

docker

zouyapeng

Published
with GitBook



目錄

介绍	0
Docker基本概念	1
Docker镜像	1.1
Docker容器	1.2
Docker仓库	1.3
Docker安装	2
Ubuntu 16.04安装	2.1
Ubuntu 14.04安装	2.2
CentOS 7安装	2.3
镜像	3
获取	3.1
列出	3.2
创建	3.3
导出导入	3.4
移除	3.5
容器	4
启动	4.1
终止	4.2
导入导出	4.3
进入	4.4
删除	4.5
仓库	5
私有仓库	5.1
数据管理	6
数据卷	6.1
数据卷容器	6.2
备份、恢复、迁移	6.3

网络	7
外部访问	7.1
容器互联	7.2

我的**Docker**学习笔记

本书只是记录我Docker的学习记录，文中内容大多来至于[Docker——从入门到实践](#)一书

Docker 基本概念

Docker 包括三个基本概念

- 镜像(Image)
- 容器(Container)
- 仓库(Repository)

理解了这三个概念,就理解了 Docker 的整个生命周期。

Docker镜像

Docker 镜像（Image）就是一个只读的模板。

例如：一个镜像可以包含一个完整的 ubuntu 操作系统环境，里面仅安装了 Apache 或用户需要的其它应用程序。镜像可以用来创建 Docker 容器。

Docker 提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。

Docker容器

Docker 利用容器（Container）来运行应用。

容器是从镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。

可以把容器看做是一个简易版的 Linux 环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

*注：镜像是只读的，容器在启动的时候创建一层可写层作为最上层。

Docker仓库

仓库（Repository）是集中存放镜像文件的场所。有时候会把仓库和仓库注册服务器（Registry）混为一谈，并不严格区分。实际上，仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。

仓库分为公开仓库（Public）和私有仓库（Private）两种形式。

最大的公开仓库是 Docker Hub，存放了数量庞大的镜像供用户下载。国内的公开仓库包括 Docker Pool、灵雀云等，可以提供大陆用户更稳定快速的访问。

当然，用户也可以在本地网络内创建一个私有仓库（参考本文“私有仓库”部分）。

当用户创建了自己的镜像之后就可以使用 `push` 命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时候，只需要从仓库上 `pull` 下来就可以了。

*注：Docker 仓库的概念跟 Git 类似，注册服务器可以理解为 GitHub 这样的托管服务。

Docker安装

官方网站上有各种环境下的[安装指南](#),这里主要介绍下Ubuntu和CentOS系列的安装。

Ubuntu 16.04安装

准备

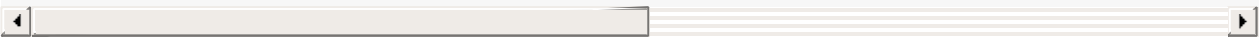
- Docker需要64位的操作系统
- Linux kernel 版本不得低于3.10

跟新源

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

添加GPG key

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
```



检查docker.list

```
$ sudo vim /etc/apt/sources.list.d/docker.list
```

修改内容：

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

跟新docker源

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

安装

- 安装docker

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

- 启动服务

```
$ sudo service docker start
```

- 测试

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

run without sudo

- 添加当前用户到docker组

```
$ sudo groupadd docker  
$ sudo gpasswd -a ${USER} docker  
$ sudo service docker restart
```

- 注销当前用户，重新登陆

Ubuntu 14.04安装

准备

- Docker需要64位的操作系统
- Linux kernel 版本不得低于3.10

跟新源

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

添加GPG key

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
```

检查docker.list

```
$ sudo vim /etc/apt/sources.list.d/docker.list
```

修改内容：

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

跟新docker源

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

安装

- 安装docker

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

- 启动服务

```
$ sudo service docker start
```

- 测试

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

run without sudo

- 添加当前用户到docker组

```
$ sudo groupadd docker  
$ sudo gpasswd -a ${USER} docker  
$ sudo service docker restart
```

- 注销当前用户，重新登陆

CentOS 7 安装

准备

- Docker需要64位的操作系统
- Linux kernel 版本不得低于3.10

安装

- 更新

```
$ sudo yum update
```

- 添加源

```
$ sudo tee /etc/yum.repos.d/docker.repo <<- 'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/$releasever
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

- 安装docker

```
$ sudo yum install docker-engine
```

- 启动服务

```
$ sudo service docker start
```

- 测试

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

run without sudo

- 添加当前用户到docker组

```
$ sudo groupadd docker  
$ sudo gpasswd -a ${USER} docker  
$ sudo service docker restart
```

- 注销当前用户，重新登陆

镜像

在之前的介绍中,我们知道镜像是 Docker 的三大组件之一。

Docker运行容器前需要本地存在对应的镜像,如果镜像不存在本地,Docker会从镜像仓库下载(默认是 Docker Hub 公共注册服务器中的仓库)。

本章将介绍更多关于镜像的内容,包括:

- 从仓库获取镜像;
- 管理本地主机上的镜像;
- 介绍镜像实现的基本原理。

获取

可以使用`docker pull` 命令来从仓库获取所需要的镜像。

下面的例子将从Docker Hub仓库下载一个Ubuntu 14.04操作系统的镜像。

```
$ docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu
6c953ac5d795: Pull complete
3eed5ff20a90: Pull complete
f8419ea7c1b5: Pull complete
51900bc9e720: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:7c151496aefa83d7d5faeff87493d471f86ff5673b829b0e1724
Status: Downloaded newer image for ubuntu:14.04
```

该命令实际上相当于

```
$ docker pull docker.io/ubuntu:14.04
```

即从官方仓库注册服务器获取镜像。

14.04为镜像的tag。

有时候官方仓库注册服务器下载较慢，甚至不能下载,这时可以从其他仓库下载或者翻墙。

列出

使用docker images显示本地已有的镜像。

```
zyp@work:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
ubuntu              14.04              8f1bd21bd25c       4 days
zouyapeng/dokuwiki  latest             c9bf1b678f65       2 weeks
```

在列出信息中,可以看到几个字段信息

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
来自于哪个仓库	标记	镜像唯一ID	创建时间	镜像大小

IMAGE ID唯一标识了镜像,注意到 ubuntu:14.04 和 ubuntu:trusty 具有相同的镜像ID,说明它们实际上是同一镜像。

TAG信息用来标记来自同一个仓库的不同镜像。

例如 ubuntu 仓库中有多个镜像,通过 TAG 信息来区分发行版本, 10.04、12.04、12.10、13.04、14.04等。

如果不指定具体的标记,则默认使用 latest 标记信息。

创建

创建Ubuntu基础镜像

```
$ sudo [debootstrap](https://wiki.debian.org/Debootstrap) trusty tr
$ sudo tar -C trusty -c . | docker import - trusty
sha256:2c2a1ef844faf88c0fd6e188ad68e08e42ab7732071c2a41a23e287af1e1
$ docker run trusty cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04 LTS"
```

在Docker GitHub仓库里还有创建其他基础镜像的脚本的例子:

- [BusyBox](#)
- CentOS / Scientific Linux CERN (SLC) on [Debian/Ubuntu](#) or on [CentOS/RHEL/SLC/etc.](#)
- [Debian / Ubuntu](#)

使用Dockerfile创建镜像

```
$ mkdir trusty && cd trusty
$ touch Dockerfile
$ touch run.sh
```

run.sh:

```
#!/bin/bash
if [ $# == 1 ];then
    echo $1
elif [ $# == 2 ];then
    echo $2
elif [ $# == 3 ];then
    echo $3
else
    echo '-----'
fi
```

Dockerfile:

```
FROM trusty:latest

MAINTAINER Zouyapeng<zyp19901009@163.com>

ENV \
    TEST_ENV1=123456 \
    TEST_ENV2=true \
    TEST_ENV3=/var/log

RUN apt-get update && apt-get install -y \
    apache2 \
    php5

COPY run.sh /

RUN chmod +x /run.sh

COPY . /tmp/
# ADD http://example.com/big.tar.xz /

VOLUME ["/etc/custom-config", "/opt/user"]

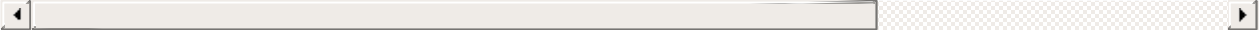
EXPOSE 80 443 162/udp 22 10080

ENTRYPOINT ["/run.sh"]
CMD ["param1", "param2", "param3"]
```

```
$ docker build -t="trusty:custom" .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM trusty:latest
----> 2c2a1ef844fa
Step 2 : MAINTAINER Zouyapeng<zyp19901009@163.com>
----> Using cache
----> 6de654f69aa5
Step 3 : ENV TEST_ENV1 123456 TEST_ENV2 true TEST_ENV3 /var/log
----> Using cache
----> 530a80f32021
Step 4 : RUN apt-get update && apt-get install -y apache2 php5
----> Using cache
----> 567c91ceeb2b
Step 5 : COPY run.sh /
----> Using cache
----> 064dbb556a6f
Step 6 : RUN chmod +x /run.sh
----> Using cache
----> 3c0398aaec8c
Step 7 : VOLUME /etc/custom-config /opt/user
----> Running in 789fed374aed
----> f900bb13e8fd
Removing intermediate container 789fed374aed
Step 8 : EXPOSE 80 443 162/udp 22 10080
----> Running in e60d416f815c
----> 0ba17f4d19b1
Removing intermediate container e60d416f815c
Step 9 : ENTRYPOINT /run.sh
----> Running in 32c7d0383e7b
----> b304a2716df7
Removing intermediate container 32c7d0383e7b
Step 10 : CMD param1 param2 param3
----> Running in b43e771e38e7
----> 2e1f283c5aa9
Removing intermediate container b43e771e38e7
Successfully built 2e1f283c5aa9
```

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATE
trusty	custom	2e1f283c5aa9	51 sec
trusty	latest	2c2a1ef844fa	About



```
$ docker run -i trusty:custom 1111
1111
$ docker run -i trusty:custom 1111 2222
2222
$ docker run -i trusty:custom 1111 2222 3333
3333
$ docker run -i trusty:custom 1111 2222 3333 4444
-----
$ docker run -i trusty:custom
param3
```

导出导入

导出

```
$ docker save -o ubuntu_14.04.tar ubuntu:14.04
```

导入

```
$ docker load --input ubuntu_14.04.tar  
$ docker load < ubuntu_14.04.tar
```


移除

使用 **docker rmi** 来移除镜像

```
$ docker rmi ubuntu:14.04
```

清理所有未打过标签的本地镜像

```
$ docker rmi $(docker images -q -f "dangling=true")  
$ docker rmi $(docker images --quiet --filter "dangling=true")
```

容器

容器是 Docker 又一核心概念。

简单的说，容器是独立运行的一个或一组应用，以及它们的运行态环境。对应的，虚拟机可以理解为模拟运行的一整套操作系统（提供了运行态环境和其他系统环境）和跑在上面的应用。

本章将具体介绍如何来管理一个容器，包括创建、启动和停止等。

er# 启动 当利用docker run 来创建容器时,Docker在后台运行的标准操作包括:

- 检查本地是否存在指定的镜像,不存在就从公有仓库下载
- 利用镜像创建并启动一个容器
- 分配一个文件系统,并在只读的镜像层外面挂载一层可读写层
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
- 从地址池配置一个ip地址给容器
- 执行用户指定的应用程序
- 执行完毕后容器被终止

hello world

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

交互模式

```
$ docker run -it ubuntu:14.04 /bin/bash  
root@57064404d74b:/#
```

```
# -t 分配一个伪终端(pseudo-tty)并绑定到容器的标准输入上  
# -i 标准输入保持打开
```

启动已终止容器

```
docker start 57064404d74b
```

后台运行

添加了-d参数

```
$ docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello  
f25d00c5b49b7601259e6f873148a924185e15921f626e12340dbfb05304354d"
```

使用docker logs 查看日志

```
$ docker logs -f dof25d00c5b49b
```

hello world

hello world

hello world

hello world

hello world

hello world

hello world

hello world

终止

可以使用 `docker stop` 来终止一个运行中的容器。

此外，当Docker容器中指定的应用终结时，容器也自动终止。例如对于上一章节中只启动了一个终端的容器，用户通过 `exit` 命令或 `Ctrl+d` 来退出终端时，所创建的容器立刻终止。

```
$ docker stop f25d00c5b49b
f25d00c5b49b
```

导入导出

导入

```
$ docker export [container id or name] > export.tar
```

导出

```
$ cat export.tar | sudo docker import - export:v1.0
```

```
$ docker import [ url ] example/imagerepo
```

进入

docker attach

在使用 `-d` 参数时，容器启动后会进入后台。某些时候需要进入容器进行操作

```
$ docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello
bf7d5922f9fe4949608ed0bdd67d2e496873580399a41d1cd21af4282aa8b104
$ docker attach bf7d5922f9fe
hello world
hello world
hello world
hello world
```



docker exec

`docker attach` 会进入容器并继续执行CMD, 类似于linux的`fg`命令

```
$ docker exec -it bf7d5922f9fe /bin/bash
```

删除

删除某个容器

```
$ docker rm [container id or name]
```

删除退出状态容器

```
$ docker rm $(docker ps -a -q)
```


仓库

仓库（Repository）是集中存放镜像的地方。

一个容易混淆的概念是注册服务器（Registry）。实际上注册服务器是管理仓库的具体服务器，每个服务器上可以有多个仓库，而每个仓库下面有多个镜像。从这方面来说，仓库可以被认为是一个具体的项目或目录。例如对于仓库地址 `dl.dockerpool.com/ubuntu` 来说，`dl.dockerpool.com` 是注册服务器地址，`ubuntu` 是仓库名。

大部分时候，并不需要严格区分这两者的概念。

私有仓库

安装

容器安装

```
# 没有国外服务器的用户建议先使用pull下镜像
$ docker pull registry

$ docker run -d -p 5000:5000 -v /opt/registry:/tmp/registry registry
```

Packages安装

- Ubuntu

```
$ sudo apt-get install -y build-essential python-dev libevent-c
$ sudo pip install docker-registry
```

- Centos

```
$ sudo yum install -y python-devel libevent-devel python-pip gc
$ sudo python-pip install docker-registry
```

其他更多查考我的wiki

[Docker私有仓库搭建\(nginx代理\)](#)

数据管理

这一章介绍如何在 Docker 内部以及容器之间管理数据，在容器中管理数据主要有两种方式：

- 数据卷（Data volumes）
- 数据卷容器（Data volume containers）

数据卷

数据卷是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

- 数据卷可以在容器之间共享和重用
- 对数据卷的修改会立马生效
- 对数据卷的更新，不会影响镜像
- 数据卷默认会一直存在，即使容器被删除

*注意：数据卷的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，能显示看的是挂载的数据卷。

创建一个数据卷

在用 `docker run` 命令的时候，使用 `-v` 标记来创建一个数据卷并挂载到容器里。在一次 run 中多次使用可以挂载多个数据卷。下面创建一个名为 `web` 的容器，并加载一个数据卷到容器的 `/webapp` 目录。

```
$ docker run -d -P --name web -v /webapp training/webapp python app
```

*注意：也可以在 `Dockerfile` 中使用 `VOLUME` 来添加一个或者多个新的卷到由该镜像创建的任意容器。

删除数据卷

数据卷是被设计用来持久化数据的，它的生命周期独立于容器，`Docker` 不会在容器被删除后自动删除数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的数据卷。如果需要在删除容器的同时移除数据卷。可以在删除容器的时候使用 `docker rm -v` 这个命令。无主的数据卷可能会占据很多空间，要清理会很麻烦。`Docker` 官方正在试图解决这个问题，相关工作的进度可以查看这个 [PR](#)

挂载一个主机目录作为数据卷

使用 `-v` 标记也可以指定挂载一个本地主机的目录到容器中去。

```
$ docker run -d -P --name web -v /src/webapp:/opt/webapp training/v
```

上面的命令加载主机的 `/src/webapp` 目录到容器的 `/opt/webapp` 目录。这个功能在进行测试的时候十分方便，比如用户可以放置一些程序到本地目录中，来查看容器是否正常工作。本地目录的路径必须是绝对路径，如果目录不存在 Docker 会自动为你创建它。

*注意：Dockerfile 中不支持这种用法，这是因为 Dockerfile 是为了移植和分享用的。然而，不同操作系统的路径格式不一样，所以目前还不能支持。

Docker 挂载数据卷的默认权限是读写，用户也可以通过 `:ro` 指定为只读。

```
$ docker run -d -P --name web -v /src/webapp:/opt/webapp:ro trainin
```

加了 `:ro` 之后，就挂载为只读了。

查看数据卷的具体信息

在主机里使用以下命令可以查看指定容器的信息

```
$ docker inspect web
```

在输出的内容中找到其中和数据卷相关的部分，可以看到所有的数据卷都是创建在主机的 `/var/lib/docker/volumes/` 下面的

```
"Volumes": {
  "/webapp": "/var/lib/docker/volumes/fac362...80535"
},
"VolumesRW": {
  "/webapp": true
}
...
```

挂载一个本地主机文件作为数据卷

-v 标记也可以从主机挂载单个文件到容器中

```
$ docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu /bin/sh
```

这样就可以记录在容器输入过的命令了。*注意：如果直接挂载一个文件，很多文件编辑工具，包括 vi 或者 sed --in-place，可能会造成文件 inode 的改变，从 Docker 1.1 .0 起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。

数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载的。首先，创建一个名为 `dbdata` 的数据卷容器：

```
$ docker run -d -v /dbdata --name dbdata training/postgres echo Data
```

然后，在其他容器中使用 `--volumes-from` 来挂载 `dbdata` 容器中的数据卷。

```
$ docker run -d --volumes-from dbdata --name db1 training/postgres  
$ docker run -d --volumes-from dbdata --name db2 training/postgres
```

可以使用超过一个的 `--volumes-from` 参数来指定从多个容器挂载不同的数据卷。也可以从其他已经挂载了数据卷的容器来级联挂载数据卷。

```
$ docker run -d --name db3 --volumes-from db1 training/postgres
```

*注意：使用 `--volumes-from` 参数所挂载数据卷的容器自己并不需要保持在运行状态。

如果删除了挂载的容器（包括 `dbdata`、`db1` 和 `db2`），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时使用 `docker rm -v` 命令来指定同时删除关联的容器。这可以让用户在容器之间升级和移动数据卷。具体的操作将在下一节中进行讲解。

备份、恢复、迁移

可以利用数据卷对其中的数据进行备份、恢复和迁移。

备份

首先使用 `--volumes-from` 标记来创建一个加载 `dbdata` 容器卷的容器，并从主机挂载当前目录到容器的 `/backup` 目录。命令如下：

```
$ docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf
```

容器启动后，使用了 `tar` 命令来将 `dbdata` 卷备份为容器中 `/backup/backup.tar` 文件，也就是主机当前目录下的名为 `backup.tar` 的文件。

恢复

如果要恢复数据到一个容器，首先创建一个带有空数据卷的容器 `dbdata2`。

```
$ docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

然后创建另一个容器，挂载 `dbdata2` 容器卷中的数据卷，并使用 `untar` 解压备份文件到挂载的容器卷中。

```
$ docker run --volumes-from dbdata2 -v $(pwd):/backup busybox tar x  
/backup/backup.tar
```

为了查看/验证恢复的数据，可以再启动一个容器挂载同样的容器卷来查看

```
$ docker run --volumes-from dbdata2 busybox /bin/ls /dbdata
```


网络

Docker 允许通过外部访问容器或容器互联的方式来提供网络服务。

外部访问

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 `-P` 或 `-p` 参数来指定端口映射。当使用 `-P` 标记时，Docker 会随机映射一个 49000~49900 的端口到内部容器开放的网络端口。使用 `docker ps` 可以看到，本地主机的 49155 被映射到了容器的 5000 端口。此时访问本机的 49155 端口即可访问容器内 web 应用提供的界面。

```
$ docker run -d -P training/webapp python app.py
$ docker ps -l
```

同样的，可以通过 `docker logs` 命令来查看应用的信息。

```
$ docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404
```

`-p`（小写的）则可以指定要映射的端口，并且，在一个指定端口上只可以绑定一个容器。支持的格式有 `ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort`。

映射所有接口地址

使用 `hostPort:containerPort` 格式本地的 5000 端口映射到容器的 5000 端口，可以执行

```
$ docker run -d -p 5000:5000 training/webapp python app.py
```

此时默认会绑定本地所有接口上的所有地址。

映射到指定地址的指定端口

可以使用 `ip:hostPort:containerPort` 格式指定映射使用一个特定地址，比如 `localhost` 地址 `127.0.0.1`

```
$ docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py
```

映射到指定地址的任意端口

使用 `ip::containerPort` 绑定 `localhost` 的任意端口到容器的 `5000` 端口，本地主机会自动分配一个端口。

```
$ sudo docker run -d -p 127.0.0.1::5000 training/webapp python app.py
```

还可以使用 `udp` 标记来指定 `udp` 端口

```
$ docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
```

查看映射端口配置

使用 `docker port` 来查看当前映射的端口配置，也可以查看到绑定的地址

```
$ docker port nostalgic_morse 5000
127.0.0.1:49155
```

注意：

容器有自己的内部网络和 `ip` 地址（使用 `docker inspect` 可以获取所有的变量，`Docker` 还可以有一个可变的网络配置。）`-p` 标记可以多次使用来绑定多个端口 例如

```
$ docker run -d -p 5000:5000 -p 3000:80 training/webapp python app.py
```

容器互联

容器的连接（linking）系统是除了端口映射外，另一种跟容器中应用交互的方式。

该系统会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

自定义容器命名

连接系统依据容器的名称来执行。因此，首先需要自定义一个好记的容器命名。

虽然当创建容器的时候，系统默认会分配一个名字。自定义命名容器有2个好处：

- 自定义的命名，比较好记，比如一个web应用容器我们可以给它起名叫web
- 当要连接其他容器时候，可以作为一个有用的参考点，比如连接web容器到db容器 使用 `--name` 标记可以为容器自定义命名。

```
$ docker run -d -P --name web training/webapp python app.py
```

使用 `docker ps` 来验证设定的命名。

```
$ docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
aed84ee21bde	training/webapp:latest	python app.py	12 hours ago	Up

也可以使用 `docker inspect` 来查看容器的名字

```
$ docker inspect -f "{{ .Name }}" aed84ee21bde
/web
```

注意：容器的名称是唯一的。如果已经命名了一个叫 `web` 的容器，当你要再次使用 `web` 这个名称的时候，需要先用 `docker rm` 来删除之前创建的同名容器。+

在执行 `docker run` 的时候如果添加 `--rm` 标记，则容器在终止后会立刻删除。注意，`--rm` 和 `-d` 参数不能同时使用。

容器互联

使用 `--link` 参数可以让容器之间安全的进行交互。下面先创建一个新的数据库容器。

```
$ docker run -d --name db training/postgres
```

删除之前创建的 web 容器

```
$ docker rm -f web
```

然后创建一个新的 web 容器，并将它连接到 db 容器

```
$ docker run -d -P --name web --link db:db training/webapp python a
```

此时，db 容器和 web 容器建立互联关系。`--link` 参数的格式为 `--link name:alias`，其中 `name` 是要链接的容器的名称，`alias` 是这个连接的别名。使用 `docker ps` 来查看容器的连接

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
349169744e49	training/postgres:latest	su postgres -c '/usr	About
aed84ee21bde	training/webapp:latest	python app.py	16 h

可以看到自定义命名的容器，db 和 web，db 容器的 `names` 列有 db 也有 web/db。这表示 web 容器链接到 db 容器，web 容器将被允许访问 db 容器的信息。

Docker 在两个互联的容器之间创建了一个安全隧道，而且不用映射它们的端口到宿主主机上。在启动 db 容器的时候并没有使用 `-p` 和 `-P` 标记，从而避免了暴露数据库端口到外部网络上。Docker 通过 2 种方式为容器公开连接信息：环境变量 更新 `/etc/hosts` 文件

使用 `env` 命令来查看 web 容器的环境变量

```
$ docker run --rm --name web2 --link db:db training/webapp env
. . .
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5000_TCP=tcp://172.17.0.5:5432
DB_PORT_5000_TCP_PROTO=tcp
DB_PORT_5000_TCP_PORT=5432
DB_PORT_5000_TCP_ADDR=172.17.0.5
. . .
```

其中 DB_ 开头的环境变量是供 web 容器连接 db 容器使用，前缀采用大写的连接别名。除了环境变量，Docker 还添加 host 信息到父容器的 /etc/hosts 的文件。下面是父容器 web 的 hosts 文件

```
$ docker run -t -i --rm --link db:db training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7 aed84ee21bde
. . .
172.17.0.5 db
```

这里有 2 个 hosts，第一个是 web 容器，web 容器用 id 作为他的主机名，第二个是 db 容器的 ip 和主机名。可以在 web 容器中安装 ping 命令来测试跟db容器的连通。

```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping db
PING db (172.17.0.5): 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

用 ping 来测试db容器，它会解析成 172.17.0.5。*注意：官方的 ubuntu 镜像默认没有安装 ping，需要自行安装。用户可以链接多个父容器到子容器，比如可以链接多个 web 到 db 容器上。