101. Docker Lab

实验环境

CentOS 7.2 系统，或者是 Ubuntu 14.04 / 16.04系统

实验大纲

1. 基本操作：
    a. 学习在 CentOS / Ubuntu 上安装 docker，并运行 hello-world
    b. 学习怎么构建一个镜像
    c. 学习怎么运行一个镜像，并查看运行状态
    d. 进一步学习对镜像打标签，并向远程仓库推送镜像

2. 高级实践：
    a. 学习在本地部署基于 python 的 web 应用

安装 Docker

CentOS 系统上安装 docker

1. RPM包更新

```
$ sudo yum update -y
```

2. 添加 yum 的 docker 仓库

注意：$后面的内容全部拷贝执行。

```
$ sudo tee /etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

3. 安装 docker 包

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

4. 打开 docker 服务

```
$ sudo systemctl enable docker.service
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service to
/usr/lib/systemd/system/docker.service.
```

5. 启动 docker 服务

```
$ sudo systemctl start docker
```

Ubuntu 系统上安装 docker

1. 包管理软件更新，并添加相关 key 文件

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

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

2. 添加 apt 的 docker repo（仓库）

Precise 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
Wily 15.10	deb https://apt.dockerproject.org/repo ubuntu-wily main
Xenial 16.04 (LTS)	deb https://apt.dockerproject.org/repo ubuntu-xenial main

注意，根据不同的 Ubuntu 版本，参考上面的表格，选择不同的 repo 内容，下面以 14.04 版本为例：

```
$ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main" | sudo tee
/etc/apt/sources.list.d/docker.list
```

3. 再次更新 apt 包管理，使得我们刚添加的 docker repo 生效

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

4. 检查 repo 添加结果

```
$ 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
```

我们发现新添加的 docker repo 已经生效，即 apt 可以找到 docker 软件包的下载地址。

5. 安装额外的内核组件，使得我们可以使用 aufs 格式的存储

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

6. 使用 apt 安装 docker

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

6. 启动 docker 服务

```
$ sudo service docker start
```

检查 docker 版本，并运行 hello-world 程序

1. 检查 docker 版本

```
$ docker version
Client:
 Version:      1.12.3
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   6b644ec
 Built:
 OS/Arch:      linux/amd64

Server:
 Version:      1.12.3
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   6b644ec
 Built:
 OS/Arch:      linux/amd64
```

上面显示了我们刚刚安装的 docker 的版本，包括客户端和服务端的版本，均为 1.12.3 版本。

2. 运行 docker hello-world

```
$ sudo docker run --rm hello-world
Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world
c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

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

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
```

```
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

这里我们使用 docker run 这个命令来运行一个容器，
--rm 的意思是当运行结束的时候，自动移除容器，
最后的 hello-world 是用来指定这次运行所使用的镜像名称，关于镜像名称定义，后面会有详细解释。

我们观察打印出来的信息：
Unable to find image 'hello-world:latest' locally，即我们在本地无法找到这个镜像，因为我们是第一次运行这个镜像，而且之前没有下载过这个镜像，
latest: Pulling from library/hello-world，表示 docker 自动从 library/hello-world 这里下载镜像，这里，由于我们没有指定从哪里下载镜像，默认的是从 docker 的官方镜像仓库下载，即 hub.docker.com 这里，
Status: Downloaded newer image for hello-world:latest，表示下载完成，

接下来，我们发现终端打印出 This message shows that your installation appears to be working correctly. 这个信息说明 docker 环境安装成功。

熟悉 docker 常用命令

1. 查看本地镜像

```
$ docker images
REPOSITORY          TAG             IMAGE ID            CREATED             SIZE
hello-world         latest          c54a2cc56cbb        4 months ago        1.848 kB
```

2. 从远程仓库拉取镜像

```
$ docker pull cargo.caicloud.io/training/ubuntu:latest
latest: Pulling from training/ubuntu
6bbedd9b76a4: Pull complete
fc19d60a83f1: Pull complete
de413bb911fd: Pull complete
2879a7ad3144: Pull complete
668604fde02e: Pull complete
Digest: sha256:2d44ae143feeb36f4c898d32ed2ab2dffeb3a573d2d8928646dfc9cb7deb1315
Status: Downloaded newer image for cargo.caicloud.io/training/ubuntu:latest
```

docker pull 命令后面指定镜像的完整名称，命名格式如下:
cargo.caicloud.io/training/ubuntu:latest，其中：
cargo.caicloud.io，代表远程仓库的域名，可以理解为仓库所在的服务器 IP 地址，
training，表示镜像所在的命名空间，可以理解为项目，
ubuntu，表示镜像的 repository，可以理解为应用程序，
latest，表示镜像的 tag，可以理解为版本号，docker 命令中，如果不指定 tag，那么默认值是 latest。

由上我们可以知道，repository + tag 可以唯一的确定一个镜像。

3. 再次查看本地镜像

```
$ docker images
REPOSITORY                          TAG       IMAGE ID       CREATED       SIZE
cargo.caicloud.io/training/ubuntu   latest    f753707788c5   5 weeks ago   127.1 MB
hello-world                         latest    c54a2cc56cbb   4 months ago  1.848 kB
```

上面显示出从远程仓库拉取的镜像信息，其中 IMAGE ID 是一个镜像的唯一标识，CREATED 表示容器构建的时间，SIZE 表示容器的大小。

4. 删除本地镜像

请注意指定要删除的镜像 ID，这里我们使用 hello-world:latest 这个镜像为例，它的镜像 ID 如上面所示，为 c54a2cc56cbb。

```
$ docker rmi c54a2cc56cbb
Untagged: hello-world:latest
Untagged:
hello-world@sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Deleted: sha256:c54a2cc56cbb2f04003c1cd4507e118af7c0d340fe7e2720f70976c4b75237dc
Deleted: sha256:a02596fdd012f22b03af6ad7d11fa590c57507558357b079c3e8cebceb4262d7
```

这里 rmi 是 remove image 的意思

3. 再次查看本地镜像

```
$ docker images
REPOSITORY                          TAG       IMAGE ID       CREATED       SIZE
cargo.caicloud.io/training/ubuntu   latest    f753707788c5   5 weeks ago   127.1 MB
```

上我们发现 hello-world:latest 这个镜像已经被删除。

构建镜像

1. 创建 Dockerfile

```
$ mkdir -p ~/mydockerfile && cd ~/mydockerfile
$ touch Dockerfile
```

2. 编辑 Dockerfile，添加下面的内容到 Dockerfile 中

```
FROM cargo.caicloud.io/training/ubuntu:latest
CMD echo "this is my first docker image"
```

Dockerfile 关键字解释：
FROM 表示这次构建基于后面指定的基础镜像，
CMD 表示当这个镜像被运行起来的时候，首先执行后面的命令。

Dockerfile 中的关键字格式是大写的，还有很多其他的关键字，后面的实验中会涉及到。

4. 检查 Dockerfile 中的内容

```
$ cat Dockerfile
FROM cargo.caicloud.io/training/ubuntu:latest
CMD echo "this is my first docker image"
```

5. 用编辑好的 Dockerfile 来构建镜像

注意：下面的命令最后是 ./，代表这次构建的目录为当前目录。

```
$ docker build -f Dockerfile ./
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM cargo.caicloud.io/training/ubuntu:latest
 ---> f753707788c5
Step 2 : CMD echo "this is my first docker image"
 ---> Running in 9b7bac1ca593
 ---> 991777d563c9
Removing intermediate container 9b7bac1ca593
Successfully built 991777d563c9
```

我们可以看到，构建的过程分为2个步骤完成，每个步骤分别是我们在 Dockerfile 中定义的内容。

6. 查看刚刚构建的镜像

```
$ docker images
REPOSITORY                          TAG       IMAGE ID       CREATED              SIZE
<none>                              <none>    991777d563c9   About a minute ago   127.1 MB
cargo.caicloud.io/training/ubuntu   latest    f753707788c5   5 weeks ago          127.1 MB
hello-world                         latest    c54a2cc56cbb   4 months ago         1.848 kB
```

我们发现有个 repository 以及 tag 均为 <none> 的镜像被构建出来，创建时间是刚刚（About a minute ago），<none> 的原因是我们构建的时候，没有指定 repository 以及 tag。

7. 重新构建这个镜像，并指定 repository 以及 tag

```
$ docker build -f Dockerfile ./ -t my-first-image:1.0
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM cargo.caicloud.io/training/ubuntu:latest
 ---> f753707788c5
Step 2 : CMD echo "this is my first docker image"
 ---> Using cache
 ---> 991777d563c9
Successfully built 991777d563c9
```

我们通过 -t 这个 option， 指定这次构建的镜像的 repository 为 my-first-image，tag 为 1.0，输出的 log 中，有 Using cache 这个信息，表示这次构建是基于缓存的，因为我们上一步已经构建过这个镜像，我们可以使用 --no-cache 这个 option 来重新构建。

8. 再次查看刚刚构建的镜像

```
$ docker images
REPOSITORY                          TAG       IMAGE ID       CREATED              SIZE
my-first-image                      1.0       991777d563c9   About a minute ago   127.1 MB
cargo.caicloud.io/training/ubuntu   latest    f753707788c5   5 weeks ago          127.1 MB
hello-world                         latest    c54a2cc56cbb   4 months ago         1.848 kB
```

我们发现 <none> 的地方被我们指定的 repository 还有 tag 替换掉，但是 image id 以及创建时间并没有变化，这说明这次构建并没有改变镜像的实质内容，下面我们加上 --no-cache 这个 option 来重新构建，观察下结果，这次我们升级到 2.0 的 tag。

7. 使用 --no-cache 重新构建这个镜像，并指定新的 tag

```
$ docker build -f Dockerfile ./ -t my-first-image:2.0 --no-cache
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM cargo.caicloud.io/training/ubuntu:latest
 ---> f753707788c5
Step 2 : CMD echo "this is my first docker image"
 ---> Running in e0e98332a96f
 ---> 097d164f0243
Removing intermediate container e0e98332a96f
Successfully built 097d164f0243
```

我们发现这次的构建没有 Using cache 这个信息输出，表面这次是执行了一次全新的构建。

9. 再次查看刚刚构建的镜像

```
$ docker images
REPOSITORY                          TAG       IMAGE ID       CREATED              SIZE
my-first-image                      2.0       097d164f0243   30 seconds ago       127.1 MB
my-first-image                      1.0       991777d563c9   About a minute ago   127.1 MB
cargo.caicloud.io/training/ubuntu   latest    f753707788c5   5 weeks ago          127.1 MB
hello-world                         latest    c54a2cc56cbb   4 months ago         1.848 kB
```

我们发现 my-first-image:2.0 的镜像被构建出来，并且 image id 以及创建时间均为新的，这说明这次构建是一次全新的构建。

运行一个镜像，并检查容器运行状态

1. 运行我们上个步骤构建出来的镜像

```
$ docker run my-first-image:1.0
this is my first docker image
```

docker run 命令会自动启动一个新的容器，我们发现，容器启动的时候，执行了我们在 dockerfile 中定义的命令，打印出 this is my first docker image 这个信息，说明容器正常运行了。

2. 检查容器运行的状态

```
$ docker ps -a
CONTAINER ID      IMAGE                  COMMAND                   CREATED
8de38c5dc5b0      my-first-image:1.0     "/bin/sh -c 'echo \"th"   3 minutes ago

STATUS                          PORTS         NAMES
Exited (0) 3 minutes ago                      backstabbing_nobel
```

上面的信息显示出刚才我们运行的容器的详细信息：
使用的哪个 image：my-first-image:1.0,

运行时执行的命令：”/bin/sh -c 'echo \"th"，这里只是部分显示，
容器创建的时间：3 minutes ago，3分钟前创建，
容器的运行状态：Exited (0) 3 minutes ago，容器已经退出，退出的code是0，退出时间是3分钟前，
容器的端口情况：这里没有使用到，
容器的名字：backstabbing_nobel，由于我们没有指定容器的名字，所以这里是随机生成的名字。

由于我们的容器已经退出，因此上面的 ps 命令，需要加上 -a 这个 option，表示显示所有容器的状态。

4. 再次运行我们上个步骤构建出来的镜像，并用 --name 这个 option 指定容器名称

```
$ docker run --name my-first-container my-first-image:1.0
this is my first docker image
```

5. 再次检查容器运行的状态

```
$ docker ps -a
CONTAINER ID      IMAGE                  COMMAND                   CREATED
c194ee30a431      my-first-image:1.0     "/bin/sh -c 'echo \"th"   13 seconds ago
8de38c5dc5b0      my-first-image:1.0     "/bin/sh -c 'echo \"th"   5 minutes ago

STATUS                            PORTS         NAMES
Exited (0) 12 seconds ago                       my-first-container
Exited (0) 3 minutes ago                        backstabbing_nobel
```

我们发现有一个新的容器刚刚被运行过，name 是 **my-first-container。**

5. 实践步骤

请修改 Dockerfile 中的 CMD 命令，将打印信息 this is my first docker image，改成 this is {yourname} first docker image，{yourname} 的部分替换成自己的名字，重新构建镜像，并使用 3.0 的 tag，然后运行这个镜像，观察输出结果。

向远程仓库推送镜像

1. 用 tag 命令重命名镜像

请修改 {your-name} 的部分，用自己的名字替换掉，以便于我们在镜像仓库中区别每个人的不同镜像。

```
$ docker tag my-first-image:3.0 cargo.caicloud.io/training/{your-name}-first-image:3.0
```

重命名完成后，请用 docker images 命令查看这个镜像。

2. 登录远程仓库

push 的操作需要权限，请使用账号 training，密码 T12345t 来登录。

```
$ docker login cargo.caicloud.io -u training -p T12345t
Login Succeeded
```

docker login 命令使用 -u 来制定用户名，使用 -p 来指定密码。

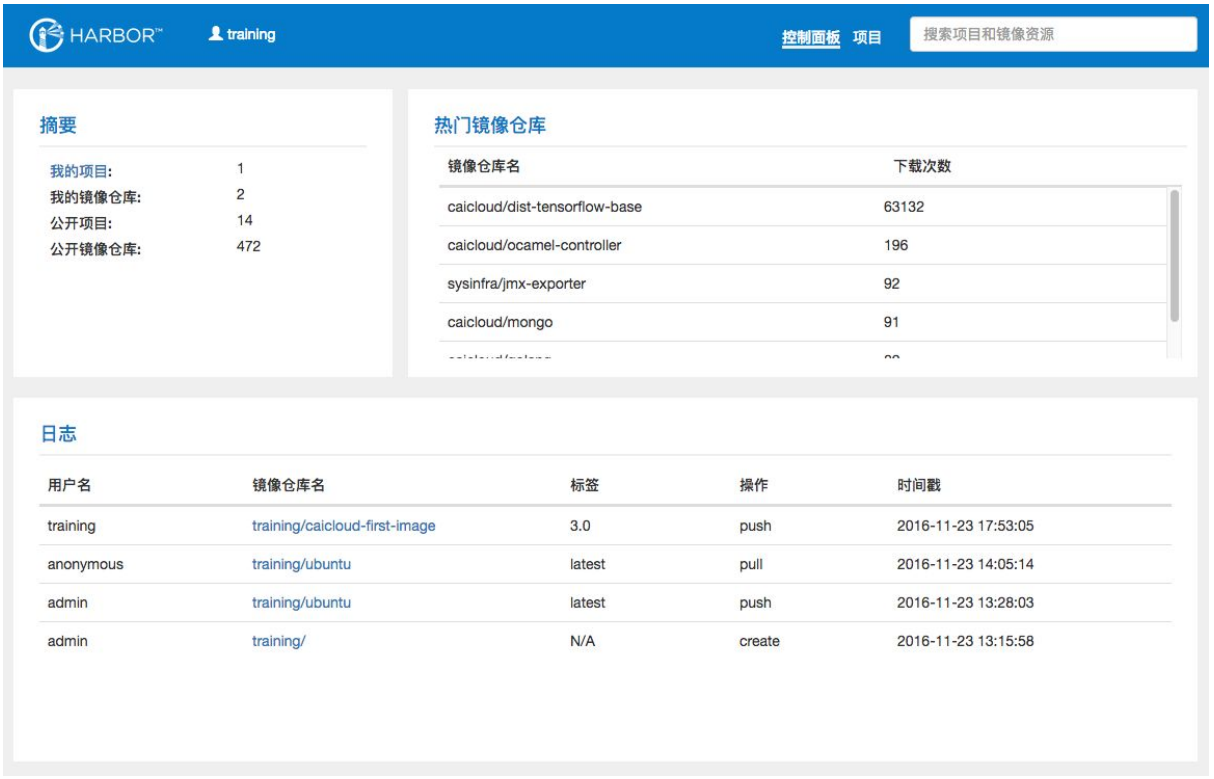
3. 向远程仓库推送镜像

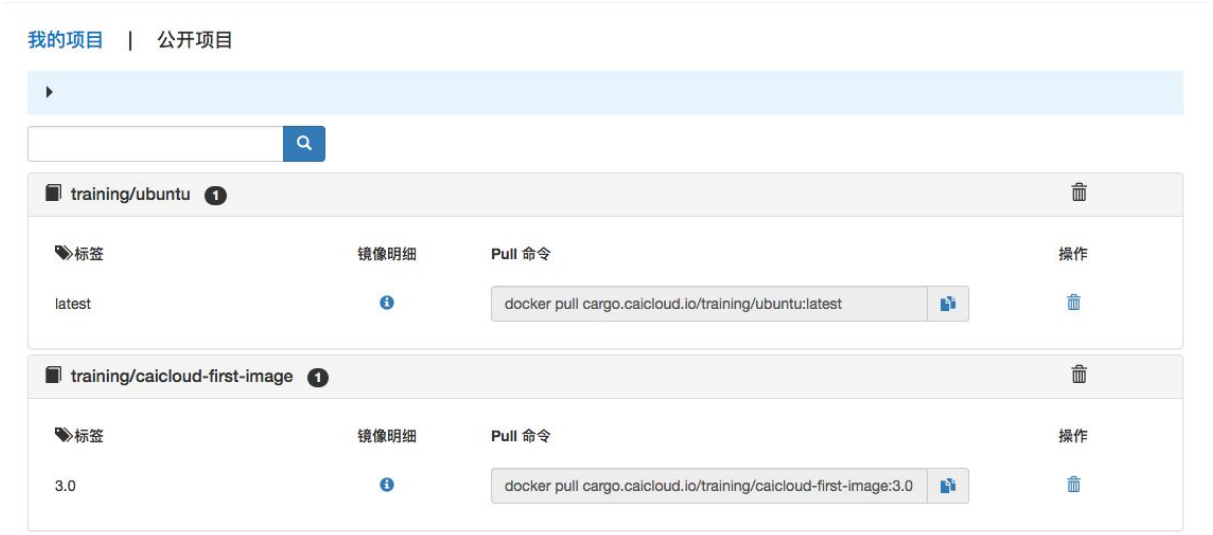
请注意修改 {your-name} 的部分。

```
$ docker push cargo.caicloud.io/training/{your-name}-first-image:3.0
The push refers to a repository [cargo.caicloud.io/training/caicloud-first-image]
0e20f4f8a593: Pushed
1633f88f8c9f: Pushed
46c98490f575: Pushed
8ba4b4ea187c: Pushed
c854e44a1a5a: Pushed
3.0: digest: sha256:231e873b28c2e6090391e1ca8109f2ef07faa5ca9d6bf71befa7f2512a66a686 size:
1357
```

4. 通过 Web 登录远程仓库，查看推送结果

请在浏览器中访问，https://cargo.caicloud.io，使用账号 training，密码 T12345t 完成登录，在项目中选择 training，并查找自己刚刚推送的镜像，入下图所示：





高级实践：部署基于 python 的 web 应用

1. 创建 my-web-app.py 文件，并赋予可执行权限

```
$ mkdir -p ~/my-web-app && cd ~/my-web-app
$ touch my-web-app.py
$ chmod +x my-web-app.py
```

2. 编辑 my-web-app.py 文件，添加下面的功能代码

```
#!/usr/bin/python
import SimpleHTTPServer
import SocketServer
from subprocess import call

class MyRequestHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_GET(self):
        print "Cool! This web server works!"
        self.send_response(200)
        self.send_header('Content-Type', 'text/plain')
        self.end_headers()

        self.wfile.write("Cool! This web server works!\n");
        self.wfile.close();
```

```
        return

PORT = 8000

Handler = MyRequestHandler

httpd = SocketServer.TCPServer(("", PORT), Handler)

print "serving at port", PORT
httpd.serve_forever()
```

代码解释：
上面的代码，基于SimpleHTTPServer，SocketServer 这两个 python 自带的组件，搭建了一个简单的 web 服务，开启 8000 这个端口，我们的 web 服务将监听这个端口，
我们重写了 do_GET 这个方法，当客户端通过 GET 方式访问我们的 web 服务，将看到 Cool! This web server works! 这样的信息，说明我们的 web 服务正常工作了。

3. pull 基于 ubuntu 的 python 基础镜像，并重命名

```
$ docker pull cargo.caicloud.io/training/ubuntu_python_base:latest
latest: Pulling from training/ubuntu_python_base
6bbedd9b76a4: Pull complete
fc19d60a83f1: Pull complete
de413bb911fd: Pull complete
2879a7ad3144: Pull complete
668604fde02e: Pull complete
05cdd15f49bb: Pull complete
e9d4ec73c583: Pull complete
Digest: sha256:21ec9239ec2972aaa313c8fb6add521e01fe4df474ccc12526d24f2f06dc45f0
Status: Downloaded newer image for cargo.caicloud.io/training/ubuntu_python_base:latest
```

这个镜像是我们提前构建好的，基于 ubuntu 镜像，安装了 python 相关的组件，由于安装组件的时间较长，这里我们直接 pull 下来使用。

这个基础镜像的 dockerfile 内容如下，供参考：

```
# build ubuntu base image with python package
# $ docker build -f dockerfile_ubuntu_base -t ubuntu_base:latest .

FROM ubuntu:16.04

MAINTAINER training@caicloud.io

# install docker
RUN apt-get update \
    && apt-get install docker.io -y

# install python
RUN apt-get install build-essential checkinstall -y \
    && apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev
tk-dev libgdbm-dev libc6-dev libbz2-dev -y \
    && cd ~/ \
    && apt-get install wget -y \
    && wget http://python.org/ftp/python/2.7.12/Python-2.7.12.tgz \
    && tar -xvf Python-2.7.12.tgz \
    && cd Python-2.7.12 \
    && ./configure \
    && make install
```

我们解释下上面的新出现的 dockerfile 关键字：
MAINTAINER，维护者，这里可以理解为作者，
RUN，每条 RUN 指令，docker 都会在基础镜像的上面，添加一个新的 layer，将 RUN 后面的命令执行内容添加到这个 layer 里面，注意 RUN 后面的命令，需要换行的话，使用 \，多个命令之间用 && 连接。

Dockerfile 关键字参考，https://docs.docker.com/engine/reference/builder/

由于镜像名称较长，我们重命名一下，方便后面的使用，我们重命名为 python_bade:latest。

```
$ docker tag cargo.caicloud.io/training/ubuntu_python_base:latest python_base:latest
```

4. 基于基础镜像来运行我们上面的 python 脚本，启动 web 应用

```
$ docker run --name my-web-server -v ~/my-web-app:/my-web-app -p 8000:8000 python_base:latest
python /my-web-app/my-web-app.py
```

命令详解：
--name，指定容器的名字，上面我们用 my-web-server 来指定我们的容器名称，

-v，等同于 --volume，用来挂载数据卷，格式是宿主机目录在前，容器内部目录在后，中间用冒号分隔，
上面的例子，我们把 ~/my-web-app，这个目录，挂载在容器内部的 /my-web-app 这个位置，相当于我们在容器内部对/my-web-app 目录进行的读写操作，直接操作在宿主机的 ~/my-web-app 这个目录，

-p，等同于 --port，用来绑定端口，格式是宿主机端口在前，容器内部端口在后，中间用冒号分隔，
上面的例子，我们把宿主机的 8000 端口，和容器的 8000 端口绑定，

python_base:latest，容器所用的镜像名称，包括 repository + tag，

python /my-web-app/my-web-app.py，容器启动所执行的命令，这里我们用 python 来启动 my-web-app.py 这个脚本。

docker run 命令参考链接，https://docs.docker.com/engine/reference/commandline/run/

5. 查看运行状态

```
$ docker ps -a
CONTAINER ID        IMAGE                 COMMAND                  CREATED
68d6274ad110        python_base:latest    "python /my-web-app/m"   About a minute ago

STATUS              PORTS                     NAMES
Up About a minute   0.0.0.0:8000->8000/tcp    my-web-server
```

状态详解：
CONTAINER ID：每个启动的容器，会分配一个唯一的容器 ID，用来标识这个容器，
IMAGE：这个容器所用到的镜像，
COMMAND：容器启动的时候，执行的命令，这里由于长度显示，只是显示了前半部分，

CREATED：容器创建的时间，这里显示大概1分钟之创建，
STATUS：容器的状态，up 表示运行中，容器还有下面几种状态，created，restarting，running，paused，exited，dead，
PORTS：容器的端口绑定情况，0.0.0.0:8000，表示宿主机的端口，箭头后面的8000表示容器的端口，tcp 表示端口绑定的是 tcp 协议，
NAMES：容器的名称，这里显示我们 run 命令上面的指定的 my-web-server 这个名字。

6. 检查 web 服务是否正常

通过浏览器，或者下面的命令行形式，访问 http://localhost:8000 这个地址。

```
$ curl http://localhost:8000
Cool! This web server works!
```

上面的信息表示我们的web服务运行正常。

使用 dockerfile 构建我们自己的 web 应用镜像

1. 创建新的 dockerfile

```
$ touch my-web-app-dockerfile
```

添加下面的内容到 my-web-app-dockerfile 中：

```
FROM python_base:latest

MAINTAINER training@caicloud.io

COPY ./my-web-app.py /my-web-app.py

WORKDIR /
EXPOSE 8000

ENTRYPOINT ["python", "my-web-app.py"]
```

关键字解释：
COPY，将宿主机的文件，拷贝至容器内，前面是宿主机的文件，后面是容器内的文件，
WORKDIR，表示RUN，CMD，ENTRYPOINT，COPY， ADD等命令的工作目录，如果这个目录不存在，那么在构建的时候，会创建这个目录，即使这个目录并没有使用，
EXPOSE，表示容器启动的时候，会监听这个端口，
ENTRYPOINT，表示容器启动的时候，入口的命令，注意，如果在dockerfile 中指定多个ENTRYPOINT，只有最后一个会起作用。

2. 构建我们自己的 web 应用镜像

```
$ docker build -f my-web-app-dockerfile ./ -t my-web-app:1.0
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM python_base:latest
 ---> 4a7f0ac1273f
Step 2 : MAINTAINER training@caicloud.io
 ---> Using cache
 ---> 64716a9e24a8
```

```
Step 3 : COPY ./my-web-app.py /my-web-app.py
 ---> 6e3c1a5ea614
Removing intermediate container 4193698cd1fa
Step 4 : WORKDIR /
 ---> Running in 6e802505e262
 ---> 4ca40d9776d3
Removing intermediate container 6e802505e262
Step 5 : EXPOSE 8000
 ---> Running in e65fa076ee8f
 ---> 64ffb0b62eab
Removing intermediate container e65fa076ee8f
Step 6 : ENTRYPOINT python my-web-app.py
 ---> Running in 0bd83b28dbf0
 ---> 0113d8df572b
Removing intermediate container 0bd83b28dbf0
Successfully built 0113d8df572b
```

3. 运行 web 应用镜像

```
$ docker run --name my-web-app -p 8888:8000 -d my-web-app:1.0
c89e5b0bb4e26d357ecac16533cad84494da1902d860e84cebd18ecb79ae9567
```

这次我们使用宿主机的 8888 这个端口来绑定容器的 8000 端口，所以，我们需要通过 8888 这个端口访问我们的 web 服务，
我们会发现这次的 docker run 命令多了 -d 这个参数，这个参数相当于 --detach，
意思是，启动这个容器，并在后台运行，
同时，打印出这个容器的 ID，
c89e5b0bb4e26d357ecac16533cad84494da1902d860e84cebd18ecb79ae9567，我们会发现，这个 ID 比我们的 docker ps -a 命令显示的要长，这个是完整的 ID，
docker ps -a 命令显示的这个 ID 的前12位，如下所示：

```
$ docker ps -a
CONTAINER ID        IMAGE                                       COMMAND
c89e5b0bb4e2        my-web-app:1.0                              "python my-web-app.py"

CREATED             STATUS              PORTS                       NAMES
9 minutes ago       Up 9 minutes        0.0.0.0:8888->8000/tcp      my-web-app
```

4. 检查我们的 web 服务是否正常工作

通过浏览器，或者下面的命令行形式，访问 http://localhost:8888 这个地址。

```
$ curl http://localhost:8888
Cool! This web server works!
```

5. 停止一个容器

```
$ docker stop c89e5b0bb4e2
c89e5b0bb4e2
```

我们使用 docker stop 来停止一个容器，docker stop 后面可以指定容器 ID，也可以指定容器 name。

```
$ docker ps -a
CONTAINER ID        IMAGE                                COMMAND                  CREATED
```

```
c89e5b0bb4e2        my-web-app:1.0                  "python my-web-app.py"      19 minutes
ago

STATUS                              PORTS                         NAMES
Exited (137) 12 seconds ago         0.0.0.0:8888->8000/tcp        my-web-app
```

我们发现这个容器的状态是 Exited (137) 12 seconds ago，说明容器已经退出了，并且退出的时间是12秒之前，
下面我们验证一下我们的 web 服务是不是停止了：

```
$ curl http://localhost:8888
curl: (7) Failed to connect to localhost port 8888: Connection refused
```

连接被拒绝，说明 web 服务已经停止了。

6. 重启启动一个容器

```
$ docker start my-web-app
my-web-app
```

我们使用 docker start 来停止一个容器，类似 docker stop 命令，docker start 后面可以指定容器 ID，也可以指定容器 name，这里我们指定容器的名字。

```
$ curl http://localhost:8888
Cool! This web server works!
```

访问成功，说明 web 服务已经重新启动了。
当然，也可以使用 docker ps -a 来查看容器的状态。

自由实践

请参考下面的链接，修改 dockerfile 的内容，使用 VOLUME 关键字，替换掉 COPY 关键字，即把宿主机的一个目录，挂载到容器内，这样在容器内即可访问我们宿主机上的文件，具体步骤如下：

参考链接， https://docs.docker.com/engine/reference/builder/

1. 修改 my-web-app.py 里面的内容，在 GET 方法返回的字符串中添加自己的名字，例如改成下面的内容：Cool! This caicloud's web server works!,
1. 修改 dockerfile 的内容，重新构建我们的 my-web-app 镜像，请以自己的名字做 tag,
2. 运行它，验证是否正常工作,
3. 将这个镜像 push 到远程仓库，cargo.caicloud.io/training/xxx,
4. 请两两自由结对，把你的镜像名称告诉你的队友，让他从远程仓库 pull 你的镜像，并在他的本地运行，验证是否正常工作。

注意，如果启动对方的镜像时，启动失败，有可能是宿主机端口被之前的容器占用，请更换宿主机端口，或者使用 docker rm 加容器名，或者容器 ID 的方法移除之前的容器。

实践过程中，如有任何疑问，请举手提问，谢谢。