
  "comment": "Mongo Repset keyfile",
  "key-file": "generated base 64 key above"
}
```

Шаг 2: Загрузите кулинарную книгу докеров с шеф-повара, а затем создайте поваренную книгу custom_mongo

```
knife cookbook site download docker
knife cookbook create custom_mongo
```

в metadat.rb custom_mongo добавить

```
depends          'docker', '~> 2.0'
```

Шаг 3: создать атрибут и рецепт

Атрибуты

```
default['custom_mongo']['mongo_keyfile'] = '/data/keyfile'
default['custom_mongo']['mongo_datadir'] = '/data/db'
default['custom_mongo']['mongo_datapath'] = '/data'
default['custom_mongo']['keyfilename'] = 'mongodb-keyfile'
```

Рецепт

```
#
# Cookbook Name:: custom_mongo
# Recipe:: default
#
# Copyright 2017, Innocent Anigbo
#
# All rights reserved - Do Not Redistribute
#

data_path = "#{node['custom_mongo']['mongo_datapath']}"
data_dir = "#{node['custom_mongo']['mongo_datadir']}"
key_dir = "#{node['custom_mongo']['mongo_keyfile']}"
keyfile_content = data_bag_item('mongo-keyfile', 'keyfile')
keyfile_name = "#{node['custom_mongo']['keyfilename']}"

#chown of keyfile to docker user
execute 'assign-user' do
 command "chown 999 #{key_dir}/#{keyfile_name}"
 action :nothing
end

#Declaration to create Mongo data DIR and Keyfile DIR
%W[ #{data_path} #{data_dir} #{key_dir} ].each do |path|
directory path do
  mode '0755'
  end
end

#declaration to copy keyfile from data_bag to keyfile DIR on your mongo server
file "#{key_dir}/#{keyfile_name}" do
  content keyfile_content['key-file']
  group 'root'
  mode '0400'
  notifies :run, 'execute[assign-user]', :immediately
```

```
end

#Install docker
docker_service 'default' do
  action [:create, :start]
end

#Install mongo 3.4.2
docker_image 'mongo' do
  tag '3.4.2'
  action :pull
end
```

Создать роль, называемую mongo-role в каталоге ролей

```
{
  "name": "mongo-role",
  "description": "mongo DB Role",
  "run_list": [
    "recipe[custom_mongo]"
  ]
}
```

Добавить роль выше в список запуска трех монго

```
knife node run_list add FQDN_of_node_01 'role[mongo-role]'
knife node run_list add FQDN_of_node_02 'role[mongo-role]'
knife node run_list add FQDN_of_node_03 'role[mongo-role]'
```

Шаг 4: Инициализируйте три узла Mongo, чтобы сформировать repset

Я предполагаю, что вышеупомянутая роль уже была применена ко всем трем монго-узлам. Только на узле 01, Начать Mongo с --auth, чтобы включить аутентификацию

```
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
01.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --auth
```

Получите доступ к интерактивной оболочке работающего контейнера докеров на узле 01 и Create admin user

```
docker exec -it mongo /bin/sh
    mongo
    use admin
    db.createUser( {
         user: "admin-user",
         pwd: "password",
         roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
       });
```

Создание пользователя root

```
db.createUser( {
         user: "RootAdmin",
```

```
        pwd: "password",
        roles: [ { role: "root", db: "admin" } ]
      });
```

Остановить и удалить контейнер Docker, созданный выше на узле 01. Это не повлияет на данные и ключевой файл в DIR хоста. После удаления Монго снова на узел 01, но на этот раз с флагом repset

```
docker rm -fv mongo
docker run --name mongo-uat -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --
hostname="mongo-01.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-
keyfile --replSet "rs0"
```

теперь запустите mongo на узле 02 и 03 с установленным значком rep

```
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
02.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --replSet
"rs0"
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo-
03.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --replSet
"rs0"
```

Аутентификация с пользователем root на узле 01 и инициирование набора реплик

```
use admin
db.auth("RootAdmin", "password");
rs.initiate()
```

На узле 01 добавьте узел 2 и 3 в набор реплик, чтобы сформировать кластер repset0

```
rs.add("mongo-02.example.com")
rs.add("mongo-03.example.com")
```

тестирование

В основном запуске db.printSlaveReplicationInfo () и наблюдать за SyncedTo и за основным временем. Позже должно быть 0 секунд, как показано ниже

Выход

```
rs0:PRIMARY> db.printSlaveReplicationInfo()
   source: mongo-02.example.com:27017
           syncedTo: Mon Mar 27 2017 15:01:04 GMT+0000 (UTC)
           0 secs (0 hrs) behind the primary
   source: mongo-03.example.com:27017
           syncedTo: Mon Mar 27 2017 15:01:04 GMT+0000 (UTC)
           0 secs (0 hrs) behind the primary
```

Я надеюсь, что это помогает кому-то

Прочитайте Как настроить три узла Mongo Replica с использованием изображения Docker

и с помощью шеф-повара онлайн: https://riptutorial.com/ru/docker/topic/10014/как-настроить-три-узла-mongo-replica-с-использованием-изображения-docker-и-с-помощью-шеф-повара

глава 17: Контрольно-пропускные пункты и контейнеры для восстановления

Examples

Компиляция докеров с включенной контрольной точкой и восстановлением (ubuntu)

Чтобы скомпилировать докер, рекомендуется иметь как минимум **2 ГБ оперативной памяти** . Даже с этим иногда случается так, что лучше пойти на **4 ГБ** .

1. убедитесь, что установлены git и make

```
sudo apt-get install make git-core -y
```

2. установить новое ядро (не менее 4.2)

```
sudo apt-get install linux-generic-lts-xenial
```

3. перезагрузите компьютер, чтобы активировать новое ядро

```
sudo reboot
```

4. компилировать `criu` который необходим для запуска `docker checkpoint`

```
sudo apt-get install libprotobuf-dev libprotobuf-c0-dev protobuf-c-compiler protobuf-
compiler python-protobuf libnl-3-dev libcap-dev -y
wget http://download.openvz.org/criu/criu-2.4.tar.bz2 -O - | tar -xj
cd criu-2.4
make
make install-lib
make install-criu
```

5. проверьте, выполняется ли каждое требование для запуска criu

```
sudo criu check
```

6. компилировать экспериментальный докер (нам нужен докер для компиляции докеров)

```
cd ~
wget -qO- https://get.docker.com/ | sh
sudo usermod -aG docker $(whoami)
```

- **На этом этапе нам нужно выйти из системы и снова войти в систему, чтобы иметь**

демон докеров. После повторного продолжения с шагом компиляции

```
git clone https://github.com/boucher/docker
cd docker
git checkout docker-checkpoint-restore
make #that will take some time - drink a coffee
DOCKER_EXPERIMENTAL=1 make binary
```

7. Теперь у нас есть скомпилированный докер. Позволяет переместить двоичные файлы. Обязательно замените `<version>` установленной версией

```
sudo service docker stop
sudo cp $(which docker) $(which docker)_ ; sudo cp ./bundles/latest/binary-client/docker-
<version>-dev $(which docker)
sudo cp $(which docker-containerd) $(which docker-containerd)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd $(which docker-containerd)
sudo cp $(which docker-containerd-ctr) $(which docker-containerd-ctr)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd-ctr $(which docker-containerd-ctr)
sudo cp $(which docker-containerd-shim) $(which docker-containerd-shim)_ ; sudo cp
./bundles/latest/binary-daemon/docker-containerd-shim $(which docker-containerd-shim)
sudo cp $(which dockerd) $(which dockerd)_ ; sudo cp ./bundles/latest/binary-
daemon/dockerd $(which dockerd)
sudo cp $(which docker-runc) $(which docker-runc)_ ; sudo cp ./bundles/latest/binary-
daemon/docker-runc $(which docker-runc)
sudo service docker start
```

Не беспокойтесь - мы создали резервные копии старых двоичных файлов. Они все еще там, но с подчеркиванием, добавленным к его именам (`docker_`).

Поздравляем, у вас теперь есть экспериментальный докер с возможностью проверки контейнера и его восстановления.

Обратите внимание, что экспериментальные функции НЕ готовы к производству

Контрольная точка и восстановление контейнера

```
# create docker container
export cid=$(docker run -d --security-opt seccomp:unconfined busybox /bin/sh -c 'i=0; while
true; do echo $i; i=$(expr $i + 1); sleep 1; done')

# container is started and prints a number every second
# display the output with
docker logs $cid

# checkpoint the container
docker checkpoint create $cid checkpointname

# container is not running anymore
docker np

# lets pass some time to make sure

# resume container
docker start $cid --checkpoint=checkpointname
```

```
# print logs again
docker logs $cid
```

Прочитайте Контрольно-пропускные пункты и контейнеры для восстановления онлайн: https://riptutorial.com/ru/docker/topic/5291/контрольно-пропускные-пункты-и-контейнеры-для-восстановления

глава 18: Концепция объемов докеров

замечания

Люди, не знакомые с Docker, часто не понимают, что файловые системы Docker по умолчанию являются временными. Если вы запустите изображение Docker, вы получите контейнер, который на поверхности ведет себя так же, как виртуальная машина. Вы можете создавать, изменять и удалять файлы. Однако, в отличие от виртуальной машины, если вы остановите контейнер и запустите его снова, все ваши изменения будут потеряны - все файлы, которые вы ранее удалили, теперь будут возвращены, а любые новые файлы или изменения, которые вы сделали, не будут присутствовать.

Объемы в контейнерах докеров допускают постоянные данные и обмен данными с хост-машиной внутри контейнера.

Examples

A) Запустите контейнер с объемом

```
[root@localhost ~]# docker run -it -v  /data  --name=vol3   8251da35e7a7 /bin/bash
root@d87bf9607836:/# cd /data/
root@d87bf9607836:/data# touch abc{1..10}
root@d87bf9607836:/data# ls
```

abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9

B) Теперь нажмите [cont + P + Q], чтобы выйти из контейнера, не завершая проверку контейнера для контейнера, который работает

```
[root@localhost ~]# docker ps
```

CONTAINER ID IMAGE COMMAND СОЗДАН СОСТОЯНИЯ ПОРТЫ НАЗВАНИЯ d87bf9607836 8251da35e7a7 "/ bin / bash" Около минуты назад До 31 секунды vol3 [root @ localhost ~] #

C) Запустите «docker inspect», чтобы узнать больше об объеме

```
[root@localhost ~]# docker inspect  d87bf9607836
```

«Mounts»: [{«Name»: «cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c», «Источник»: «/ var / lib / docker / volume / cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c / _data»,

«Destination»: «/ data», «Driver»: «local», «Режим»: «», «RW»: true

D) Вы можете присоединить текущий контейнерный контейнер к другим контейнерам

```
[root@localhost ~]# docker run -it  --volumes-from  vol3  8251da35e7a7  /bin/bash

root@ef2f5cc545be:/# ls
```

bin boot data dev и т. д. home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

`root@ef2f5cc545be:/# ls` / data abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9

E) Вы также можете установить базовый каталог внутри контейнера

```
[root@localhost ~]# docker run -it  -v  /etc:/etc1 8251da35e7a7 /bin/bash
```

Здесь: / etc - каталог хост-компьютера, а / etc1 - цель внутри контейнера

Прочитайте Концепция объемов докеров онлайн: https://riptutorial.com/ru/docker/topic/5908/концепция-объемов-докеров

глава 19: логирование

Examples

Настройка драйвера журнала в службе systemd

```
[Service]

# empty exec prevents error "docker.service has more than one ExecStart= setting, which is
only allowed for Type=oneshot services. Refusing."
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// --log-driver=syslog
```

Это позволяет регистрировать syslog для демона докеров. Файл должен быть создан в соответствующем каталоге с корнем владельца, который обычно будет `/etc/systemd/system/docker.service.d` например, Ubuntu 16.04.

обзор

Подход Docker к регистрации - это то, что вы создаете свои контейнеры таким образом, чтобы журналы записывались на стандартный вывод (консоль / терминал).

Если у вас уже есть контейнер, который записывает журналы в файл, вы можете перенаправить его, создав символическую ссылку:

```
ln -sf /dev/stdout /var/log/nginx/access.log
ln -sf /dev/stderr /var/log/nginx/error.log
```

После этого вы можете использовать различные драйверы журналов, чтобы разместить свои журналы там, где они вам нужны.

Прочитайте логирование онлайн: https://riptutorial.com/ru/docker/topic/7378/логирование

глава 20: Несколько процессов в одном экземпляре контейнера

замечания

Обычно в каждом контейнере должен быть один процесс. Если вам нужно несколько процессов в одном контейнере (например, SSH-сервер для входа в ваш экземпляр запущенного контейнера), вы можете получить идею написать собственный сценарий оболочки, который запускает эти процессы. В этом случае вам нужно было позаботиться об обработке `SIGNAL` самостоятельно (например, перенаправление пойманного `SIGINT` дочерним процессам вашего скрипта). Это не то, что вы хотите. Простое решение - использовать `supervisord` в качестве корневого процесса контейнеров, который заботится об обработке `SIGNAL` и о продолжительности его дочерних процессов.

Но имейте в виду, что это не «путь докеров». Чтобы достичь этого примера на докере, вы должны войти в `docker host` (машина, на которой запущен контейнер) и запустить `docker exec -it container_name /bin/bahs` . Эта команда открывает оболочку внутри контейнера, как это делает ssh.

Examples

Dockerfile + supervisord.conf

Чтобы запустить несколько процессов, например, веб-сервер Apache вместе с демоном SSH внутри одного и того же контейнера, вы можете использовать `supervisord` .

Создайте свой файл конфигурации `supervisord.conf` например:

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

Затем создайте `Dockerfile` например:

```
FROM ubuntu:16.04
RUN apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/lock/apache2 /var/run/apache2 /var/run/sshd /var/log/supervisor
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
CMD ["/usr/bin/supervisord"]
```

Затем вы можете создать свой образ:

```
docker build -t supervisord-test .
```

После этого вы можете запустить его:

```
$ docker run -p 22 -p 80 -t -i supervisord-test
2016-07-26 13:15:21,101 CRIT Supervisor running as root (no user in config file)
2016-07-26 13:15:21,101 WARN Included extra file     "/etc/supervisor/conf.d/supervisord.conf"
during parsing
2016-07-26 13:15:21,112 INFO supervisord started with pid 1
2016-07-26 13:15:21,113 INFO spawned: 'sshd' with pid 6
2016-07-26 13:15:21,115 INFO spawned: 'apache2' with pid 7
...
```

Прочитайте Несколько процессов в одном экземпляре контейнера онлайн: https://riptutorial.com/ru/docker/topic/4053/несколько-процессов-в-одном-экземпляре-контейнера

глава 21: Объемы данных докеров

Вступление

Точки данных Docker предоставляют возможность сохранять данные независимо от жизненного цикла контейнера. Объемы представлены рядом полезных функций, таких как:

Установка каталога хоста в контейнере, совместное использование данных между контейнерами с использованием файловой системы и сохранение данных, если контейнер удаляется

Синтаксис

- объем докера [ОПЦИИ] [КОМАНДА]

Examples

Установка каталога с локального хоста в контейнер

Можно установить каталог хоста на определенный путь в вашем контейнере, используя `--volume` командной строки `-v` или `--volume` . Следующий пример будет монтировать `/etc` на хосте в `/mnt/etc` в контейнере:

```
(on linux) docker run -v "/etc:/mnt/etc" alpine cat /mnt/etc/passwd
(on windows)  docker run -v "/c/etc:/mnt/etc" alpine cat /mnt/etc/passwd
```

Доступ по умолчанию к тому внутри контейнера - чтение и запись. Чтобы установить том, доступный только для чтения внутри контейнера, используйте суффикс `:ro` :

```
docker run -v "/etc:/mnt/etc:ro" alpine touch /mnt/etc/passwd
```

Создание именованного тома

```
docker volume create --name="myAwesomeApp"
```

Использование именованного тома делает управление томами гораздо более удобочитаемым для человека. Можно создать именованный том, используя указанную выше команду, но также можно создать именованный том внутри команды `docker run` с помощью опции `-v` или `--volume` командной строки:

```
docker run -d --name="myApp-1" -v="myAwesomeApp:/data/app" myApp:1.5.3
```

Обратите внимание: создание именованного тома в этой форме аналогично установке файла хоста / каталога в качестве тома, за исключением того, что вместо допустимого пути указывается имя тома. После создания названные тома могут использоваться совместно с другими контейнерами:

```
docker run -d --name="myApp-2" --volumes-from "myApp-1" myApp:1.5.3
```

После выполнения вышеуказанной команды был создан новый контейнер с именем `myApp-2` из `myApp:1.5.3` , который делится `myAwesomeApp` именем volume с помощью `myApp-1` . Мост `myAwesomeApp` именем volume устанавливается в `/data/app` в контейнере `myApp-2` , так же как он монтируется в `/data/app` в контейнере `myApp-1` .

Прочитайте Объемы данных докеров онлайн: https://riptutorial.com/ru/docker/topic/1318/объемы-данных-докеров

глава 22: Объемы данных и контейнеры данных

Examples

Контейнеры с данными

Контейнеры только для данных устарели и теперь считаются анти-шаблонами!

В дни yore, перед подкомандой `volume` Docker, и до того, как было возможно создать именованные тома, Docker удалил тома, когда больше не было ссылок на них в каких-либо контейнерах. Контейнеры, предназначенные только для данных, устарели, поскольку Docker теперь предоставляет возможность создавать именованные тома, а также гораздо больше полезности через подкоманду `docker volume` . По этой причине контейнеры только для данных теперь считаются анти-шаблонами.

Многие ресурсы в Интернете за последние пару лет упоминают использование шаблона, называемого «контейнером только для данных», который представляет собой просто контейнер Docker, который существует только для хранения ссылки на объем данных.

Помните, что в этом контексте «том данных» является томом Докера, который не монтируется с хоста. Чтобы уточнить, «том данных» - это тома, который создается либо с помощью директивы `VOLUME` Dockerfile, либо с помощью ключа `-v` в командной строке в команде `docker run` , в частности, с форматом `-v /path/on/container` . Поэтому «контейнер только для данных» представляет собой контейнер, единственная цель которого заключается в том, что он подключен к тому данных, который используется `--volumes-from` в команде `docker run` . Например:

```
docker run -d --name "mysql-data" -v "/var/lib/mysql" alpine /bin/true
```

Когда эта команда запускается, создается «контейнер только для данных». Это просто пустой контейнер с прикрепленным объемом данных. Тогда можно было использовать этот том в другом контейнере следующим образом:

```
docker run -d --name="mysql" --volumes-from="mysql-data" mysql
```

Теперь контейнер `mysql` имеет тот же объем, что и в `mysql-data` .

Поскольку Docker теперь предоставляет подкоманду `volume` и именованные тома, этот шаблон теперь устарел и не рекомендуется.

Чтобы начать работу с подкомандой `volume` и названными томами, см. Создание

именованного тома

Создание объема данных

```
docker run -d --name "mysql-1" -v "/var/lib/mysql" mysql
```

Эта команда создает новый контейнер из образа `mysql` . Он также создает новый том данных, который затем монтируется в контейнере в `/var/lib/mysql` . Этот том помогает любым данным, которые внутри него сохраняются за пределами срока службы контейнера. То есть, когда контейнер удаляется, его изменения в файловой системе также удаляются. Если база данных хранит данные в контейнере, а контейнер удален, все эти данные также удаляются. Объемы будут сохраняться в определенном месте даже за пределами, когда его контейнер будет удален.

Можно использовать один и тот же том в нескольких контейнерах с параметром `--volumes-from` командной строки:

```
docker run -d --name="mysql-2" --volumes-from="mysql-1" mysql
```

В контейнере `mysql-2` теперь есть объем данных из подключенного к нему `mysql-1` , также используя путь `/var/lib/mysql` .

Прочитайте Объемы данных и контейнеры данных онлайн:
https://riptutorial.com/ru/docker/topic/3224/объемы-данных-и-контейнеры-данных

глава 23: Ограничение доступа к сети контейнеров

замечания

Пример сетей докеров, которые блокируют трафик. Использовать в качестве сети при запуске контейнера с `--net docker network connect --net` или `docker network connect` .

Examples

Блокировать доступ к локальной сети и

```
docker network create -o "com.docker.network.bridge.enable_ip_masquerade"="false" lan-restricted
```

- Блоки
  - Локальная сеть
  - интернет
- Не блокирует
  - Хост, выполняющий демон docker (пример доступа к `10.0.1.10:22`)

Блокировать доступ к другим контейнерам

```
docker network create -o "com.docker.network.bridge.enable_icc"="false" icc-restricted
```

- Блоки
  - Контейнеры, получающие доступ к другим контейнерам в одной и той же сети с `icc-restricted` ICC.
- Не блокирует
  - Доступ к ведущему демонстранту докеров
  - Локальная сеть
  - интернет

Блокировать доступ из контейнеров на локальный хост, запускающий демон докеров

```
iptables -I INPUT -i docker0 -m addrtype --dst-type LOCAL -j DROP
```

- Блоки
  - Доступ к ведущему демонстранту докеров

- Не блокирует
  - Контейнер в контейнерный трафик
  - Локальная сеть
  - интернет
  - Пользовательские сети `docker0` которые не используют `docker0`

Блокировать доступ из контейнеров на локальный хост, выполняющий демон docker (настраиваемая сеть)

```
docker network create --subnet=192.168.0.0/24 --gateway=192.168.0.1 --ip-range=192.168.0.0/25
local-host-restricted
iptables -I INPUT -s 192.168.0.0/24 -m addrtype --dst-type LOCAL -j DROP
```

Создает сеть с именем `local-host-restricted` которая:

- Блоки
  - Доступ к ведущему демонстранту докеров
- Не блокирует
  - Контейнер в контейнерный трафик
  - Локальная сеть
  - интернет
  - Доступ, исходящий из других докерных сетей

Пользовательские сети имеют имена мостов, такие как `br-15bbe9bb5bf5` , поэтому вместо этого мы используем его подсеть.

Прочитайте Ограничение доступа к сети контейнеров онлайн:
https://riptutorial.com/ru/docker/topic/6331/ограничение-доступа-к-сети-контейнеров

глава 24: Отладка контейнера

Синтаксис

- docker stats [ОПЦИИ] [КОНТЕЙНЕР ...]
- докерные журналы [ОПЦИИ] КОНТЕЙНЕР
- докерный верх [ОПЦИИ] КОНТЕЙНЕР [ps OPTIONS]

Examples

Вход в рабочий контейнер

Для выполнения операций в контейнере используйте команду `docker exec` . Иногда это называется «входом в контейнер», поскольку все команды выполняются внутри контейнера.

```
docker exec -it container_id bash
```

или же

```
docker exec -it container_id /bin/sh
```

И теперь у вас есть оболочка в вашем запущенном контейнере. Например, перечислите файлы в каталоге и выйдите из контейнера:

```
docker exec container_id ls -la
```

Вы можете использовать `-u flag` для входа в контейнер с определенным пользователем, например `uid=1013` , `gid=1023` .

```
docker exec -it -u 1013:1023 container_id ls -la
```

Uid и gid не должны существовать в контейнере, но команда может привести к ошибкам. Если вы хотите запустить контейнер и немедленно войти внутрь, чтобы что-то проверить, вы можете сделать

```
docker run...; docker exec -it $(docker ps -lq) bash
```

команда `docker ps -lq` выводит только идентификатор последнего (контейнера l `-lq`). (это предполагает, что у вас есть bash как интерпретатор, доступный в вашем контейнере, у вас может быть sh или zsh или любой другой)

Мониторинг использования ресурсов

Проверка использования ресурсов системы - эффективный способ поиска неверных приложений. Этот пример является эквивалентом традиционной `top` команды для контейнеров:

```
docker stats
```

Чтобы следить за статистикой конкретных контейнеров, перечислите их в командной строке:

```
docker stats 7786807d8084 7786807d8085
```

Статистика Docker отображает следующую информацию:

```
CONTAINER     CPU %   MEM USAGE / LIMIT   MEM %    NET I/O               BLOCK I/O
7786807d8084  0.65%   1.33 GB / 3.95 GB   33.67%   142.2 MB / 57.79 MB   46.32 MB / 0 B
```

По умолчанию `docker stats` отображает идентификатор контейнеров, и это не очень полезно, если вы предпочитаете отображать имена контейнера, просто выполните

```
docker stats $(docker ps --format '{{.Names}}')
```

Процессы мониторинга в контейнере

Проверка использования ресурсов системы - эффективный способ сузить проблему в реальном приложении. Этот пример является эквивалентом традиционной команды `ps` для контейнеров.

```
docker top 7786807d8084
```

Чтобы отфильтровать формат вывода, добавьте опции `ps` в командной строке:

```
docker top 7786807d8084 faux
```

Или, чтобы получить список процессов, выполняемых как root, что является потенциально опасной практикой:

```
docker top 7786807d8084 -u root
```

Команда `docker top` оказывается особенно полезной при поиске минималистических контейнеров без оболочки или команды `ps` .

Прикрепите к работающему контейнеру

«Присоединение к контейнеру» - это действие запуска терминального сеанса в контексте, в котором работает контейнер (и любые его программы). Это в основном используется для

целей отладки, но может также потребоваться, если конкретные данные должны быть переданы программам, запущенным в контейнере.

Для этого используется команда `attach` . Он имеет этот синтаксис:

```
docker attach <container>
```

`<container>` может быть либо идентификатором контейнера, либо именем контейнера. Например:

```
docker attach c8a9cf1a1fa8
```

Или же:

```
docker attach graceful_hopper
```

Вы , возможно , потребуются `sudo` выше команда, в зависимости от пользователя и как докер устанавливаются.

> Примечание. Attach only позволяет одновременно подключать один сеанс оболочки к контейнеру.
>
> Предупреждение: *все* ввод клавиатуры будет отправлен в контейнер. Нажатие `Ctrl-c` *убьет* ваш контейнер.

Чтобы отсоединиться от прикрепленного контейнера, последовательно нажмите `Ctrl-p`, затем `Ctrl-q`

Чтобы подключить несколько сеансов оболочки к контейнеру или просто как альтернативу, вы можете использовать `exec` . Использование идентификатора контейнера:

```
docker exec -i -t c8a9cf1a1fa8 /bin/bash
```

Использование имени контейнера:

```
docker exec -i -t graceful_hopper /bin/bash
```

`exec` запускает программу внутри контейнера, в этом случае `/bin/bash` (оболочка, предположительно одна из которых имеет контейнер). `-i` указывает на интерактивный сеанс, в то время как `-t` выделяет псевдо-TTY.

> Примечание. В отличие от *attach* , нажатие `Ctrl-c` завершает команду *exec* 'd при запуске в интерактивном режиме.

Печать журналов

Следуя журналам, это менее навязчивый способ отладки в реальном времени приложения. В этом примере воспроизводится поведение традиционного `tail -f some-application.log` на контейнере `7786807d8084` .

```
docker logs --follow --tail 10 7786807d8084
```

Эта команда в основном показывает стандартный вывод процесса контейнера (процесс с pid 1).

Если ваши журналы не включают в себя `--timestamps` метку, вы можете добавить флаг `--timestamps` .

Можно просмотреть журналы остановленного контейнера, либо

- запустите неудачный контейнер с `docker run ... ; docker logs $(docker ps -lq)`
- найти идентификатор или имя контейнера

```
docker ps -a
```

а потом

`docker logs container-id` или

`docker logs containername`

так как можно просмотреть журналы остановленного контейнера

Отладка процесса контейнера докеров

Docker - просто причудливый способ запуска процесса, а не виртуальная машина. Поэтому отладка процесса «в контейнере» также возможна «на хосте», просто исследуя выполняемый процесс контейнера как пользователя с соответствующими разрешениями для проверки этих процессов на хосте (например, root). Например, можно указать каждый «контейнерный процесс» на хосте, запустив простой `ps` как root:

```
sudo ps aux
```

Любые текущие контейнеры Docker будут перечислены в выводе.

Это может быть полезно при разработке приложений для отладки процесса, выполняющегося в контейнере. Как пользователь с соответствующими разрешениями, типичные утилиты отладки могут использоваться в контейнерном процессе, такие как strace, ltrace, gdb и т. Д.

Прочитайте Отладка контейнера онлайн: https://riptutorial.com/ru/docker/topic/1333/отладка-контейнера

глава 25: Отладка при сбое сборки докеров

Вступление

Когда `docker build -t mytag .` с сообщением, например `---> Running in d9a42e53eb5a The command '/bin/sh -c returned a non-zero code: 127` (127 означает «команда не найдена, а 1) для всех нет 3) 127 может быть заменено на 6 или что угодно), может быть нетривиально найти ошибку в длинной строке

Examples

базовый пример

В качестве последнего слоя, созданного

```
docker build -t mytag .
```

показал

```
---> Running in d9a42e53eb5a
```

Вы просто запускаете последнее созданное изображение с помощью оболочки и запускаете команду, и у вас будет более четкое сообщение об ошибке

```
docker run -it d9a42e53eb5a /bin/bash
```

(это предполагает наличие / bin / bash, это может быть / bin / sh или что-то еще)

и с подсказкой вы запускаете последнюю команду failing и видите, что отображается

Прочитайте Отладка при сбое сборки докеров онлайн:
https://riptutorial.com/ru/docker/topic/8078/отладка-при-сбое-сборки-докеров

глава 26: передача секретных данных в запущенный контейнер

Examples

способы передачи секретов в контейнере

Не очень безопасный способ (потому что `docker inspect` будет показывать его) - передать переменную окружения в

```
docker run
```

такие как

```
docker run -e password=abc
```

или в файле

```
docker run --env-file myfile
```

где myfile может содержать

```
password1=abc password2=def
```

можно также поместить их в объем

```
docker run -v $(pwd)/my-secret-file:/secret-file
```

несколько лучших способов, используйте

keywhiz https://square.github.io/keywhiz/

vault https://www.hashicorp.com/blog/vault.html

и т. д. crypt https://xordataexchange.github.io/crypt/

Прочитайте передача секретных данных в запущенный контейнер онлайн: https://riptutorial.com/ru/docker/topic/6481/передача-секретных-данных-в-запущенный-контейнер

глава 27: Подключение контейнеров

параметры

параметр	подробности
`tty:true`	В файле docker-compose.yml флаг `tty: true` сохраняет команду `sh` контейнера, ожидающую ввода.

замечания

Драйверы сети `host` и `bridge` могут подключать контейнеры на одном докере-хосте. Чтобы контейнеры могли общаться за пределами одной машины, создайте оверлейную сеть. Шаги по созданию сети зависят от того, как управляются хосты ваших докеров.

- Режим Swarm: `docker network create --driver overlay`
- docker / swarm : требует наличия внешнего хранилища ключей

Examples

Докерная сеть

Контейнеры в одной докерной сети имеют доступ к открытым портам.

```
docker network create sample
docker run --net sample --name keys consul agent -server -client=0.0.0.0 -bootstrap
```

Консоль Dockerfile предоставляет `8500` , `8600` и еще несколько портов. Чтобы продемонстрировать, запустите другой контейнер в той же сети:

```
docker run --net sample -ti alpine sh
/ # wget -qO- keys:8500/v1/catalog/nodes
```

Здесь контейнер consul разрешен из `keys` , имя которого указано в первой команде. Докер обеспечивает разрешение DNS в этой сети, чтобы найти контейнеры по их `--name` .

Docker-Compose

Сети могут быть указаны в файле компоновки (v2). По умолчанию все контейнеры находятся в общей сети.

Начните с этого файла: `example/docker-compose.yml` :

```
version: '2'
services:
  keys:
    image: consul
    command: agent -server -client=0.0.0.0 -bootstrap
  test:
    image: alpine
    tty: true
    command: sh
```

Запуск этого стека с помощью `docker-compose up -d` приведет к созданию сети с именем после родительского каталога, в данном случае `example_default` . Проверить с помощью `docker network ls`

```
 > docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
719eafa8690b        example_default     bridge              local
```

Подключитесь к альпийскому контейнеру, чтобы убедиться, что контейнеры могут разрешать и сообщать:

```
 > docker exec -ti example_test_1 sh
/ # nslookup keys
...
/ # wget -qO- keys:8500/v1/kv/?recurse
...
```

В составном файле могут быть `networks:` раздел верхнего уровня, чтобы указать сетевое имя, драйвер и другие параметры из команды сети докеров .

Контейнерная связь

Аргумент docker `--link` и `link:` разделы docker-compose делают *псевдонимы* для других контейнеров.

```
docker network create sample
docker run -d --net sample --name redis redis
```

При ссылке либо исходное имя, либо сопоставление разрешит контейнер redis.

```
> docker run --net sample --link redis:cache -ti python:alpine sh -c "pip install redis &&
python"
>>> import redis
>>> r = redis.StrictRedis(host='cache')
>>> r.set('key', 'value')
True
```

До `1.10.0` контейнера докеры `1.10.0` также устанавливаются сетевые подключения - поведение теперь обеспечивается сетью докеров. Ссылки в более поздних версиях обеспечивают только `legacy` сети моста по умолчанию.

Прочитайте Подключение контейнеров онлайн: https://riptutorial.com/ru/docker/topic/6528/подключение-контейнеров

глава 28: Порядок размещения докеров

замечания

1. Объявление базового изображения (`FROM`)
2. Метаданные (например, `MAINTAINER` , `LABEL`)
3. Установка системных зависимостей (например, `apt-get install` , `apk add`)
4. Копирование файла зависимостей приложений (например, `bower.json` , `package.json` , `build.gradle` , `requirements.txt`)
5. Установка зависимостей приложений (например, `npm install` `pip install`)
6. Копирование всей базы кода
7. Настройка `ENTRYPOINT` времени выполнения по умолчанию (например, `CMD` , `ENTRYPOINT` , `ENV` , `EXPOSE`)

Эти заказы выполняются для оптимизации времени сборки с использованием встроенного механизма кэширования Docker.

Правило:

> Части, которые часто меняются (например, кодовая база), должны располагаться вблизи дна Dockerfile и наоборот. Части, которые редко меняются (например, зависимости), следует размещать сверху.

Examples

Простой файл докеров

```
# Base image
FROM python:2.7-alpine

# Metadata
MAINTAINER John Doe <johndoe@example.com>

# System-level dependencies
RUN apk add --update \
    ca-certificates \
    && update-ca-certificates \
    && rm -rf /var/cache/apk/*

# App dependencies
COPY requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

# App codebase
WORKDIR /app
COPY . ./

# Configs
```

```
ENV DEBUG true
EXPOSE 5000
CMD ["python", "app.py"]
```

MAINTAINER будет устаревшим в Docker 1.13 и должен быть заменен с помощью LABEL. (Источник)

Пример: LABEL Maintainer = "John Doe johndoe@example.com"

Прочитайте Порядок размещения докеров онлайн: https://riptutorial.com/ru/docker/topic/6448/порядок-размещения-докеров

глава 29: Проверка работающего контейнера

Синтаксис

- докер проверяет [ОПЦИИ] КОНТЕЙНЕР | ИЗОБРАЖЕНИЕ [КОНТЕЙНЕР | ИЗОБРАЖЕНИЕ ...]

Examples

Получить информацию о контейнере

Чтобы получить всю информацию о контейнере, вы можете запустить:

```
docker inspect <container>
```

Получить конкретную информацию из контейнера

Вы можете получить определенную информацию из контейнера, выполнив:

```
docker inspect -f '<format>' <container>
```

Например, вы можете получить настройки сети, выполнив:

```
docker inspect -f '{{ .NetworkSettings }}' <container>
```

Вы также можете получить только IP-адрес:

```
docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container>
```

Параметр -f означает формат и получит шаблон Go в качестве входных данных для форматирования ожидаемого, но это не принесет прекрасного результата, поэтому попробуйте:

```
docker inspect -f '{{ json .NetworkSettings }}' {{containerIdOrName}}
```

ключевое слово json приведет к возврату как JSON.

Итак, чтобы закончить, небольшой совет - использовать там python для форматирования вывода JSON:

```
docker inspect -f '{{ json .NetworkSettings }}' <container> | python -mjson.tool
```

И вуаля, вы можете запросить что-нибудь на докер-инспекции и сделать ее красивой в вашем терминале.

Также можно использовать утилиту под названием « jq », чтобы помочь обработать вывод команды `docker inspect` докеров .

```
docker inspect -f '{{ json .NetworkSettings }}' aa1 | jq [.Gateway]
```

Вышеуказанная команда вернет следующий результат:

```
[
  "172.17.0.1"
]
```

Этот вывод фактически представляет собой список, содержащий один элемент. Иногда `docker inspect` отображает список нескольких элементов, и вы можете захотеть обратиться к определенному элементу. Например, если `Config.Env` содержит несколько элементов, вы можете обратиться к первому элементу этого списка, используя `index` :

```
docker inspect --format '{{ index (index .Config.Env) 0 }}' <container>
```

Первый элемент индексируется в нуле, что означает, что второй элемент этого списка имеет индекс `1` :

```
docker inspect --format '{{ index (index .Config.Env) 1 }}' <container>
```

Используя `len` можно получить количество элементов списка:

```
docker inspect --format `{{ len .Config.Env }}` <container>
```

И используя отрицательные числа, можно обратиться к последнему элементу списка:

```
docker inspect -format "{{ index .Config.Cmd $[$(docker inspect -format `{{ len .Config.Cmd
}}` <container>)-1]}}" <container>
```

Некоторые `docker inspect` информацию, поступают в виде словаря ключа: значение, вот выдержка `docker inspect` в контейнере jess / spotify

```
"Config": { "Hostname": "8255f4804dde", "Domainname": "", "User": "spotify", "AttachStdin":
false, "AttachStdout": false, "AttachStderr": false, "Tty": false, "OpenStdin": false,
"StdinOnce": false, "Env": [ "DISPLAY=unix:0",
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin", "HOME=/home/spotify" ],
"Cmd": [ "-stylesheet=/home/spotify/spotify-override.css" ], "Image": "jess/spotify", "Volumes":
null, "WorkingDir": "/home/spotify", "Entrypoint": [ "spotify" ], "OnBuild": null, "Labels": {}
},
```

поэтому я получаю значения всего раздела Config

```
docker inspect -f '{{.Config}}' 825

{8255f4804dde spotify false false false map[] false false false [DISPLAY=unix:0
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin HOME=/home/spotify] [-
stylesheet=/home/spotify/spotify-override.css] false jess/spotify map[] /home/spotify [spotify]
false [] map[] }
```

но также и одно поле, например значение Config.Image

```
docker inspect -f '{{index (.Config) "Image" }}' 825

jess/spotify
```

или Config.Cmd

```
docker inspect -f '{{.Config.Cmd}}' 825

[-stylesheet=/home/spotify/spotify-override.css]
```

Проверьте изображение

Чтобы проверить изображение, вы можете использовать идентификатор изображения или имя изображения, состоящее из репозитория и тега. Скажем, у вас есть базовое изображение CentOS 6:

```
➜ ~ docker images
REPOSITORY        TAG             IMAGE ID        CREATED         SIZE
centos            centos6         cf2c3ece5e41    2 weeks ago     194.6 MB
```

В этом случае вы можете выполнить одно из следующих действий:

- ➜ ~ docker inspect cf2c3ece5e41
- ➜ ~ docker inspect centos:centos6

Обе эти команды предоставят вам всю информацию, доступную в массиве JSON:

```
[
    {
        "Id": "sha256:cf2c3ece5e418fd063bfad5e7e8d083182195152f90aac3a5ca4dbfbf6a1fc2a",
        "RepoTags": [
            "centos:centos6"
        ],
        "RepoDigests": [],
        "Parent": "",
        "Comment": "",
        "Created": "2016-07-01T22:34:39.970264448Z",
        "Container": "b355fe9a01a8f95072e4406763138c5ad9ca0a50dbb0ce07387ba905817d6702",
        "ContainerConfig": {
            "Hostname": "68a1f3cfce80",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
```

```
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) CMD [\"/bin/bash\"]"
            ],
            "Image":
"sha256:cdbcc7980b002dc19b4d5b6ac450993c478927f673339b4e6893647fe2158fa7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "build-date": "20160701",
                "license": "GPLv2",
                "name": "CentOS Base Image",
                "vendor": "CentOS"
            }
        },
        "DockerVersion": "1.10.3",
        "Author": "https://github.com/CentOS/sig-cloud-instance-images",
        "Config": {
            "Hostname": "68a1f3cfce80",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/bash"
            ],
            "Image":
"sha256:cdbcc7980b002dc19b4d5b6ac450993c478927f673339b4e6893647fe2158fa7",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "build-date": "20160701",
                "license": "GPLv2",
                "name": "CentOS Base Image",
                "vendor": "CentOS"
            }
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 194606575,
        "VirtualSize": 194606575,
        "GraphDriver": {
            "Name": "aufs",
```

```
            "Data": null
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:2714f4a6cdee9d4c987fef019608a4f61f1cda7ccf423aeb8d7d89f745c58b18"
            ]
        }
    }
]
```

Печать информации

docker `inspect` поддерживает шаблоны Go с помощью опции `--format` . Это позволяет лучше интегрировать скрипты, не прибегая к традиционным инструментам pipe / sed / grep.

Распечатайте внутренний контейнер контейнера :

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' 7786807d8084
```

Это полезно для прямого сетевого доступа к автоматической настройке балансировщика нагрузки.

Печать контейнера *init* PID :

```
docker inspect --format '{{ .State.Pid }}' 7786807d8084
```

Это полезно для более глубокого контроля через `/proc` или таких инструментов, как `strace` .

Расширенное форматирование :

```
docker inspect --format 'Container {{ .Name }} listens on {{ .NetworkSettings.IPAddress }}:{{
range $index, $elem := .Config.ExposedPorts }}{{ $index }}{{ end }}' 5765847de886 7786807d8084
```

Вывод:

```
Container /redis listens on 172.17.0.3:6379/tcp
Container /api listens on 172.17.0.2:4000/tcp
```

Отладка журналов контейнеров с помощью проверки докеров

команда `docker inspect` может использоваться для отладки журналов контейнера.

Stdout и stderr контейнера можно проверить для отладки контейнера, местоположение которого можно получить с помощью `docker inspect` .

Команда: `docker inspect <container-id> | grep Source`

Он дает расположение контейнеров stdout и stderr.

Изучение stdout / stderr работающего контейнера

```
docker logs --follow <containerid>
```

Это приводит к выходу из работающего контейнера. Это полезно, если вы не настроили драйвер регистрации на демке docker.

Прочитайте Проверка работающего контейнера онлайн:
https://riptutorial.com/ru/docker/topic/1336/проверка-работающего-контейнера

глава 30: Реестр докеров

Examples

Запуск реестра

Не используйте `registry:latest` ! Это изображение указывает на старый реестр v1. Этот проект Python больше не разрабатывается. Новый реестр v2 написан на Go и активно поддерживается. Когда люди ссылаются на «частный реестр», они ссылаются на реестр v2, а *не* на реестр v1!

```
docker run -d -p 5000:5000 --name="registry" registry:2
```

Вышеупомянутая команда запускает новейшую версию реестра, которая может быть найдена в проекте Docker Distribution .

Дополнительные примеры функций управления изображениями, такие как тегирование, вытягивание или нажатие, см. В разделе об управлении изображениями.

Конфигурирование реестра с помощью сервера хранения AWS S3

Настройка частного реестра для использования бэкэнда AWS S3 проста. Реестр может сделать это автоматически с правильной конфигурацией. Вот пример того, что должно быть в вашем файле `config.yml` :

```
storage:
    s3:
        accesskey: AKAAAAAACCCCCCCBBBDA
        secretkey: rn9rjnNuX44iK+26qpM4cDEoOnonbBW98FYaiDtS
        region: us-east-1
        bucket: registry.example.com
        encrypt: false
        secure: true
        v4auth: true
        chunksize: 5242880
        rootdirectory: /registry
```

`secretkey` `accesskey` и `secretkey` являются учетными данными IAM с определенными разрешениями S3 (дополнительную информацию см. В документации). Он также может легко использовать учетные данные с `AmazonS3FullAccess` политикой `AmazonS3FullAccess` . `region` является областью вашего сегмента S3. `bucket` - это имя ковша. Вы можете выбрать для хранения изображений, зашифрованных с помощью `encrypt` . `secure` поле должно указывать на использование HTTPS. Обычно вы должны установить `v4auth` в true, хотя его значение по умолчанию - false. Поле `chunksize` позволяет вам соблюдать требования S3 API, что размер загружаемых файлов не менее 5 мегабайт. Наконец, `rootdirectory` указывает

каталог под вашим веером S3 для использования.

Существуют и другие серверы хранения, которые можно настроить так же легко.

Прочитайте Реестр докеров онлайн: https://riptutorial.com/ru/docker/topic/4173/реестр-докеров

глава 31: Реестр закрытого / безопасного Docker с API v2

Вступление

Частный и безопасный реестр докеров, а не Docker Hub. Требуются базовые навыки докеров.

параметры

команда	объяснение
sudo docker run -p 5000: 5000	Запустите контейнер докеров и привяжите порт 5000 от контейнера к порту 5000 физической машины.
Реестр имен	Название контейнера (лучше использовать для чтения «докер ps»).
-v 'pwd' / certs: / certs	Bind CURRENT_DIR / certs физической машины на / certs контейнера (например, «общая папка»).
-e REGISTRY_HTTP_TLS_CERTIFICATE = / certs / server.crt	Мы указываем, что реестр должен использовать файл /certs/server.crt для запуска. (переменная env)
-e REGISTRY_HTTP_TLS_KEY = / certs / server.key	То же самое для ключа RSA (server.key).
-v / root / images: / var / lib / registry /	Если вы хотите сохранить все изображения реестра, вы должны сделать это на физической машине. Здесь мы сохраняем все изображения на / root / images на физической машине. Если вы это сделаете, вы можете остановить и перезапустить реестр без потери изображений.
реестр: 2	Мы указываем, что мы хотели бы вытащить изображение реестра из узла докеров (или локально), и мы добавим «2», потому что мы хотим установить версию 2 реестра.

замечания

Как установить docker-engine (называемый клиентом в этом учебнике)

Как создать самоподписанный сертификат SSL

Examples

Создание сертификатов

Создайте закрытый ключ RSA: `openssl genrsa -des3 -out server.key 4096`

На этом этапе Openssl должен запросить пропущенную фразу. Обратите внимание, что мы будем использовать только сертификат для связи и аутентификации без пропущенной фразы. Например, используйте 123456.

Сгенерировать запрос `openssl req -new -key server.key -out server.csr` **сертификата:** `openssl req -new -key server.key -out server.csr`

Этот шаг важен, потому что вас попросят получить некоторую информацию о сертификатах. Наиболее важной информацией является «Общее имя», которое является доменным именем, которое используется для связи между частным реестром докеров и всей другой машиной. Пример: mydomain.com

Удаление фразы из закрытого ключа RSA: `cp server.key server.key.org && openssl rsa -in server.key.org -out server.key`

Как я уже сказал, мы сосредоточимся на сертификате без фразы. Поэтому будьте осторожны со всеми файлами вашего ключа (.key, .csr, .crt) и держите их в надежном месте.

Создайте самозаверяющий сертификат: `openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt`

Теперь у вас есть два важных файла: *server.key* и *server.crt* , которые необходимы для проверки подлинности частного реестра.

Запуск реестра с самозаверяющим сертификатом

Чтобы запустить частный реестр (безопасно), вы должны создать самозаверяющий сертификат, вы можете обратиться к предыдущему примеру для его создания.

В моем примере я помещал *server.key* и *server.crt* в / root / certs

Перед запуском команды docker вы должны поместить (используйте `cd`) в каталог, содержащий папку *certs* . Если вы этого не сделаете, и вы попытаетесь запустить команду,

вы получите сообщение об ошибке

> level = fatal msg = "open /certs/server.crt: нет такого файла или каталога"

Когда вы находитесь (`cd /root` в моем примере), вы можете в основном запустить защищенный / закрытый реестр, используя: `sudo docker run -p 5000:5000 --restart=always --name registry -v `pwd`/certs:/certs -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/server.crt -e REGISTRY_HTTP_TLS_KEY=/certs/server.key -v /root/Documents:/var/lib/registry/ registry:2`
Объяснения о команде доступны в разделе «Параметры».

Потяните или нажмите от клиента докера

Когда вы запускаете рабочий реестр, вы можете тянуть или нажимать на него изображения. Для этого вам нужен файл *server.crt* в специальную папку на вашем docker-клиенте. Сертификат позволяет выполнить аутентификацию с помощью реестра, а затем шифровать связь.

Скопируйте *server.crt* из реестра в файл /etc/docker/certs.d/mydomain.com:5000/ на клиентской машине. Затем переименуйте его в *ca-certificates.crt* : `mv /etc/docker/certs.d/mydomain.com:5000/server.crt /etc/docker/certs.d/mydomain.com:5000/ca-certificates.crt`

На этом этапе вы можете извлекать или перемещать изображения из своего частного реестра:
PULL: `docker pull mydomain.com:5000/nginx` или
ОТ СЕБЯ :

1. Получите официальное изображение с hub.docker.com: `docker pull nginx`
2. Отметьте это изображение перед тем, как `docker tag IMAGE_ID mydomain.com:5000/nginx` в частный реестр: `docker tag IMAGE_ID mydomain.com:5000/nginx` (используйте `docker images` докеров, чтобы получить IMAGE_ID)
3. Вставьте изображение в реестр: `docker push mydomain.com:5000/nginx`

Прочитайте Реестр закрытого / безопасного Docker с API v2 онлайн:
https://riptutorial.com/ru/docker/topic/8707/реестр-закрытого---безопасного-docker-с-api-v2

глава 32: Режим рокировки докеров

Вступление

Рой - это ряд двигателей докеров (или *узлов*), которые развертывают *службы в* совокупности. Рой используется для распространения обработки на многих физических, виртуальных или облачных машинах.

Синтаксис

- Инициализировать рой : старинный старинный рой [ОПЦИИ]
- Присоединиться к рою как к узлу и / или к менеджеру : присоединиться к докерам [OPTIONS] HOST: PORT
- Создайте новый сервис : docker service создайте [ОПЦИИ] ИЗОБРАЖЕНИЕ [КОМАНДА] [ARG ...]
- Отобразите подробную информацию об одной или нескольких услугах : осмотр службы докеров [ОПЦИИ] СЕРВИС [СЕРВИС ...]
- Список услуг : docker service ls [ОПЦИИ]
- Удалите одну или несколько служб : docker service rm SERVICE [SERVICE ...]
- Масштабировать один или несколько реплицированных сервисов : шкала обслуживания докеров SERVICE = REPLICAS [SERVICE = REPLICAS ...]
- Перечислите задачи одной или нескольких служб : docker service ps [ОПЦИИ] СЕРВИС [СЕРВИС ...]
- Обновление службы : обновление службы докеров [ОПЦИИ] СЕРВИС

замечания

Режим Swarm реализует следующие функции:

- Управление кластерами, интегрированное с Docker Engine
- Децентрализованный дизайн
- Модель декларативной службы
- пересчет
- Требуемое согласование состояния
- Мульти-хост-сеть

- Обнаружение службы
- Балансировки нагрузки
- Безопасный дизайн по умолчанию
- Перемещение обновлений

Для получения дополнительной официальной документации Docker относительно посещения Swarm : обзор режима Swarm

Команды командной строки Swarm Mode

Нажмите описание команд для документации

Инициализировать рой

```
docker swarm init [OPTIONS]
```

Присоединитесь к рою в качестве узла и / или менеджера

```
docker swarm join [OPTIONS] HOST:PORT
```

Создать новую услугу

```
docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Отображение подробной информации об одной или нескольких услугах

```
docker service inspect [OPTIONS] SERVICE [SERVICE...]
```

Список услуг

```
docker service ls [OPTIONS]
```

Удалить одну или несколько служб

```
docker service rm SERVICE [SERVICE...]
```

Масштабирование одной или нескольких реплицированных служб

```
docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS...]
```

Перечислите задачи одной или нескольких служб

```
docker service ps [OPTIONS] SERVICE [SERVICE...]
```

Обновление службы

```
docker service update [OPTIONS] SERVICE
```

Examples

Создайте рой на Linux, используя докер-машину и VirtualBox

```
# Create the nodes
# In a real world scenario we would use at least 3 managers to cover the fail of one manager.
docker-machine create -d virtualbox manager
docker-machine create -d virtualbox worker1

# Create the swarm
# It is possible to define a port for the *advertise-addr* and *listen-addr*, if none is
defined the default port 2377 will be used.
docker-machine ssh manager \
    docker swarm init \
    --advertise-addr $(docker-machine ip manager)
    --listen-addr $(docker-machine ip manager)

# Extract the Tokens for joining the Swarm
# There are 2 different Tokens for joining the swarm.
MANAGER_TOKEN=$(docker-machine ssh manager docker swarm join-token manager --quiet)
WORKER_TOKEN=$(docker-machine ssh manager docker swarm join-token worker --quiet)


# Join a worker node with the worker token
docker-machine ssh worker1 \
    docker swarm join \
    --token $WORKER_TOKEN \
    --listen-addr $(docker-machine ip worker1) \
    $(docker-machine ip manager):2377
```

Узнайте, как токен рабочего и менеджера

При автоматизации подготовки новых узлов к рою вам нужно знать, что означает правильный токен присоединения для роя, а также объявленный адрес менеджера. Вы можете найти это, выполнив следующие команды на любом из существующих узлов менеджера:

```
# grab the ipaddress:port of the manager (second last line minus the whitespace)
export MANAGER_ADDRESS=$(docker swarm join-token worker | tail -n 2 |  tr -d '[[:space:]]')

# grab the manager and worker token
export MANAGER_TOKEN=$(docker swarm join-token manager -q)
export WORKER_TOKEN=$(docker swarm join-token worker -q)
```

Опция -q выводит только токен. Без этого варианта вы получите полную команду для регистрации на рой.

Затем на вновь созданных узлах вы можете присоединиться к рою.

```
docker swarm join --token $WORKER_TOKEN $MANAGER_ADDRESS
```

Привет, мир

Обычно вы хотите создать стек служб для создания реплицированного и организованного приложения.

Типичное современное веб-приложение состоит из базы данных, api, frontend и обратного прокси.

Упорство

База данных нуждается в постоянстве, поэтому нам нужна некоторая файловая система, которая разделяется между всеми узлами роя. Это могут быть NAS, сервер NFS, GFS2 или что-то еще. Настройка здесь не входит в сферу применения. В настоящее время Docker не содержит и не управляет упорством в рое. В этом примере предполагается, что на всех узлах установлено `/nfs/` shared location.

сеть

Чтобы иметь возможность общаться друг с другом, службы в рое должны находиться в одной сети.

Выберите диапазон IP (здесь `10.0.9.0/24`) и имя сети (`hello-network`) и выполните команду:

```
docker network create \
  --driver overlay \
  --subnet 10.0.9.0/24 \
  --opt encrypted \
  hello-network
```

База данных

Первой нашей услугой является база данных. Давайте используем postgresql в качестве примера. Создайте папку для базы данных в `nfs/postgres` и запустите ее:

```
docker service create --replicas 1 --name hello-db \
       --network hello-network -e PGDATA=/var/lib/postgresql/data \
       --mount type=bind,src=/nfs/postgres,dst=/var/lib/postgresql/data \
       kiasaki/alpine-postgres:9.5
```

Обратите внимание, что мы использовали `--network hello-network` и `--mount` .

API

Создание API выходит за рамки этого примера, поэтому давайте притвориться, что у вас есть образ API под `username/hello-api` .

```
docker service create --replicas 1 --name hello-api \
       --network hello-network \
       -e NODE_ENV=production -e PORT=80 -e POSTGRESQL_HOST=hello-db \
       username/hello-api
```

Обратите внимание, что мы передали имя нашей службы базы данных. Docker swarm имеет встроенный циклический DNS-сервер, поэтому API сможет подключаться к базе данных с использованием своего DNS-имени.

Обратный прокси

Давайте создадим службу nginx для обслуживания нашего API во внешний мир. Создайте файлы конфигурации nginx в общей папке и запустите:

```
docker service create --replicas 1 --name hello-load-balancer \
       --network hello-network \
       --mount type=bind,src=/nfs/nginx/nginx.conf,dst=/etc/nginx/nginx.conf \
       -p 80:80 \
       nginx:1.10-alpine
```

Обратите внимание, что мы использовали параметр `-p` для публикации порта. Этот порт будет доступен для любого узла в рое.

Доступность узла

Доступ к режиму роя:

- Active означает, что планировщик может назначать задачи узлу.
- Пауза означает, что планировщик не назначает новые задачи для узла, но существующие задачи остаются в рабочем состоянии.
- Слив означает, что планировщик не назначает новые задачи для узла. Планировщик отключает любые существующие задачи и планирует их на доступном узле.

Чтобы изменить доступность режима:

```
#Following commands can be used on swarm manager(s)
docker node update --availability drain node-1
#to verify:
docker node ls
```

Продвигать или уничтожать узлы рой

Чтобы продвинуть узел или набор узлов, запустите `docker node promote` с узла менеджера:

```
docker node promote node-3 node-2

Node node-3 promoted to a manager in the swarm.
Node node-2 promoted to a manager in the swarm.
```

Чтобы понизить рейтинг узла или набора узлов, выполните `docker node demote` с узла-менеджера:

```
docker node demote node-3 node-2

Manager node-3 demoted in the swarm.
Manager node-2 demoted in the swarm.
```

Оставив Рой

Рабочий узел:

```
#Run the following on the worker node to leave the swarm.

docker swarm leave
Node left the swarm.
```

Если узел имеет роль « *Менеджер*» , вы получите предупреждение о сохранении кворума менеджеров. Вы можете использовать --force для выхода на узел менеджера:

```
#Manager Node

docker swarm leave --force
Node left the swarm.
```

Узлы, оставившие рой, по-прежнему будут отображаться на выходе `docker node ls` .

Чтобы удалить узлы из списка:

```
docker node rm node-2

node-2
```

Прочитайте Режим рокировки докеров онлайн: https://riptutorial.com/ru/docker/topic/749/режим-рокировки-докеров

глава 33: События докеров

Examples

Запустить контейнер и получить уведомление о связанных событиях

Документация для `docker events` содержит подробные сведения, но при ее отладке может быть полезно запустить контейнер и незамедлительно получить уведомление о любом связанном с ним событии:

```
docker run... & docker events --filter 'container=$(docker ps -lq)'
```

В `docker ps -lq` `l` обозначает `last` , а `q` `quiet` . Это удаляет `id` последнего запущенного контейнера и немедленно создает уведомление, если контейнер умирает или имеет другое событие.

Прочитайте События докеров онлайн: https://riptutorial.com/ru/docker/topic/6200/события-докеров

глава 34: Создание изображений

параметры

параметр	подробности
--вытащить	Убедитесь, что базовое изображение (`FROM`) обновлено до создания остальной части файла Docker.

Examples

Создание изображения из файла Docker

После того, как у вас есть файл Docker, вы можете создать образ из него с помощью `docker build` . Основная форма этой команды:

```
docker build -t image-name path
```

Если ваш файл Docker не называется `Dockerfile` , вы можете использовать флаг `-f` чтобы указать имя файла Dockerfile для сборки.

```
docker build -t image-name -f Dockerfile2 .
```

Например, чтобы создать образ с именем `dockerbuild-example:1.0.0` из `Dockerfile` в текущем рабочем каталоге:

```
$ ls
Dockerfile Dockerfile2

$ docker build -t dockerbuild-example:1.0.0 .

$ docker build -t dockerbuild-example-2:1.0.0 -f Dockerfile2 .
```

Дополнительную информацию о дополнительных параметрах и настройках см. В документации по использованию `docker build` докеров .

Общей ошибкой является создание файла Docker в домашнем каталоге пользователя (`~`). Это плохая идея, потому что во время `docker build -t mytag .` это сообщение появится в течение длительного времени:

> Загрузка контекста

Причиной является демон docker, пытающийся скопировать все файлы пользователя (как домашний каталог, так и его подкаталоги). Избегайте этого, всегда указывая каталог для

файла Docker.

Добавление файла `.dockerignore` в каталог сборки является хорошей практикой . Его синтаксис подобен файлам `.gitignore` и гарантирует, что только `.gitignore` файлы и каталоги будут загружены в качестве контекста сборки.

Простой файл Dockerfile

```
FROM node:5
```

Директива `FROM` указывает изображение, с которого нужно начинать. Можно использовать любую допустимую ссылку на изображение .

```
WORKDIR /usr/src/app
```

Директива `WORKDIR` устанавливает текущий рабочий каталог внутри контейнера, что эквивалентно запуску `cd` внутри контейнера. (Примечание: `RUN cd` *не* изменит текущий рабочий каталог.)

```
RUN npm install cowsay knock-knock-jokes
```

`RUN` выполняет заданную команду внутри контейнера.

```
COPY cowsay-knockknock.js ./
```

`COPY` копирует файл или каталог, указанные в первом аргументе из контекста сборки (*`path`* переданный `docker build` *`path`*) к местоположению в контейнере, указанном вторым аргументом.

```
CMD node cowsay-knockknock.js
```

`CMD` указывает команду для выполнения, когда изображение выполняется, и команда не указана. Его можно переопределить, передав команду на `docker run` .

Есть много других инструкций и опций; см. ссылку Dockerfile для полного списка.

Разница между ENTRYPOINT и CMD

Существует две директивы `Dockerfile` для указания, какую команду запускать по умолчанию в построенных изображениях. Если вы укажете только `CMD` то докер выполнит эту команду, используя стандартную `ENTRYPOINT` , которая является `/bin/sh -c` . При запуске встроенного образа вы можете переопределить любую или оба точки входа и / или команды. Если вы укажете оба, то `ENTRYPOINT` указывает исполняемый файл вашего контейнерного процесса, а `CMD` будет предоставлен в качестве параметров этого исполняемого файла.

Например, если ваш `Dockerfile` содержит

```
FROM ubuntu:16.04
CMD ["/bin/date"]
```

Затем вы используете директиву `ENTRYPOINT` по умолчанию `/bin/sh -c` и run `/bin/date` с этой точкой входа по умолчанию. Команда вашего процесса контейнера будет `/bin/sh -c /bin/date` . После запуска этого изображения оно будет по умолчанию распечатывать текущую дату

```
$ docker build -t test .
$ docker run test
Tue Jul 19 10:37:43 UTC 2016
```

Вы можете переопределить `CMD` в командной строке, и в этом случае он выполнит указанную вами команду.

```
$ docker run test /bin/hostname
bf0274ec8820
```

Если вы укажете директиву `ENTRYPOINT` , Docker будет использовать этот исполняемый файл, а директива `CMD` задает параметры (параметры) по умолчанию. Поэтому, если ваш `Dockerfile` содержит:

```
FROM ubuntu:16.04
ENTRYPOINT ["/bin/echo"]
CMD ["Hello"]
```

Тогда запуск будет производить

```
$ docker build -t test .
$ docker run test
Hello
```

Вы можете предоставить различные параметры, если хотите, но все они будут запускать `/bin/echo`

```
$ docker run test Hi
Hi
```

Если вы хотите переопределить точку входа, указанную в вашем файле Docker (т. Е. Если вы хотите запустить другую команду, чем `echo` в этом контейнере), вам нужно указать параметр `--entrypoint` в командной строке:

```
$ docker run --entrypoint=/bin/hostname test
b2c70e74df18
```

Обычно вы используете директиву `ENTRYPOINT` чтобы указать на основное приложение,

которое вы хотите запустить, и `CMD` на параметры по умолчанию.

Отображение порта в файле докеров

```
EXPOSE <port> [<port>...]
```

Из документации Докера:

> Команда `EXPOSE` информирует Docker о том, что контейнер прослушивает указанные сетевые порты во время выполнения. `EXPOSE` не делает порты контейнера доступными для хоста. Для этого вы должны использовать флаг `-p` для публикации диапазона портов или флага `-P` для публикации всех открытых портов. Вы можете открыть один номер порта и опубликовать его извне под другим номером.

Пример:

Внутри вашего файла Docker:

```
EXPOSE 8765
```

Чтобы получить доступ к этому порту с главного компьютера, включите этот аргумент в команду `docker run` :

```
-p 8765:8765
```

ENTRYPOINT и CMD рассматриваются как глагол и параметр

Предположим, что у вас есть файл Dockerfile, заканчивающийся

```
ENTRYPOINT [ "nethogs"] CMD ["wlan0"]
```

если вы построите это изображение с помощью

```
docker built -t inspector .
```

запустите изображение, построенное с помощью такого файла Docker, с помощью команды, например

```
docker run -it --net=host --rm inspector
```

, nethogs будет контролировать интерфейс wlan0

Теперь, если вы хотите контролировать интерфейс eth0 (или wlan1, или ra1 ...), вы будете делать что-то вроде

```
docker run -it --net=host --rm inspector eth0
```

или же

```
docker run -it --net=host --rm inspector wlan1
```

Нажатие и вытягивание изображения в концентратор докеров или другой реестр

Локально созданные изображения могут быть перенесены в Docker Hub или на любой другой реко-сервер докеры, известный как реестр. Используйте `docker login` в `docker login` для входа в существующий аккаунт концентратора докеров.

```
docker login

Login with your Docker ID to push and pull images from Docker Hub.
If you don't have a Docker ID, head over to https://hub.docker.com to create one.

Username: cjsimon
Password:
Login Succeeded
```

Указание имени сервера можно использовать в другом реестре докеров. Это также работает для частных или самостоятельных реестров. Кроме того, возможно использование внешнего хранилища учетных данных для обеспечения безопасности.

```
docker login quay.io
```

Затем вы можете пометить и вставить изображения в реестр, в который вы вошли. Ваш репозиторий должен быть указан как `server/username/reponame:tag` Опускание сервера в настоящее время по умолчанию - Docker Hub. (Реестр по умолчанию не может быть изменен другому провайдеру, и нет никаких планов по реализации этой функции.)

```
docker tag mynginx quay.io/cjsimon/mynginx:latest
```

Различные теги могут использоваться для представления разных версий или ветвей одного и того же изображения. Изображение с несколькими разными тегами отображает каждый тег в одном и том же репо.

Используйте `docker images` чтобы просмотреть список установленных изображений, установленных на вашем локальном компьютере, включая ваше недавно помеченное изображение. Затем используйте push, чтобы загрузить его в реестр и потянуть, чтобы загрузить изображение.

```
docker push quay.io/cjsimon/mynginx:latest
```

Все метки изображений можно вытащить, указав параметр `-a`

```
docker pull quay.io/cjsimon/mynginx:latest
```

Создание с использованием прокси

Часто при создании изображения Docker Dockerfile содержит инструкции, которые запускают программы для извлечения ресурсов из Интернета (например, `wget` чтобы вытащить двоичную сборку программы на GitHub).

Можно поручить Docker передать заданные переменные окружения, чтобы такие программы выполняли эти выборки через прокси:

```
$ docker build --build-arg http_proxy=http://myproxy.example.com:3128 \
               --build-arg https_proxy=http://myproxy.example.com:3128 \
               --build-arg no_proxy=internal.example.com \
               -t test .
```

`build-arg` - переменные окружения, доступные только во время сборки.

Прочитайте Создание изображений онлайн: https://riptutorial.com/ru/docker/topic/713/создание-изображений

глава 35: Создание службы с сохранением

Синтаксис

- docker volume create --name <volume_name> # Создает том с именем <имя_домена>
- docker run -v <volume_name>: <mount_point> -d crramirez / limesurvey: последний # Установите том <volume_name> в каталог <mount_point> в контейнере

параметры

параметр	подробности
--name <volume_name>	Укажите имя тома, которое будет создано
-v <имя_события>: < mount_point>	Укажите, где именованный том будет установлен в контейнере

замечания

Стойкость создается в контейнерах докеров с использованием томов. Докер имеет много способов справиться с объемами. Именованные объемы очень удобны:

- Они сохраняются даже при удалении контейнера с использованием опции -v.
- Единственный способ удаления именованного тома - явный вызов объема докеры rm
- Названные тома могут быть разделены между контейнером без ссылки или --volumes-from.
- У них нет проблем с разрешениями, на которых установлены смонтированные тома.
- Их можно манипулировать с помощью команды громкости докеров.

Examples

Настойчивость с именованными томами

Стойкость создается в контейнерах докеров с использованием томов. Давайте создадим контейнер Limesurvey и сохраним базу данных, загрузили содержимое и конфигурацию в именованном томе:

```
docker volume create --name mysql
docker volume create --name upload

docker run -d --name limesurvey -v mysql:/var/lib/mysql -v upload:/app/upload -p 80:80
crramirez/limesurvey:latest
```

Резервное копирование именованного тома

Нам нужно создать контейнер для монтирования тома. Затем архивируйте его и загрузите архив на наш хост.

Давайте сначала создадим объем данных с некоторыми данными:

```
docker volume create --name=data
echo "Hello World" |  docker run -i --rm=true -v data:/data ubuntu:trusty tee /data/hello.txt
```

Давайте сделаем резервную копию данных:

```
docker run -d --name backup -v data:/data ubuntu:trusty tar -czvf /tmp/data.tgz /data
docker cp backup:/tmp/data.tgz data.tgz
docker rm -fv backup
```

Давайте проверим:

```
tar -xzvf data.tgz
cat data/hello.txt
```

Прочитайте Создание службы с сохранением онлайн:
https://riptutorial.com/ru/docker/topic/7429/создание-службы-с-сохранением

глава 36: Управление изображениями

Синтаксис

- docker images [OPTIONS] [REPOSITORY [: TAG]]
- докер проверяет [ОПЦИИ] КОНТЕЙНЕР | ИЗОБРАЖЕНИЕ [КОНТЕЙНЕР | ИЗОБРАЖЕНИЕ ...]
- docker pull [ОПЦИИ] ИМЯ [: TAG | @DIGEST]
- docker rmi [ОПЦИИ] ИЗОБРАЖЕНИЕ [ИЗОБРАЖЕНИЕ ...]
- тег докера [OPTIONS] ИЗОБРАЖЕНИЕ [: TAG] [REGISTRYHOST /] [USERNAME /] NAME [: TAG]

Examples

Получение изображения из Docker Hub

Обычно изображения автоматически извлекаются из Docker Hub . Docker попытается вытащить любое изображение из Docker Hub, который еще не существует на хосте Docker. Например, использование `docker run ubuntu` когда изображение `ubuntu` еще не находится на хосте Docker, заставит Docker инициировать вытягивание последнего изображения `ubuntu` . Можно потянуть изображение отдельно, используя приведение `docker pull` чтобы вручную извлечь или обновить изображение с Docker Hub.

```
docker pull ubuntu
docker pull ubuntu:14.04
```

Возможны дополнительные варианты вытягивания из другого реестра изображений или вытаскивания определенной версии изображения. Указание альтернативного реестра выполняется с использованием полного имени изображения и дополнительной версии. Например, следующая команда попытается вытащить изображение `ubuntu:14.04` из `registry.example.com` registry.example.com:

```
docker pull registry.example.com/username/ubuntu:14.04
```

Отображение локально загруженных изображений

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              693bce725149        6 days ago          967 B
postgres            9.5                 0f3af79d8673        10 weeks ago        265.7 MB
postgres            latest              0f3af79d8673        10 weeks ago        265.7 MB
```

Ссылка на изображения

Команды докеров, которые принимают имя изображения, принимают четыре разных формы:

Тип	пример
Короткий идентификатор	`693bce725149`
название	`hello-world` *(по умолчанию `:latest` тег)*
Имя + тег	`hello-world:latest`
дайджест	`hello-world@sha256:e52be8ffeeb1f374f440893189cd32f44cb166650e7ab185fa7735b7dc48d619`

Примечание. Вы можете ссылаться только на изображение по его дайджесту, если это изображение было первоначально вытащено с помощью этого дайджеста. Чтобы увидеть дайджест для изображения (если он доступен), запустите `docker images --digests` .

Удаление изображений

Команда `docker rmi` используется для удаления изображений:

```
docker rmi <image name>
```

Для удаления изображения необходимо использовать полное имя изображения. Если изображение не было помечено для удаления имени реестра, его необходимо указать. Например:

```
docker rmi registry.example.com/username/myAppImage:1.3.5
```

Также можно удалить изображения по их идентификатору:

```
docker rmi 693bce725149
```

В качестве удобства можно удалить изображения по их идентификатору изображения, указав только первые несколько символов идентификатора изображения, если указанная подстрока недвусмысленная:

```
docker rmi 693
```

> **Примечание.** Изображения могут быть удалены, даже если существуют существующие контейнеры, которые используют это изображение; docker rmi просто «разворачивает» изображение.

Если никакие контейнеры не используют изображение, это сбор мусора. Если контейнер

использует изображение, изображение будет собираться с мусором, как только все контейнеры, использующие его, будут удалены. Например:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
PORTS               NAMES
5483657ee07b        hello-world         "/hello"            Less than a second ago   Exited
(0) 2 seconds ago                       small_elion

$ docker rmi hello-world
Untagged: hello-world:latest

$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED                  STATUS
PORTS               NAMES
5483657ee07b        693bce725149        "/hello"            Less than a second ago   Exited
(0) 12 seconds ago                       small_elion
```

Удалить все изображения без начальных контейнеров

Чтобы удалить все локальные изображения без начальных контейнеров, вы можете предоставить список изображений в качестве параметра:

```
docker rmi $(docker images -qa)
```

Удалить все изображения

Если вы хотите удалить изображения, независимо от того, есть ли у них запущенный контейнер, используйте флаг силы (`-f`):

```
docker rmi -f $(docker images -qa)
```

Удаление оборванных изображений

Если изображение не помечено и не используется каким-либо контейнером, оно «оборвано» и может быть удалено следующим образом:

```
docker images -q --no-trunc -f dangling=true | xargs -r docker rmi
```

Поиск в Docker Hub для изображений

Вы можете найти Docker Hub для изображений с помощью команды поиска :

```
docker search <term>
```

Например:

```
$ docker search nginx
NAME                      DESCRIPTION                                     STARS     OFFICIAL
```

```
AUTOMATED
nginx                     Official build of Nginx.                        3565      [OK]
jwilder/nginx-proxy       Automated Nginx reverse proxy for docker c...   717
[OK]
richarvey/nginx-php-fpm   Container running Nginx + PHP-FPM capable ...   232
[OK]
...
```

Проверка изображений

```
docker inspect <image>
```

Выходной сигнал находится в формате JSON. Вы можете использовать `jq` командной строки `jq` для анализа и печати только нужных ключей.

```
docker inspect <image> | jq -r '.[0].Author'
```

В приведенной выше команде будет отображаться имя автора изображений.

Маркировка изображений

Пометка изображения полезна для отслеживания различных версий изображений:

```
docker tag ubuntu:latest registry.example.com/username/ubuntu:latest
```

Другой пример тегирования:

```
docker tag myApp:1.4.2 myApp:latest
docker tag myApp:1.4.2 registry.example.com/company/myApp:1.4.2
```

Сохранение и загрузка изображений докеров

```
docker save -o ubuntu.latest.tar ubuntu:latest
```

Эта команда сохранит `ubuntu:latest` изображение в архиве tarball в текущем каталоге с именем `ubuntu.latest.tar` . Затем этот архив tarball можно перенести на другой хост, например, с помощью `rsync` или архивировать в хранилище.

После перемещения tarball следующая команда создаст изображение из файла:

```
docker load -i /tmp/ubuntu.latest.tar
```

Теперь можно создать контейнеры из `ubuntu:latest` изображение, как обычно.

Прочитайте Управление изображениями онлайн: https://riptutorial.com/ru/docker/topic/690/управление-изображениями

глава 37: Управление контейнерами

Синтаксис

- docker rm [ОПЦИИ] КОНТЕЙНЕР [КОНТЕЙНЕР ...]
- прикрепление докеров [ОПЦИИ] КОНТЕЙНЕР
- docker exec [OPTIONS] CONTAINER COMMAND [ARG ...]
- docker ps [ОПЦИИ]
- докерные журналы [ОПЦИИ] КОНТЕЙНЕР
- докер проверяет [ОПЦИИ] КОНТЕЙНЕР | ИЗОБРАЖЕНИЕ [КОНТЕЙНЕР | ИЗОБРАЖЕНИЕ ...]

замечания

- В приведенных выше примерах каждый раз, когда контейнер является параметром команды docker, он упоминается как `<container>` или `container id` или `<CONTAINER_NAME>` . Во всех этих местах вы можете либо передать имя контейнера, либо идентификатор контейнера, чтобы указать контейнер.

Examples

Листинг контейнеров

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
2bc9b1988080        redis               "docker-entrypoint.sh"   2 weeks ago         Up 2
hours          0.0.0.0:6379->6379/tcp    elephant-redis
817879be2230        postgres            "/docker-entrypoint.s"   2 weeks ago         Up 2
hours          0.0.0.0:65432->5432/tcp   pt-postgres
```

`docker ps` по себе печатает только текущие контейнеры. Чтобы просмотреть все контейнеры (включая остановленные), используйте флаг `-a` :

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
9cc69f11a0f7        docker/whalesay     "ls /"                   26 hours ago        Exited
(0) 26 hours ago                            berserk_wozniak
2bc9b1988080        redis               "docker-entrypoint.sh"   2 weeks ago         Up 2
hours           0.0.0.0:6379->6379/tcp    elephant-redis
817879be2230        postgres            "/docker-entrypoint.s"   2 weeks ago         Up 2
hours           0.0.0.0:65432->5432/tcp   pt-postgres
```

Чтобы отобразить контейнеры с определенным статусом, используйте параметр командной

строки `-f` для фильтрации результатов. Ниже приведен пример перечисления всех завершенных контейнеров:

```
$ docker ps -a -f status=exited
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                     NAMES
9cc69f11a0f7        docker/whalesay     "ls /"                   26 hours ago        Exited
(0) 26 hours ago
```

Также можно указать только идентификаторы контейнера с ключом `-q` . Это позволяет легко работать с результатом с другими утилитами Unix (такими как `grep` и `awk`):

```
$ docker ps -aq
9cc69f11a0f7
2bc9b1988080
817879be2230
```

При запуске контейнера с `docker run --name mycontainer1` **вы** `docker run --name mycontainer1` конкретное имя, а не произвольное имя (в форме mood_famous, например nostalgic_stallman), и их легко найти с такой командой

```
docker ps -f name=mycontainer1
```

Ссылка на контейнеры

Команды докеров, которые принимают имя контейнера, принимают три разных формы:

Тип	пример
Полный UUID	`9cc69f11a0f76073e87f25cb6eaf0e079fbfbd1bc47c063bcd25ed3722a8cc4a`
Короткая UUID	`9cc69f11a0f7`
название	`berserk_wozniak`

Используйте `docker ps` для просмотра этих значений для контейнеров в вашей системе.

UUID генерируется Docker и не может быть изменен. Вы можете указать имя контейнера при запуске `docker run --name <given name> <image>` . Docker генерирует случайное имя в контейнер, если вы не укажете его во время запуска контейнера.

***ПРИМЕЧАНИЕ** . Значение UUID (или «короткого» UUID) может быть любой длины, пока данное значение уникально для одного контейнера*

Запуск и остановка контейнеров

Чтобы остановить запущенный контейнер:

```
docker stop <container> [<container>...]
```

Это отправит основной процесс в контейнер SIGTERM, а затем SIGKILL, если он не остановится в течение льготного периода. Название каждого контейнера печатается по мере его прекращения.

Чтобы запустить контейнер, который остановлен:

```
docker start <container> [<container>...]
```

Это запустит каждый контейнер, переданный в фоновом режиме; имя каждого контейнера печатается при его запуске. Чтобы запустить контейнер на переднем плане, передайте флаг `-a` (`--attach`).

Список контейнеров с пользовательским форматом

```
docker ps --format 'table {{.ID}}\t{{.Names}}\t{{.Status}}'
```

Поиск конкретного контейнера

```
docker ps --filter name=myapp_1
```

Найти контейнерный IP-адрес

Чтобы узнать IP-адрес вашего контейнера, используйте:

```
docker inspect <container id> | grep IPAddress
```

или использовать докер-инспекцию

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' ${CID}
```

Перезапуск контейнера докеров

```
docker restart <container> [<container>...]
```

Вариант **--time:** Секунды ждать остановки перед умерщвлением контейнера (по умолчанию 10)

```
docker restart <container> --time 10
```

Удаление, удаление и очистка контейнеров

`docker rm` можно использовать для удаления определенных контейнеров следующим

образом:

```
docker rm <container name or id>
```

Чтобы удалить все контейнеры, вы можете использовать это выражение:

```
docker rm $(docker ps -qa)
```

По умолчанию докер не удаляет контейнер, который запущен. Любой запущенный контейнер выдаст предупреждающее сообщение и не будет удален. Все остальные контейнеры будут удалены.

В качестве альтернативы вы можете использовать `xargs` :

```
docker ps -aq -f status=exited | xargs -r docker rm
```

Если `docker ps -aq -f status=exited` вернет список идентификаторов контейнеров контейнеров, имеющих статус «Выход».

> Внимание: все приведенные выше примеры удаляют только контейнеры «остановлены».

Чтобы удалить контейнер, независимо от того, остановлен он или нет, вы можете использовать флаг силы `-f` :

```
docker rm -f <container name or id>
```

Чтобы удалить все контейнеры, независимо от состояния:

```
docker rm -f $(docker ps -qa)
```

Если вы хотите удалить только контейнеры с `dead` статусом:

```
docker rm $(docker ps --all -q -f status=dead)
```

Если вы хотите удалить только контейнеры с `exited` статусом:

```
docker rm $(docker ps --all -q -f status=exited)
```

Это все перестановки фильтров, используемых при перечислении контейнеров .

Чтобы удалить как нежелательные контейнеры, так и оборванные изображения, использующие пространство после версии 1.3 , используйте следующее (похожее на Unix-инструмент `df`):

```
$ docker system df
```

Чтобы удалить все неиспользуемые данные:

```
$ docker system prune
```

Запустить команду на уже существующий контейнер докеров

`docker exec –it <container id> /bin/bash`

Обычно для входа в уже запущенный контейнер можно выполнить несколько быстрых тестов или посмотреть, что делает приложение. Часто это означает плохую практику использования контейнеров из-за журналов, а измененные файлы должны размещаться в томах. Этот пример позволяет нам войти в контейнер. Это предполагает, что / bin / bash доступен в контейнере, это может быть / bin / sh или что-то еще.

```
docker exec <container id> tar –czvf /tmp/backup.tgz /data
docker cp <container id>:/tmp/backup.tgz .
```

Этот пример архивирует содержимое каталога данных в tar. Затем с помощью `docker cp` вы можете получить его.

Контейнерные журналы

```
Usage:  docker logs [OPTIONS] CONTAINER

Fetch the logs of a container

  -f, --follow=false        Follow log output
  --help=false              Print usage
  --since=                  Show logs since timestamp
  -t, --timestamps=false    Show timestamps
  --tail=all                Number of lines to show from the end of the logs
```

Например:

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND                  CREATED     STATUS       PORTS
ff9716dda6cb    nginx    "nginx -g 'daemon off"  8 days ago  Up 22 hours  443/tcp,
0.0.0.0:8080->80/tcp

$ docker logs ff9716dda6cb
xx.xx.xx.xx - - [15/Jul/2016:14:03:44 +0000] "GET /index.html HTTP/1.1" 200 511
"https://google.com" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.75 Safari/537.36"
xx.xx.xx.xx - - [15/Jul/2016:14:03:44 +0000] "GET /index.html HTTP/1.1" 200 511
"https://google.com" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/50.0.2661.75 Safari/537.36"
```

Подключиться к экземпляру, запущенному как демон

Существует два способа достижения этого, первый и наиболее известный:

```
docker attach --sig-proxy=false <container>
```

Это буквально прикрепляет ваш bash к контейнеру bash, а это означает, что если у вас запущенный скрипт, вы увидите результат.

Чтобы отсоединить, просто введите: `Ctl-P Ctl-Q`

Но если вам нужен более дружеский способ и чтобы создавать новые экземпляры bash, просто выполните следующую команду:

```
docker exec -it <container> bash
```

Копирование файла из / в контейнеры

от контейнера до принимающего

```
docker cp CONTAINER_NAME:PATH_IN_CONTAINER PATH_IN_HOST
```

от хоста до контейнера

```
docker cp PATH_IN_HOST CONTAINER_NAME:PATH_IN_CONTAINER
```

Если я использую jess / transmission из

https://hub.docker.com/r/jess/transmission/builds/bsn7eqxrkzrhxazcuytbmzp/

, файлы в контейнере находятся в / передаче / загрузке

и мой текущий каталог на хосте - / home / $ USER / abc, после

```
docker cp transmission_id_or_name:/transmission/download .
```

У меня будут файлы, скопированные в

```
/home/$USER/abc/transmission/download
```

вы не можете, используя `docker cp` копировать только один файл, вы копируете дерево каталогов и файлы

Удаление, удаление и очистка докеров

Объемы докеров не удаляются автоматически, когда контейнер остановлен. Чтобы удалить связанные тома при остановке контейнера:

```
docker rm -v <container id or name>
```

Если флаг `-v` не указан, то объем остается на диске как «оборванный том». Чтобы удалить

все оборванные тома:

```
docker volume rm $(docker volume ls -qf dangling=true)
```

Объем `docker volume ls -qf dangling=true` filter возвращает список имен `docker volume ls -qf dangling=true` , включая немаркированные, которые не привязаны к контейнеру.

Кроме того, вы можете использовать `xargs` :

```
docker volume ls -f dangling=true -q | xargs --no-run-if-empty docker volume rm
```

Экспорт и импорт файловых систем Docker container

Можно сохранить содержимое файловой системы контейнера Docker в архив архива tarball. Это полезно в том, что касается перемещения файловых систем контейнеров на разные хосты, например, если в контейнере базы данных есть важные изменения, и в противном случае невозможно повторить эти изменения в другом месте. **Обратите внимание,** что предпочтительно создавать совершенно новый контейнер из обновленного изображения с помощью команды `docker run docker-compose.yml` или файла `docker-compose.yml` вместо экспорта и перемещения файловой системы контейнера. Частью мощности Docker является проверяемость и отчетность его декларативного стиля создания изображений и контейнеров. Используя `docker import docker export` и `docker import` , эта мощность подавляется из-за обфускации изменений, сделанных внутри файловой системы контейнера из исходного состояния.

```
docker export -o redis.tar redis
```

Вышеупомянутая команда создаст пустое изображение, а затем экспортирует файловую `redis` контейнера `redis` в это пустое изображение. Чтобы импортировать из архива tarball, используйте:

```
docker import ./redis.tar redis-imported:3.0.7
```

Эта команда создаст изображение `redis-imported:3.0.7` , из которого могут быть созданы контейнеры. Также возможно создавать изменения при импорте, а также устанавливать сообщение фиксации:

```
docker import -c="ENV DEBUG true" -m="enable debug mode" ./redis.tar redis-changed
```

Директивы Dockerfile, доступные для использования с параметром командной строки `-c` - `CMD` , `ENTRYPOINT` , `ENV` , `EXPOSE` , `ONBUILD` , `USER` , `VOLUME` , `WORKDIR` .

Прочитайте Управление контейнерами онлайн: https://riptutorial.com/ru/docker/topic/689/управление-контейнерами

кредиты

S. No	Главы	Contributors
1	Начало работы с Docker	abaracedo, Aminadav, Braiam, Carlos Rafael Ramirez, Community, ganesshkumar, HankCa, Josha Inglis, L0j1k, mohan08p, Nathaniel Ford, schumacherj, Siddharth Srinivasan, SztupY, Vishrant
2	API-интерфейс Docker Engine	Ashish Bista, atv, BMitch, L0j1k, Radoslav Stoyanov, SztupY
3	docker проверяет получение различных полей для ключа: значение и элементы списка	user2915097
4	Docker фиксирует все запущенные контейнеры	Kostiantyn Rybnikov
5	Dockerfiles	BMitch, foraidt, k0pernikus, kubanczyk, L0j1k, ob1, Ohmen, rosysnake, satsumas, Stephen Leppik, Thiago Almeida, Wassim Dhif, yadutaf
6	Iptables with Docker	Adrien Ferrand
7	бегущий консул в докере 1.12 рой	Jilles van Gurp
8	безопасность	user2915097
9	Докер в Докере	Ohmen
10	Докерная машина	Amine24h, kubanczyk, Nik Rahmel, user2915097, yadutaf
11	Докерная сеть	HankCa, L0j1k, Nathaniel Ford
12	Докеры - сетевые режимы (мост, hots, сопоставленный контейнер и ни	mohan08p

	один).	
13	Запуск контейнеров	abaracedo, Adri C.S., AlcaDotS, atv, Binary Nerd, BMitch, Camilo Silva, Carlos Rafael Ramirez, cizixs, cjsimon, Claudiu, ElMesa, Emil Burzo, enderland, Felipe Plets, ganesshkumar, Gergely Fehérvári, ISanych, L0j1k, Nathan Arthur, Patrick Auld, RoyB, ssice, SztupY, Thomasleveil, tommyyards, VanagaS, Wolfgang, zinking
14	Запуск приложения Simple Node.js	Siddharth Srinivasan
15	Запуск услуг	Mateusz Mrozewski, Philip
16	Как настроить три узла Mongo Replica с использованием изображения Docker и с помощью шеф-повара	Innocent Anigbo
17	Контрольно-пропускные пункты и контейнеры для восстановления	Bastian, Fuzzyma
18	Концепция объемов докеров	Amit Poonia, Rob Bednark, serieznyi
19	логирование	Jilles van Gurp, Vanuan
20	Несколько процессов в одном экземпляре контейнера	h3nrik, Ohmen, Xavier Nicollet
21	Объемы данных докеров	James Hewitt, L0j1k, NRKirby, Nuno Curado, Scott Coates, t3h2mas
22	Объемы данных и контейнеры данных	GameScripting, L0j1k, melihovv
23	Ограничение доступа к сети контейнеров	xeor
24	Отладка	allprog, Binary Nerd, foraidt, L0j1k, Nathaniel Ford,

	контейнера	user2915097, yadutaf
25	Отладка при сбое сборки докеров	user2915097
26	передача секретных данных в запущенный контейнер	user2915097
27	Подключение контейнеров	Jett Jones
28	Порядок размещения докеров	akhyar, Philip
29	Проверка работающего контейнера	AlcaDotS, devopskata, Felipe Plets, h3nrik, Jilles van Gurp, L0j1k, Milind Chawre, Nik Rahmel, Stephen Leppik, user2915097, yadutaf
30	Реестр докеров	Ashish Bista, L0j1k
31	Реестр закрытого / безопасного Docker с API v2	bastien enjalbert, kubanczyk
32	Режим рокировки докеров	abronan, Christian, Farhad Farahi, Jilles van Gurp, kstromeiraos , kubanczyk, ob1, Philip, Vanuan
33	События докеров	Nathaniel Ford, user2915097
34	Создание изображений	cjsimon, ETL, Ken Cochrane, L0j1k, Nathan Arthur, Nathaniel Ford, Nour Chawich, SztupY, user2915097, Wolfgang
35	Создание службы с сохранением	Carlos Rafael Ramirez, Vanuan
36	Управление изображениями	akhyar, Björn Enochsson, dsw88, L0j1k, Nathan Arthur, Nathaniel Ford, Szymon Biliński, user2915097, Wolfgang, zygimantus
37	Управление контейнерами	akhyar, atv, Binary Nerd, BrunoLM, Carlos Rafael Ramirez, Emil Burzo, Felipe Plets, ganesshkumar, L0j1k, Matt, Nathaniel Ford, Rafal Wiliński, Sachin Malhotra, serieznyi, sk8terboi87 ツ, tommyyards, user2915097, Victor Oliveira Antonino, Wolfgang, Xavier Nicollet, zygimantus