**Docker容器**

#交互式运行ubuntu容器，而且退出即删除

docker run -it --rm ubuntu:14.04 bash

#启动nginx容器，并修改nginx主页，-d后台运行

docker run --name webserver -d -p 80:80 nginx

docker exec -it webserver bash

echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

exit

#查看容器的改动内容

docker diff webserver



#保存修改(创建新的镜像)--author 是指定修改的作者，而 --message 则是记录本次修改的内容。

这点和 git 版本控制相似

docker commit --author "MT" --message "Modify host page" webserver nginx:v2

#docker history nginx:v2



使用 docker commit 意味着所有对镜像的操作都是黑箱操作，生成的镜

像也被称为黑箱镜像，换句话说，就是除了制作镜像的人知道执行过什么命令、怎

么生成的镜像，别人根本无从得知。而且，即使是这个制作镜像的人，过一段时间

后也无法记清具体在操作的。虽然 docker diff 或许可以告诉得到一些线索，

但是远远不到可以确保生成一致镜像的地步。这种黑箱镜像的维护工作是非常痛苦

的。

docker commit 命令除了学习之外，还有一些特殊的应用场合，比如被入侵后保

存现场等。但是，不要使用 docker commit 定制镜像，定制行为应该使用

Dockerfile 来完成

Dockerfile:

[

From nginx

RUN echo '<h1>Hello , Docker!</h1>' > /usr/share/nginx/html/index.html  
]

FROM 就是指定基础镜像，因此一个 Dockerfile 中 FROM 是必备的指令，并

且必须是第一条指令。

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为

scratch 。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。

FROM scratch

.........

不以任何系统为基础，直接将可执行文件复制进镜像的做法并不罕见，比如

swarm 、 coreos/etcd 。对于 Linux 下静态编译的程序来说，并不需要有操作

系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 FROM

scratch 会让镜像体积更加小巧。使用 Go 语言 开发的应用很多会使用这种方式

来制作镜像，这也是为什么有人认为 Go 是特别适合容器微服务架构的语言的原因

之一。

简化Dockerfile：

[

FROM debian:jessie

RUN apt-get update

RUN apt-get install -y gcc libc6-dev make

RUN wget -O redis.tar.gz "http://download.redis.io/releases/redi

s-3.2.5.tar.gz"

RUN mkdir -p /usr/src/redis

RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1

RUN make -C /usr/src/redis

RUN make -C /usr/src/redis install  
]

这样会创建7层镜像

[

FROM debian:jessie

RUN buildDeps='gcc libc6-dev make' \

&& apt-get update \

&& apt-get install -y $buildDeps \

&& wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz" \

&& mkdir -p /usr/src/redis \

&& tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \

&& make -C /usr/src/redis \

&& make -C /usr/src/redis install \

&& rm -rf /var/lib/apt/lists/\* \

&& rm redis.tar.gz \

&& rm -r /usr/src/redis \

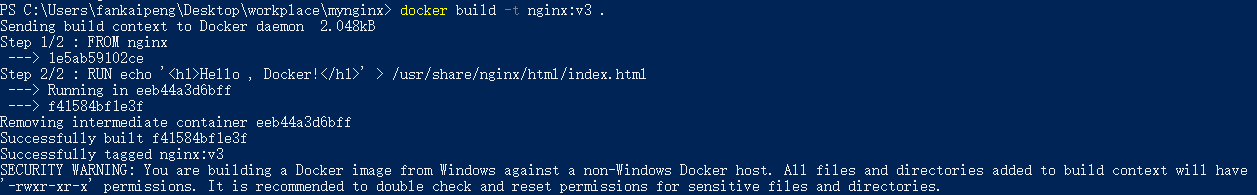
&& apt-get purge -y --auto-remove $buildDeps

]

这一组命令的最后添加了清理工作的命令，删除了为了编译构建

所需要的软件，清理了所有下载、展开的文件，并且还清理了 apt 缓存文件，很多人初学 Docker 制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件。

docker build -t nginx:v3 .



上述命令的 ’. ’为指定上下文路径

当构建的时候，用户会指定构建镜像上下文的路

径， docker build 命令得知这个路径后，会将路径下的所有内容打包，然后上

传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜

像所需的一切文件。

如果在 Dockerfile 中这么写：

COPY ./package.json /app/

这并不是要复制执行 docker build 命令所在的目录下的 package.json ，也

不是复制 Dockerfile 所在目录下的 package.json ，而是复制 上下文

（**context**） 目录下的 package.json 。

因此， COPY 这类指令中的源文件的路径都是相对路径。这也是初学者经常会问

的为什么 COPY ../package.json /app 或者 COPY /opt/xxxx /app 无法工

作的原因，因为这些路径已经超出了上下文的范围，Docker 引擎无法获得这些位

置的文件。如果真的需要那些文件，应该将它们复制到上下文目录中去。

如果目录下有些东西

确实不希望构建时传给 Docker 引擎，那么可以用 .gitignore 一样的语法写一个 .dockerignore ，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

COPY package.json /usr/src/app/

仅在需要自动解压缩的场合使用 ADD

CMD [ "sh", "-c", "echo $HOME" ]

CMD 指令的格式和 RUN 相似，也是两种格式：

shell 格式： CMD <命令>

exec 格式： CMD ["可执行文件", "参数1", "参数2"...]

一般推荐使用 exec 格式，这类格式在解析时会被解析为 JSON数组，因此一定要使用双引号 " ，而不要使用单引号。

FROM ubuntu:16.04

RUN apt-get update \

&& apt-get install -y curl \

&& rm -rf /var/lib/apt/lists/\*

CMD [ "curl", "-s", "http://ip.cn" ]

FROM ubuntu:16.04

RUN apt-get update \

&& apt-get install -y curl \

&& rm -rf /var/lib/apt/lists/\*

ENTRYPOINT [ "curl", "-s", "http://ip.cn" ]

docker run myip



docker run myip –i



ENTRYPOINT 看上去与 CMD 很像，它们都可以指定要执行的命令及其参数。不同的地方在于 ENTRYPOINT 不会被忽略，一定会被执行，即使运行 docker run 时指定了其他命令。

ENV 这个指令很简单，就是设置环境变量而已，无论是后面的其它指令，如 RUN ，还

是运行时的应用，都可以直接使用这里定义的环境变量。

docker exec –it [container name] /bin/bash （必须运行docker时有-it参数）

docker exec –it [container name] /bin/sh （万能方法，针对没有开启/bin/bash）