编号： 002

Docker安装与使用介绍

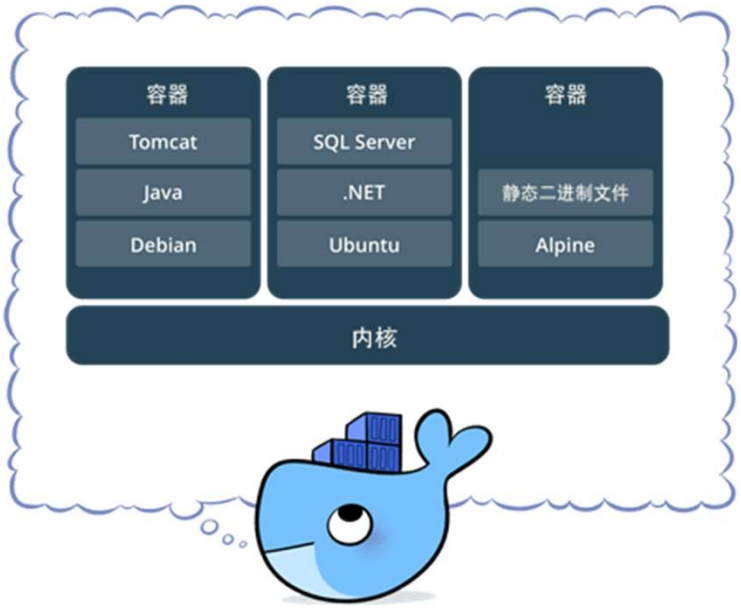
2019.7

目录

概念介绍

1.概念

Docker 是世界领先的软件容器平台。开发人员利用 Docker 可以消除协作编码时“在我的机器上可正常工作”的问题。运维人员利用 Docker 可以在隔离容器中并行运行和管理应用，获得更好的计算密度。企业利用 Docker 可以构建敏捷的软件交付管道，以更快的速度、更高的安全性和可靠的信誉为 Linux 和 Windows Server 应用发布新功能。

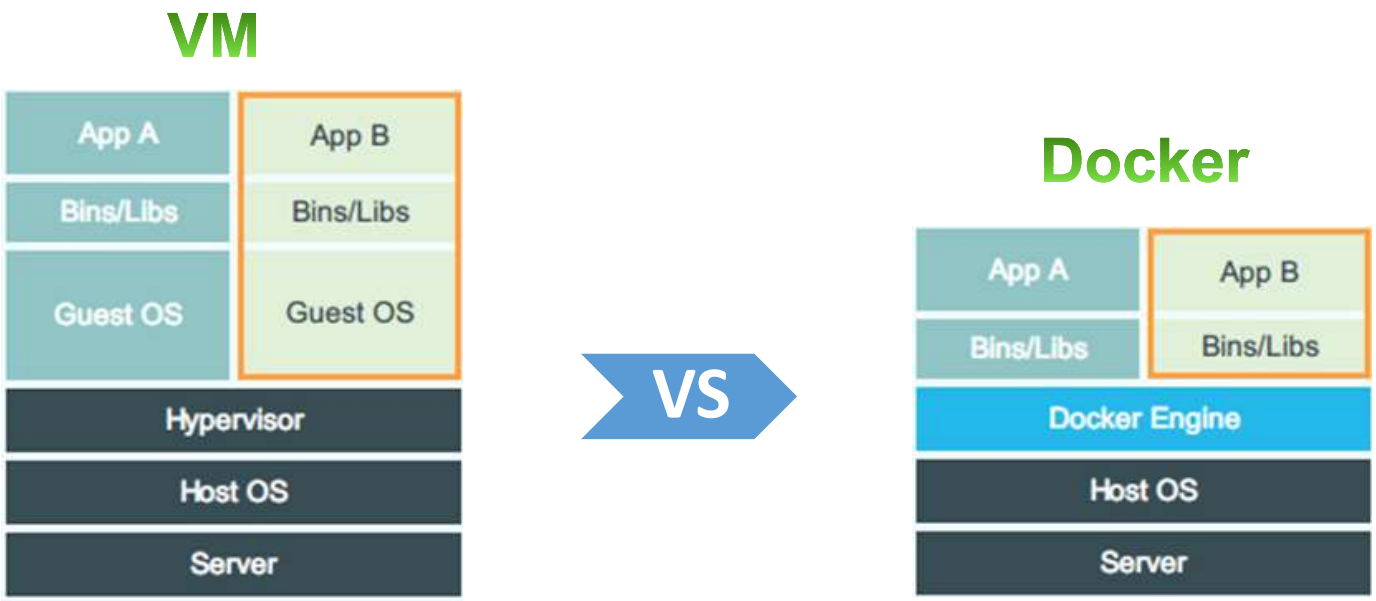


2.关于 Docker

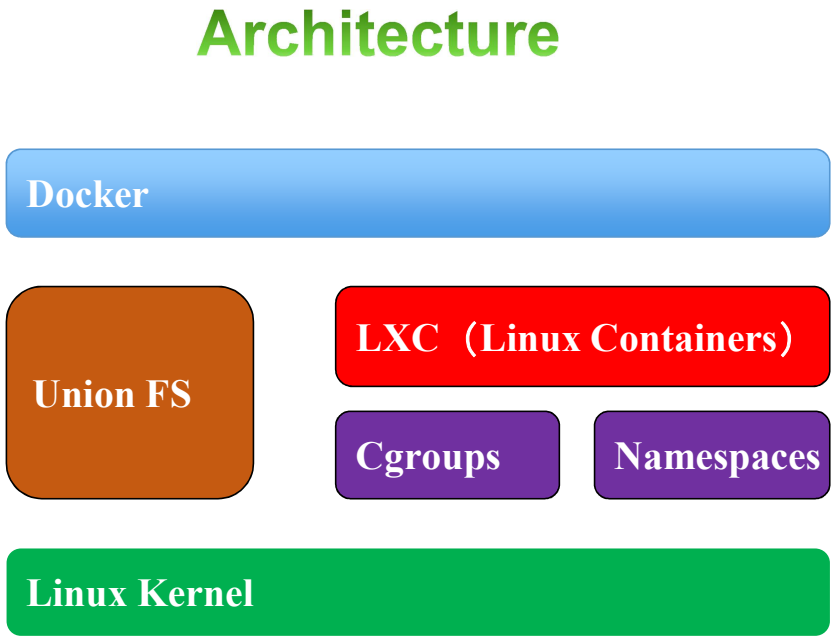
容器镜像是轻量的、可执行的独立软件包，包含软件运行所需的所有内容：代码、运行时环境、系统工具、系统库和设置。容器化软件适用于基于 Linux 和 Windows 的应用，在任何环境中都能够始终如一地运行。容器赋予了软件独立性，使其免受外在环境差异（例如，开发和预演环境的差异）的影响，从而有助于减少团队间在相同基础设施上运行不同软件时的冲突。

了解Docker

Why not VMs？用户需要的是高效运行环境而非OS，GuestOS既浪费资源又难于管理，轻量级的Container更加灵活和快速。



了解Docker



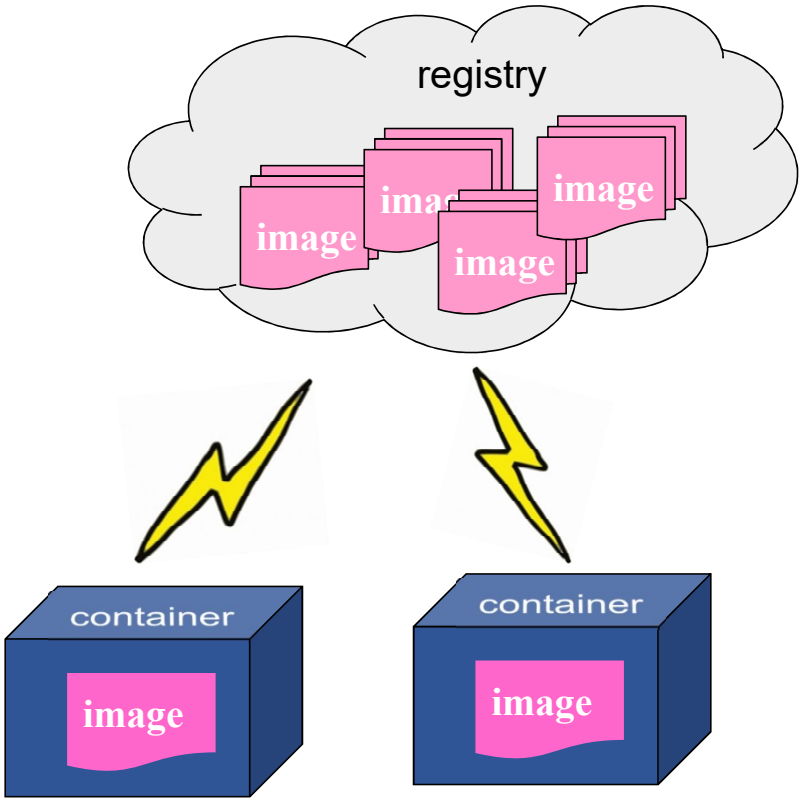
➢**Namespaces**：LXC所实现的隔离性主要是来自kernel的namespace, 其中pid, net, ipc, mnt, uts 等namespace将container的进程, 网络, 消息, 文件系统和hostname 隔离开

➢**Cgroups**：实现了对资源的配额和度量。

➢**UnionFS**：是一种支持将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)的文件系统

➢**LXC**：Linux Container，提供了一种操作系统级的虚拟化方法。借助于namespace的隔离机制和cgroup限额功能来管理container

You need to Know



➢Docker images:A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache and your web application installed. Images are used to create Docker containers.

➢Docker Registries:Docker registries hold images. These are public or private stores from which you upload or download images. The public Docker registry is called Docker Hub.

➢Docker containers:Each container is created from a Docker image. A Docker container holds everything that is needed for an application to run

仓库（Registry）

- 所谓仓库，其实是个镜像仓库，里面有很多别人已经打包好的镜像，可以直接使用docker pull命令将仓库中的镜像拉到本地，默认的仓库Docker的官方仓库Docker Hub Registry。因为墙的缘故，官方仓库的速度会比较慢，可以配一个官方的中国加速镜像，方法是：修改/etc/docker/daemon.json，加上如下的键值

```
{
    "registry-mirrors": ["https://registry.docker-cn.com"]
}
```

之后重启docker服务即可生效。

镜像（Image）

通过docker images命令可以看到本地已有的镜像：

```
[root@iZ8vb8vuru3z96w7whl4yhZ ~]# docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
beyond/vsftp-byeond           1.0.0               3cec6b9defcb        6 days ago          357MB
<none>                        <none>              5d48f1139fb0        6 days ago          357MB
<none>                        <none>              696e0558c720        6 days ago          357MB
<none>                        <none>              d410585312f9        6 days ago          356MB
sonatype/nexus3               latest              36b0df681a47        10 days ago         582MB
openfrontier/gerrit           latest              3b8efd967977        2 weeks ago         213MB
sameersbn/redmine             latest              c7a976e0ea70        3 weeks ago         817MB
nginx                         latest              719cd2e3ed04        3 weeks ago         109MB
```

每个镜像都有一个IMAGE ID作为唯一标识，可以看出这个镜像的IMAGE ID为5d48f1139fb0，使用镜像的id可以将它删除，命令如下：

```
[root@iZ8vb8vuru3z96w7whl4yhZ ~]# docker rmi 5d48f1139fb0
Deleted: sha256:5d48f1139fb00e34f5b6f8786a083e1e30a9b18c08e5527cd64f7dc600738646
Deleted: sha256:c2bfdaee541e210dcfbc9e558dc0fa79361db30a1f5b0f9de4fb728635d04bdc
Deleted: sha256:50bf6bc289caf994db6a17c1987f1ef19c277f58de2db41e6dc2e5250450ba9f
Deleted: sha256:54d0c3b7b51282bf46500173eb006da968e293b55e40bbcab29033f8109070b6
Deleted: sha256:b37b4085fca638bd4e02c8a3a7905693a0e3c3bc7397480dc627cede285a4de0
Deleted: sha256:e2669fb19a5935cb21bf56f6f301665be5019862c7a207e9915bffa319f57277
Deleted: sha256:a1b7d01ae6b3863812dbac6867c9b9bac7e1e02a35b6cd28fdbc47d4a5c13173
Deleted: sha256:550932050db1f7a13dd26767f2109620c0b1ccc69901b08a9ad97fc5c10c3f65
Deleted: sha256:8086ea543505642395f5a7ebfc32fc05adc6a7ee3296a12616b624580ea31088
Deleted: sha256:d0a27ae9047be88c5762fa8cba193ce9ef3d2c2acfa4eebb232a489c6cd00d1c
Deleted: sha256:24fe592367c59f48efe050934d3091b614643e29110ebb08a79f7f0aa3ce3bf1
Deleted: sha256:95e7eed18dab67b71754a5fce4d74af2474b41208c602540eeef83286e527680
Deleted: sha256:494a1bd17caf6ea1dc1c63d2a819afb24bd90cab4bae51a950a6ff3c8e4db562
Deleted: sha256:72d9a8ab169dbf282ed8bd29d17a455dbbf2fedd1e33a14c2574df30cd7b5b79
Deleted: sha256:1b306e9844eac60c0796b71283c1a4f48f2fc6ce83c59620dac582f6590b4357
Deleted: sha256:b68eb56b87c95c9b495934deb90ff8e77d0db6710f61f89e138b1dfe98fdb5f3
Deleted: sha256:6c4b58d806d2f9e5829c182e7a84f3d4d7d6591e2d50172f82e91c756bb78373
Deleted: sha256:b2d157a7d00752286812240233f63e7af12dc329239b17b4603932be81468a23
```

容器（Container）

- 然后使用docker run来运行这个镜像（运行之后镜像就变成一个容器）：

```
docker run  --name nginx -d \
-p 80:80 \
-p 443:443 \
-v /data/devops/nginx/data:/usr/share/nginx/ \
-v /data/devops/nginx/logs:/var/log/nginx \
-v /data/devops/nginx/conf.d:/etc/nginx/conf.d \
 nginx
```

使用docker stop命令可以停止这个后台运行的容器：

```
docker stop 6d194c12a8ae
```

如果你想继续让这个容器运行，可以使用docker start命令：

```
docker start 6d194c12a8ae
```

这里把容器和镜像容易混淆的命令总结在了下表中：

	删除	启动
镜像	docker rmi	docker run
容器	docker rm	docker start

特性

- 轻量

在一台机器上运行的多个 Docker 容器可以共享这台机器的操作系统内核；它们能够迅速启动，只需占用很少的计算和内存资源。镜像是通过文件系统层进行构造的，并共享一些公共文件。这样就能尽量降低磁盘用量，并能更快地下载镜像。

- 标准

Docker 容器基于开放式标准，能够在所有主流 Linux 版本、Microsoft Windows 以及包括 VM、裸机服务器和云在内的任何基础设施上运行。

- 安全

Docker 赋予应用的隔离性不仅限于彼此隔离，还独立于底层的基础设施。Docker 默认提供最强的隔离，因此应用出现问题，也只是单个容器的问题，而不会波及到整台机器。

目录

安装Docker

准备一个Centos7.6 (64-bit)版本，如CentOS-7-x86_64-Minimal-1810.iso

1、Docker 要求 CentOS 系统的内核版本高于 3.10 ，查看本页面的前提条件来验证你的CentOS 版本是否支持 Docker 。

通过 **uname -r** 命令查看你当前的内核版本

```
$ uname -r
```

2、使用 root 权限登录 Centos。确保 yum 包更新到最新。

```
$ sudo yum update
```

3、卸载旧版本(如果安装过旧版本的话)

```
$ sudo yum remove docker  docker-common docker-selinux docker-engine
```

4、安装需要的软件包， yum-util 提供yum-config-manager功能，另外两个是devicemapper驱动依赖的

```
$ sudo yum install -y yum-utils device-mapper-persistent-data lvm2
```

5、设置yum源

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

```
[root@localhost local]# sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
已加载插件：fastestmirror, langpacks
adding repo from: https://download.docker.com/linux/centos/docker-ce.repo
grabbing file https://download.docker.com/linux/centos/docker-ce.repo to /etc/yum.repos.d/docker-ce.repo
repo saved to /etc/yum.repos.d/docker-ce.repo
```

6、可以查看所有仓库中所有docker版本，并选择特定版本安装

```
$ yum list docker-ce --showduplicates | sort -r
```

```
[root@localhost local]# yum list docker-ce --showduplicates | sort -r
已加载插件：fastestmirror, langpacks
可安装的软件包
 * updates: mirrors.aliyun.com
Loading mirror speeds from cached hostfile
 * extras: mirrors.aliyun.com
docker-ce.x86_64            17.12.0.ce-1.el7.centos             docker-ce-stable
docker-ce.x86_64            17.09.1.ce-1.el7.centos             docker-ce-stable
docker-ce.x86_64            17.09.0.ce-1.el7.centos             docker-ce-stable
docker-ce.x86_64            17.06.2.ce-1.el7.centos             docker-ce-stable
docker-ce.x86_64            17.06.1.ce-1.el7.centos             docker-ce-stable
docker-ce.x86_64            17.06.0.ce-1.el7.centos             docker-ce-stable
docker-ce.x86_64            17.03.2.ce-1.el7.centos             docker-ce-stable
docker-ce.x86_64            17.03.1.ce-1.el7.centos             docker-ce-stable
docker-ce.x86_64            17.03.0.ce-1.el7.centos             docker-ce-stable
 * base: mirrors.aliyun.com
```

7、安装docker

```
$ sudo yum install docker-ce  #由于repo中默认只开启stable仓库，故这里安装的是最新稳定版17.12.0
$ sudo yum install <FQPN>  # 例如: sudo yum install docker-ce-17.12.0.ce
```

```
安装  1 软件包

总计：30 M
安装大小：123 M
Is this ok [y/d/N]: y
Downloading packages:
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  正在安装    : docker-ce-17.12.0.ce-1.el7.centos.x86_64
  验证中      : docker-ce-17.12.0.ce-1.el7.centos.x86_64

已安装:
  docker-ce.x86_64 0:17.12.0.ce-1.el7.centos

完毕！
```

8、启动并加入开机启动

```
$ sudo systemctl start docker
$ sudo systemctl enable docker
```

9、验证安装是否成功(有client和service两部分表示docker安装启动都成功了)

```
$ docker version
```

Docker配置--Docker 开启2375端口，提供外部访问docker

1.编辑docker文件：/usr/lib/systemd/system/docker.service修改修改ExecStart行为下面内容

```
[root@localhost ~]# cat /usr/lib/systemd/system/docker.service
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
BindsTo=containerd.service
After=network-online.target firewalld.service containerd.service
Wants=network-online.target
Requires=docker.socket

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock  -H tcp://0.0.0.0:2375  -H unix:///var/run/docker.sock
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always

# Note that StartLimit* options were moved from "Service" to "Unit" in systemd 229.
# Both the old, and new location are accepted by systemd 229 and up, so using the old location
# to make them work for either version of systemd.
StartLimitBurst=3

# Note that StartLimitInterval was renamed to StartLimitIntervalSec in systemd 230.
# Both the old, and new name are accepted by systemd 230 and up, so using the old name to make
# this option work for either version of systemd.
StartLimitInterval=60s
```

2. 重新加载docker配置

systemctl daemon-reload *// 1，加载docker守护线程*

systemctl restart docker *// 2，重启docker*

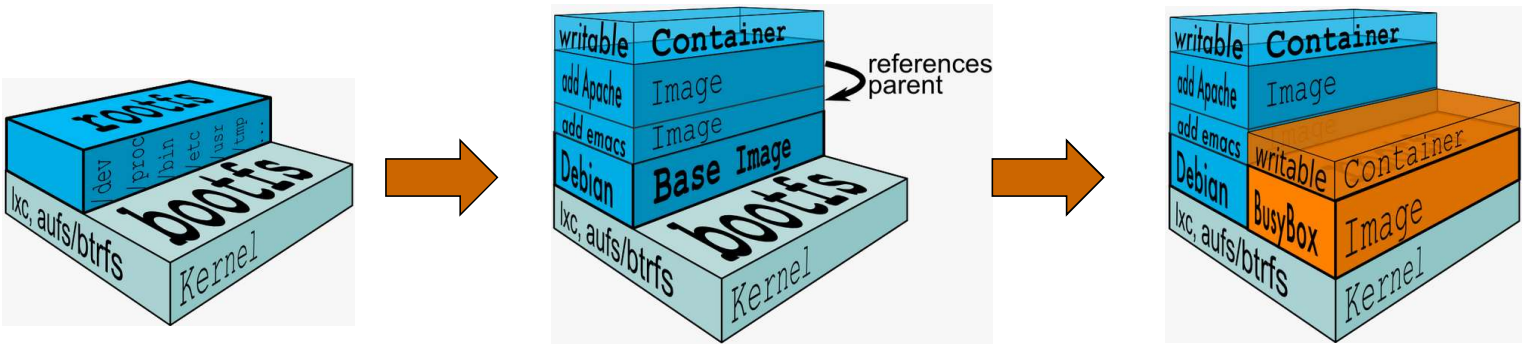
目录

Refresher on Docker

典型的Linux启动到运行需要两个FS：bootfs + rootfs。bootfs 主要包含 bootloader 和 kernel, bootloader主要是引导加载kernel, 当boot成功后 kernel 被加载到内存中后 bootfs就被umount了；rootfs (root file system) 包含的就是典型 Linux 系统中的 /dev, /proc,/bin, /etc 等标准目录和文件。

在docker中，对 rootfs先以readonly方式加载并检查，接下来利用 union mount 将一个 readwrite 文件系统挂载在 readonly 的rootfs之上，并且允许再次将下层的 file system设定为readonly 并且向上叠加, 这样一组readonly和一个writeable的结构构成一个container的运行目录, 每一个被称作一个Layer。每一个对readonly层文件/目录的修改都只会存在于上层的writeable层中。由于不存在竞争, 多个container可以共享readonly的layer。所以docker将readonly的层称作 **"image"** - 对于container而言整个rootfs都是read-write的，但事实上所有的修改都写入最上层的writeable层中, image不保存用户状态，可以用于模板、重建和复制。

从一个image启动一个container时，docker会先加载其下层image直到base image，用户的进程运行在writeable的layer中。所有image中的数据信息以及ID、网络和lxc管理的资源限制等具体container的配置，构成一个docker概念上的container。



Working with Containers

➢Running a Web Application in Docker

```
docker run -d -P  --name nginxTest    nginx
```

-P flag： tells Docker to map any required network ports inside our container to our host. This lets us view our web application.

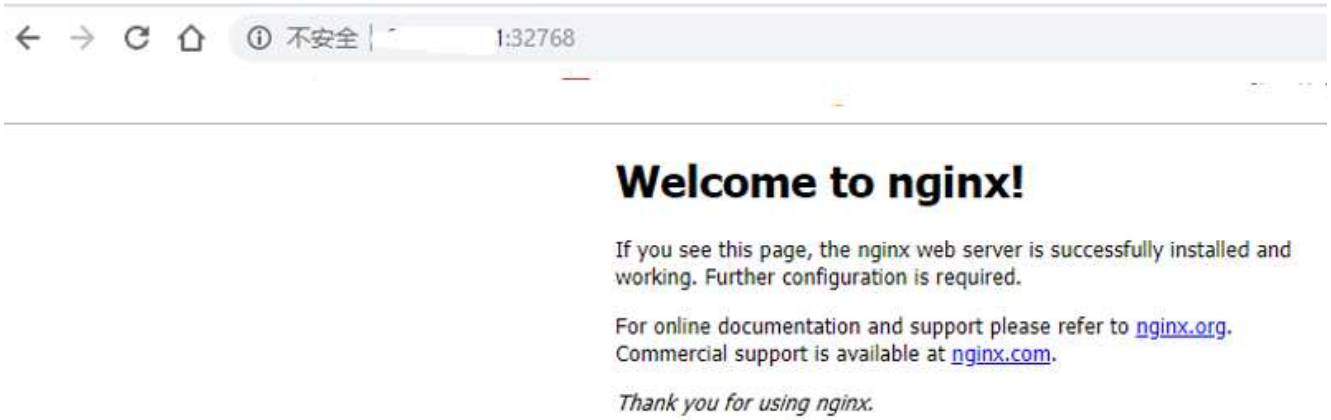
```
[root@iZ8vb8vuru3z96w7whl4yhZ html]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                     NAMES
6d194c12a8ae        nginx               "nginx -g 'daemon of…"   3 seconds ago       Up 1 second         0.0.0.0:32768->80/tcp     nginxTest
```

Docker has exposed port 80 (the default nginx http port) on local docker host port 32768.

Tips：如果是在虚拟机上使用docker，local docker host为虚拟机的ip地址

示例

示例：



Working with Docker images

关于images的操作命令

- Listing images on the host: docker images
- Getting a new image： docker pull centos
- Finding images： docker search sinatra
- .....

Tips：目前为止，我们看到有两种类型的images，不带前缀的如ubuntu，称为base或root images，由Docker Inc创建；带前缀的如openfrontier/gerrit 或gitlab/gitlab-ce ，称为user images由Docker社区创建和维护，前缀openfrontier表示创建该image的user或组织。

创建自己的images

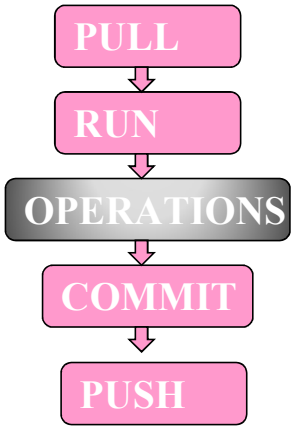
docker pull nginx

docker run -i -t nginx /bin/bash

docker commit -m "beyond nginx" -a="frank li" 6d194c12a8ae beyond/nginx:v0.5.0

```
[root@iZ8vb8vuru3z96w7whl4yhZ html]# docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
beyond/nginx                  v0.5.0              86ff75d8ee4d        6 seconds ago       109MB
```

docker push beyond/nginx



Docker Container Linking

➢给Container取一个有意义的名字

It's useful to name containers that do specific functions in a way that makes it easier for you to remember them, for example naming a container with a web application in it web.

It provides Docker with a reference point that allows it to refer to other containers, for example link container web to container db.

```
docker run -d \
--name mysql-server \
--hostname mysql-server \
-p 53306:3306 \
-e MYSQL_ROOT_PASSWORD=123456 \
-v $DOCKER_RUNTIME/data:/var/lib/mysql \
-v $DOCKER_RUNTIME/conf:/etc/mysql/conf.d \
mysql:5.6.35C
```

➢Container Linking

```
docker run --name=redmine -d \
  --link=mysql-server:mysql-server \
…
  sameersbn/redmine
```

--link name:alias： Where name is the name of the container we're linking to and alias is an alias for the link name.

Docker Container Linking

实际上， container web link container db ， 只是在web container内部做了如下两件事情：

- 增加环境变量；
- 更新/etc/hosts文件

```
[root@iZ8vb8vuru3z96w7whl4yhZ devops]# docker exec -it redmine /bin/bash
root@1212eff28166:/home/redmine/redmine# cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      mysql-server mysql-server
172.17.0.3      openldap openldap
172.17.0.5      1212eff28166
root@1212eff28166:/home/redmine/redmine#
```

```
[root@iZ8vb8vuru3z96w7whl4yhZ devops]# docker exec -it redmine env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=1212eff28166
TERM=xterm
MYSQL_SERVER_PORT=tcp://172.17.0.2:3306
MYSQL_SERVER_PORT_3306_TCP=tcp://172.17.0.2:3306
MYSQL_SERVER_PORT_3306_TCP_ADDR=172.17.0.2
MYSQL_SERVER_PORT_3306_TCP_PORT=3306
MYSQL_SERVER_PORT_3306_TCP_PROTO=tcp
MYSQL_SERVER_NAME=/redmine/mysql-server
MYSQL_SERVER_ENV_MYSQL_ROOT_PASSWORD=123456
MYSQL_SERVER_ENV_GOSU_VERSION=1.7
MYSQL_SERVER_ENV_MYSQL_MAJOR=5.6
MYSQL_SERVER_ENV_MYSQL_VERSION=5.6.35-1debian8
```

Docker Container Linking

现在使用另外一种方式替代--link来达到容器间的通信：docker network

查看local的网络信息：

```
[root@docker ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
5133ec415c3c        bridge              bridge              local
f359ca4e2d39        host                host                local
8d68673c045c        none                null                local
```

现在创建一个网络名为my_net且driver为bridge的网络：（默认创建的就是bridge）

```
[root@docker ~]# docker network create my_net
67e29f0e4a77c79144efc337a081a889188b5b8e289968f22be6e4ddd9b80610
[root@docker ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
5133ec415c3c        bridge              bridge              local
f359ca4e2d39        host                host                local
67e29f0e4a77        my_net              bridge              local
8d68673c045c        none                null                local
```

利用--network启动容器提供服务：

```
[root@docker ~]# docker run -d --name=php --network my_net --network-alias php -v /www:/usr/local/nginx/html php
6b493cbe8207dee4cb4d5945cfce305dba96914083bd7f46841b0b42376bcb99
[root@docker ~]# docker run -d --name=nginx --network my_net --network-alias nginx -v /www:/usr/local/nginx/html -p 80:80 nginx
5ab220196b52bb768bef508433f0b920eecee70c3ee47880ebc5e2a74b5ee254
```

通过选项--network-alias将取名的my_net起了一个别名

```
[root@docker ~]# docker exec -it nginx ping php
PING php (172.18.0.2) 56(84) bytes of data.
64 bytes from php.my_net (172.18.0.2): icmp_seq=1 ttl=64 time=0.079 ms
64 bytes from php.my_net (172.18.0.2): icmp_seq=2 ttl=64 time=0.090 ms
```

Managing Data in Containers

Data volumes

A data volume is a specially-designated directory within one or more containers that bypasses the Union File System to provide several useful features for persistent or shared data:

- Data volumes can be shared and reused between containers;
- Changes to a data volume are made directly;
- Changes to a data volume will not be included when you update an image;
- Volumes persist until no containers use them;

Adding a data volumes to a container

```
docker run -d \
...
-v $DOCKER_RUNTIME/data:/var/lib/mysql  mysql:5.6.35
```

-v:This will create a new volume inside a container at $DOCKER_RUNTIME/data
This will mount the local directory, $DOCKER_RUNTIME/data, into the container as the :/var/lib/mysql directory.这样实现了host与container的数据共享，对于测试比较有用，比如，将host上的source code （比如script）mount到container内之后，只要我们在host上修改代码，就可以立即影响到container中app的工作行为，达到代码测试目的。

Managing Data in Containers

Share persistent data between containers

```
$ sudo docker run -d -v /dbdata --name dbdata training/postgres echo Data-only container for
 postgres
$ sudo docker run -d --volumes-from dbdata --name db1 training/postgres
$ sudo docker run -d --volumes-from dbdata --name db2 training/postgres
```

如果你有一些数据想在多个容器间共享，或者想在一些临时性的容器中使用该数据，那么最好的方案就是你创建一个数据卷容器，然后从该临时性的容器中挂载该数据卷容器的数据。

--volumes-from flag： mount the /dbdata volume in another container.

Tips：在docker的整个设计中image是一个无状态的， 这对升级重用非常有利。那么标记状态的数据， 比如数据库的数据， 生产的log之类的，就可以放到data volume里，data volumn的持久化本质就是文件的持久化。

目录

Dockerfile

如果你想要从一个基础镜像开始建立一个自定义镜像，可以选择一步一步进行构建，也可以选择写一个配置文件，然后一条命令（docker build）完成构建，显然配置文件的方式可以更好地应对需求的变更，这个配置文件就是Dockerfile。学习Dockerfile的最好方式就是阅读别人写的Dockerfile，遇到不会的指令就查一查Dockerfile的文档，文档地址如下：https://docs.docker.com/engine/reference/builder/大家遇到不知道的指令多去里面翻一翻，下面我带着大家读几个开源Dockerfile，在读的过程中学习相关知识。第一个Dockerfile以阿里中间件大赛给的debian-jdk8镜像为例，Dockerfile文件如下：

1. 编译镜像

Dockerfile类似于Makfile，用户使用docker build就可以编译镜像，使用该命令可以设置编译镜像时使用的CPU数量、内存大小、文件路径等

```
语法：docker build [OPTIONS] PATH| URL| -
常见选项：
        -t 设置镜像的名称和TAG，格式为name:tag
        -f Dockerfile的名称，默认为PATH/Dockerfile
例子：docker build -f ~/php.Dockerfile .
注意：PATH是编译镜像使用的工作目录，Docker Daemon在编译开始时，会扫描PATH中的所有文件，可以在编译目录中加入.dockerignore过滤不需要的文件
```

2. dockerignore文件

编译开始前，Docker Daemon会读取编译目录中的.dockerignore文件，忽略其中的文件和目录，在其中可以使用通配符（?代表一个字符，*代表零个或任意个字符），使用通配符时，总会出现那么几个例外，这时可以使用!+文件名，Docker Daemon会读取!后面的文件

```
1  */temp* 忽略PATH路径下一级子目录中以temp开头的文件和目录，如PAHT/A/temp.txt
2  */*/temp* 忽略PATH路径下二级子目录中以temp开头的文件和目录，如PATH/A/B/temp.txt
3  *.md
4  !README.md 忽略所有md文件，除了README.md
```

Dockerfile指令详解

指令	说明
FROM	设置镜像使用的基础镜像
MAINTAINER	设置镜像的作者
RUN	编译镜像时运行的脚本
CMD	设置容器的启动命令
LABEL	设置镜像的标签
EXPOESE	设置镜像暴露的端口
ENV	设置容器的环境变量
ADD	编译镜像时复制文件到镜像中
COPY	编译镜像时复制文件到镜像中
ENTRYPOINT	设置容器的入口程序
VOLUME	设置容器的挂载卷
USER	设置运行RUN CMD ENTRYPOINT的用户名
WORKDIR	设置RUN CMD ENTRYPOINT COPY ADD指令的工作目录
ARG	设置编译镜像时加入的参数
ONBUILD	设置镜像的ONBUILD指令
STOPSIGNAL	设置容器的退出信号量

使用 Dockerfile 定制镜像

镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是 `Dockerfile`。

`Dockerfile` 是一个文本文件，其内包含了一条条的 `指令`(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

此处以定制 nginx 镜像为例，使用 Dockerfile 来定制。

在一个空白目录中，建立一个文本文件，并命名为 `Dockerfile`：

```
1 $ mkdir mynginx
2 $ cd mynginx
3 $ touch Dockerfile
```

其内容为：

```
1 FROM nginx
2 RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
```

这个 Dockerfile 很简单，一共就两行。涉及到了两条指令，`FROM` 和 `RUN`。

Dockerfile 指令详解

指令：MAINTAINER
功能描述：设置镜像作者
语法：MAINTAINER < name>

指令：RUN
功能描述：
语法：RUN < command>
RUN [“executable” ,” param1” ,” param2”]
提示：RUN指令会生成容器，在容器中执行脚本，容器使用当前镜像，脚本指令完成后，Docker Daemon会将该容器提交为一个中间镜像，供后面的指令使用
补充：RUN指令第一种方式为shell方式，使用/bin/sh -c < command>运行脚本，可以在其中使用\将脚本分为多行
RUN指令第二种方式为exec方式，镜像中没有/bin/sh或者要使用其他shell时使用该方式，其不会调用shell命令
例子：RUN source $HOME/.bashrc;\
echo $HOME
RUN [“/bin/bash” ,” -c” ,” echo hello”]
RUN [“sh” ,” -c” ,” echo” ,” $HOME”] 使用第二种方式调用shell读取环境变量

指令：CMD
功能描述：设置容器的启动命令
语法：CMD [“executable” ,” param1” ,” param2”]
CMD [“param1” ,” param2”]
CMD < command>
提示：CMD第一种、第三种方式和RUN类似，第二种方式为ENTRYPOINT参数方式，为entrypoint提供参数列表
注意：Dockerfile中只能有一条CMD命令，如果写了多条则最后一条生效

Dockerfile 指令详解

指令：LABEL
功能描述：设置镜像的标签
延伸：镜像标签可以通过docker inspect查看
格式：LABEL < key>=< value> < key>=< value> ...
提示：不同标签之间通过空格隔开
注意：每条指令都会生成一个镜像层，Docker中镜像最多只能有127层，如果超出Docker Daemon就会报错，如LABEL ..=.. <假装这里有个换行>
LABEL ..=..合在一起用空格分隔就可以减少镜像层数量，同样，可以使用连接符\将脚本分为多行
镜像会继承基础镜像中的标签，如果存在同名标签则会覆盖

指令：EXPOSE
功能描述：设置镜像暴露端口，记录容器启动时监听哪些端口
语法：EXPOSE < port> < port> ...
延伸：镜像暴露端口可以通过docker inspect查看
提示：容器启动时，Docker Daemon会扫描镜像中暴露的端口，如果加入-P参数，Docker Daemon会把镜像中所有暴露端口导出，并为每个暴露端口分配一个随机的主机端口（暴露端口是容器监听端口，主机端口为外部访问容器的端口）
注意：EXPOSE只设置暴露端口并不导出端口，只有启动容器时使用-P/-p才导出端口，这个时候才能通过外部访问容器提供的服务

指令：ENV
功能描述：设置镜像中的环境变量
语法：ENV < key>=< value>...|< key> < value>
注意：环境变量在整个编译周期都有效，第一种方式可设置多个环境变量，第二种方式只设置一个环境变量
提示：通过${变量名}或者 $变量名使用变量，使用方式${变量名}时可以用${变量名:-default} ${变量名:+cover}设定默认值或者覆盖值
ENV设置的变量值在整个编译过程中总是保持不变的

Dockerfile 指令详解

指令：ADD
功能描述：复制文件到镜像中
语法：ADD < src>... < dest>|[“< src>”,... “< dest>”]
注意：当路径中有空格时，需要使用第二种方式
　　当src为文件或目录时，Docker Daemon会从编译目录寻找这些文件或目录，而dest为镜像中的绝对路径或者相对于WORKDIR的路径
提示：src为目录时，复制目录中所有内容，包括文件系统的元数据，但不包括目录本身
　　src为压缩文件，并且压缩方式为gzip,bzip2或xz时，指令会将其解压为目录
　　如果src为文件，则复制文件和元数据
　　如果dest不存在，指令会自动创建dest和缺失的上级目录

指令：COPY
功能描述：复制文件到镜像中
语法：COPY < src>... < dest>|[“< src>”,... “< dest>”]
提示：指令逻辑和ADD十分相似，同样Docker Daemon会从编译目录寻找文件或目录，dest为镜像中的绝对路径或者相对于WORKDIR的路径

指令：ENTRYPOINT
功能描述：设置容器的入口程序
语法：ENTRYPOINT [“executable”,” param1”,” param2”]
　　ENTRYPOINT command param1 param2（shell方式）
提示：入口程序是容器启动时执行的程序，docker run中最后的命令将作为参数传递给入口程序
　　入口程序有两种格式：exec、shell，其中shell使用 `/bin/sh -c` 运行入口程序，此时入口程序不能接收信号量
　　当Dockerfile有多条ENTRYPOINT时只有最后的ENTRYPOINT指令生效
　　如果使用脚本作为入口程序，需要保证脚本的最后一个程序能够接收信号量，可以在脚本最后使用exec或gosu启动传入脚本的命令
注意：通过shell方式启动入口程序时，会忽略CMD指令和docker run中的参数
　　为了保证容器能够接受docker stop发送的信号量，需要通过exec启动程序；如果没有加入exec命令，则在启动容器时容器会出现两个进程，并且使用docker stop命令容器无法正常退出（无法接受SIGTERM信号），超时后docker stop发送SIGKILL，强制停止容器
例子：FROM ubuntu <换行> ENTRYPOINT exec top -b

Dockerfile 指令详解

指令：VOLUME
功能描述：设置容器的挂载点
语法：VOLUME [“/data”]
　　VOLUME /data1 /data2
提示：启动容器时，Docker Daemon会新建挂载点，并用镜像中的数据初始化挂载点，可以将主机目录或数据卷容器挂载到这些挂载点

指令：USER
功能描述：设置RUN CMD ENTRYPOINT的用户名或UID
语法：USER < name>

指令：WORKDIR
功能描述：设置RUN CMD ENTRYPOINT ADD COPY指令的工作目录
语法：WORKDIR < Path>
提示：如果工作目录不存在，则Docker Daemon会自动创建
　　Dockerfile中多个地方都可以调用WORKDIR，如果后面跟的是相对位置，则会跟在上条WORKDIR指定路径后（如WORKDIR /A　WORKDIR B　WORKDIR C，最终路径为/A/B/C）

Dockerfile 指令详解

指令：ARG
功能描述：设置编译变量
语法：ARG < name>[=< defaultValue>]
注意：ARG从定义它的地方开始生效而不是调用的地方，在ARG之前调用编译变量总为空，在编译镜像时，可以通过docker build –build-arg < var>=< value>设置变量，如果var没有通过ARG定义则Daemon会报错
　　可以使用ENV或ARG设置RUN使用的变量，如果同名则ENV定义的值会覆盖ARG定义的值，与ENV不同，ARG的变量值在编译过程中是可变的，会对比使用编译缓存造成影响（ARG值不同则编译过程也不同）
例子：ARG CONT_IMAG_VER <换行> RUN echo $CONT_IMG_VER
　　ARG CONT_IMAG_VER <换行> RUN echo hello
　　当编译时给ARG变量赋值hello，则两个Dockerfile可以使用相同的中间镜像，如果不为hello，则不能使用同一个中间镜像

指令：ONBUILD
功能描述：设置自径想的编译钩子指令
语法：ONBUILD [INSTRUCTION]
提示：从该镜像生成子镜像，在子镜像的编译过程中，首先会执行父镜像中的ONBUILD指令，所有编译指令都可以成为钩子指令

指令：STOPSIGNAL
功能描述：设置容器退出时，Docker Daemon向容器发送的信号量
语法：STOPSIGNAL signal
提示：信号量可以是数字或者信号量的名字，如9或者SIGKILL，信号量的数字说明在Linux系统管理中有简单介绍

Dockerfile 指令详解

补充：ONBUILD流程

- 编译时，读取所有ONBUILD镜像并记录下来，在当前编译过程中不执行指令
- 生成镜像时将所有ONBUILD指令记录在镜像的配置文件OnBuild关键字中
- 子镜像在执行FROM指令时会读取基础镜像中的ONBUILD指令并顺序执行，如果执行过程中失败则编译中断；当所有ONBUILD执行成功后开始执行子镜像中的指令
- 子镜像不会继承基础镜像中的ONBUILD指令

补充：CMD ENTRYPOINT和RUN的区别

RUN指令是设置编译镜像时执行的脚本和程序，镜像编译完成后，RUN指令的生命周期结束

容器启动时，可以通过CMD和ENTRYPOINT设置启动项，其中CMD叫做容器默认启动命令，如果在docker run命令末尾添加command，则会替换镜像中CMD设置的启动程序；ENRTYPOINT叫做入口程序，不能被docker run命令末尾的command替换，而是将command当作字符串，传递给ENTRYPOINT作为参数

```
1 FROM ubuntu
2 ENTRYPOINT ["ps"]
3
4 //通过命令docker run --rm test启动容器，打印ps的输出
5 //通过命令docker run --rm test -ef启动容器，打印ps -ef的输出
```

在docker run中，可以通过–entrypoint替换镜像中的入口程序，在Dockerfile中，应该至少有一条CMD或者ENTRYPOINT指令，如果同时定义了CMD和ENTRYPOINT则CMD会作为参数传递给ENTRYPOINT

```
1 FROM ubuntu
2 ENTRYPOINT ["ps"]
3 CMD ["-ef"]
4
5 //通过命令docker run --rm test启动容器，打印ps -ef的输出
```

Dockerfile 指令详解

指令：FROM
功能描述：设置基础镜像
语法：FROM < image>[:< tag> | @< digest>]
提示：镜像都是从一个基础镜像（操作系统或其他镜像）生成，可以在一个Dockerfile中添加多条FROM指令，一次生成多个镜像 注意：如果忽略tag选项，会使用latest镜像

FROM 指定基础镜像

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。而 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。

在 Docker Store 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如nginx 、 redis 、 mongo 、 mysql 等；也有一些方便开发、构建、运行各种语言应用的镜像，如 node 、 openjdk 、 python 等。可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。

如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如ubuntu 、 debian 、 centos 等，这些操作系统的软件库为我们提供了更广阔的扩展空间。

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。

```
1 FROM scratch
2 ...
```

如果你以 `scratch` 为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。

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。

RUN 执行命令

`RUN` 指令是用来执行命令行命令的。由于命令行的强大能力， RUN 指令在定制镜像时是最常用的指令之一。其格式有两种：

- **shell 格式：** `RUN <命令>` ，就像直接在命令行中输入的命令一样。刚才写的 Dockerfile 中的 RUN 指令就是这种格式。

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

- **exec 格式：** `RUN ["可执行文件", "参数1", "参数2"]` ，这更像是函数调用中的格式。

既然 RUN 就像 Shell 脚本一样可以执行命令，那么我们是否就可以像 Shell 脚本一样把每个命令对应一个 RUN 呢？比如这样：

```
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/redis-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
```

之前说过，**`Dockerfile` 中每一个指令都会建立一层**， RUN 也不例外。每一个 RUN 的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后， commit 这一层的修改，构成新的镜像。

而上面的这种写法，**创建了 7 层镜像。**这是完全没有意义的，而且很多运行时不需要的东西，都被装进了镜像里，比如编译环境、更新的软件包等等。结果就是产生非常 `臃肿` 、非常多层的镜像，不仅仅增加了构建部署的时间，也很容易出错。 这是很多初学 Docker 的人常犯的一个错误（我也不能原谅自己ε=(´ο｀*)))唉）。

COPY 复制文件

COPY 复制文件

格式：

- `COPY <源路径>... <目标路径>`
- `COPY ["<源路径1>",... "<目标路径>"]`

和 RUN 指令一样，也有两种格式，一种类似于命令行，一种类似于函数调用。COPY 指令将从构建上下文目录中 < 源路径 > 的文件/目录复制到新的一层的镜像内的 < 目标路径 > 位置。比如：

```
COPY package.json /usr/src/app/
```

<源路径> 可以是多个，甚至可以是通配符，其通配符规则要满足 Go 的 filepath.Match 规则，如：

```
COPY hom* /mydir/
COPY hom?.txt /mydir/
```

<目标路径> 可以是容器内的绝对路径，也可以是相对于工作目录的相对路径（工作目录可以用 WORKDIR 指令来指定）。目标路径不需要事先创建，如果目录不存在会在复制文件前先行创建缺失目录。

此外，还需要注意一点，使用 COPY 指令，源文件的各种元数据都会保留。比如读、写、执行权限、文件变更时间等。这个特性对于镜像定制很有用。特别是构建相关文件都在使用 Git进行管理的时候。

ADD 更高级的复制文件

ADD 更高级的复制文件

ADD 指令和 COPY 的格式和性质基本一致。但是在 COPY 基础上增加了一些功能。**比如 <源路径> 可以是一个 URL**，这种情况下，Docker 引擎会试图去下载这个链接的文件放到 <目标路径> 去。**下载后的文件权限自动设置为 600**，如果这并不是想要的权限，那么还需要增加额外的一层 RUN 进行权限调整，另外，如果下载的是个压缩包，需要解压缩，也一样还需要额外的一层 RUN 指令进行解压缩。所以不如直接使用 RUN 指令，然后使用 wget 或者 curl 工具下载，处理权限、解压缩、然后清理无用文件更合理。因此，**这个功能其实并不实用，而且不推荐使用。**

如果 <源路径> 为一个 tar 压缩文件的话，压缩格式为 gzip，bzip2 以及 xz 的情况下，ADD 指令将会自动解压缩这个压缩文件到 <目标路径> 去。

在某些情况下，这个自动解压缩的功能非常有用，比如官方镜像 ubuntu 中：

```
FROM scratch
ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /
...
```

但在某些情况下，如果我们真的是希望复制个压缩文件进去，而不解压缩，这时就不可以使用 ADD 命令了。

在 Docker 官方的 Dockerfile 最佳实践文档 中要求，尽可能的使用 COPY，因为 COPY 的语义很明确，就是复制文件而已，而 ADD 则包含了更复杂的功能，其行为也不一定很清晰。最适合使用 ADD 的场合，就是所提及的需要自动解压缩的场合。

另外需要注意的是，**ADD 指令会令镜像构建缓存失效，从而可能会令镜像构建变得比较缓慢。**

因此在 COPY 和 ADD 指令中选择的时候，可以遵循这样的原则，**所有的文件复制均使用 COPY 指令，仅在需要自动解压缩的场合使用 ADD**。

CMD容器启动命令

CMD 容器启动命令

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"...]`
- **参数列表格式：** `CMD ["参数1", "参数2"...]` 。在指定了 `ENTRYPOINT` 指令后，用 `CMD` 指定具体的参数。

之前介绍容器的时候曾经说过，**Docker 不是虚拟机，容器就是进程。**既然是进程，那么在启动容器的时候，需要指定所运行的程序及参数。 CMD 指令就是用于指定默认的容器主进程的启动命令的。

在运行时可以指定新的命令来替代镜像设置中的这个默认命令，比如， ubuntu 镜像默认的CMD 是 `/bin/bash` ，如果我们直接 `docker run -it ubuntu` 的话，会直接进入 `bash` 。我们也可以在运行时指定运行别的命令，如 `docker run -it ubuntu cat /etc/os-release` 。这就是用 cat /etc/os-release 命令替换了默认的 /bin/bash 命令了，输出了系统版本信息。

在指令格式上，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。

如果使用 shell 格式的话，实际的命令会被包装为 `sh -c` 的参数的形式进行执行。比如：

```
CMD echo $HOME
```

在实际执行中，会将其变更为：

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

这就是为什么我们可以使用环境变量的原因，因为这些环境变量会被 shell 进行解析处理。提到 CMD 就不得不提容器中应用在前台执行和后台执行的问题。这是初学者常出现的一个混淆。

Docker 不是虚拟机，容器中的应用都应该以前台执行，而不是像虚拟机、物理机里面那样，用 `upstart/systemd` 去启动后台服务，容器内没有后台服务的概念。

初学者一般将 CMD 写为：

```
CMD service nginx start
```

然后发现容器执行后就立即退出了。甚至在容器内去使用 `systemctl` 命令结果却发现根本执行不了。这就是因为没有搞明白前台、后台的概念，没有区分容器和虚拟机的差异，依旧在以传统虚拟机的角度去理解容器。

对于容器而言，其启动程序就是容器应用进程，容器就是为了主进程而存在的，主进程退出，容器就失去了存在的意义，从而退出，其它辅助进程不是它需要关心的东西。

而使用 `service nginx start` 命令，则是希望 `systemd` 来以后台守护进程形式启动 nginx 服务。而刚才说了 `CMD service nginx start` 会被理解为 CMD ["sh", "-c", "service nginxstart"]，**因此主进程实际上是 sh 。那么当 service nginx start 命令结束后， sh 也就结束了， sh 作为主进程退出了，自然就会令容器退出。**

正确的做法是**直接执行 nginx 可执行文件**，并且要求以前台形式运行。比如：

```
CMD ["nginx", "-g", "daemon off;"]
```

ENTRYPOINT 入口点

ENTRYPOINT 入口点

`ENTRYPOINT` 的格式和 RUN 指令格式一样，分为 `exec 格式`和 `shell 格式`。

ENTRYPOINT 的目的和 CMD 一样，都是在指定容器启动程序及参数。ENTRYPOINT 在运行时也可以替代，不过比 CMD 要略显繁琐，需要通过 docker run 的参数 –entrypoint 来指定。

当指定了 ENTRYPOINT 后， CMD 的含义就发生了改变，不再是直接的运行其命令，而是将CMD 的内容作为参数传给 ENTRYPOINT 指令，换句话说实际执行时，将变为：

```
<ENTRYPOINT> "<CMD>"
```

那么有了 CMD 后，为什么还要有 ENTRYPOINT 呢？这种 `<ENTRYPOINT> "<CMD>"` 有什么好处么？让我们来看几个场景。

场景一：让镜像变成像命令一样使用

假设我们需要一个得知自己当前公网 IP 的镜像，那么可以先用 CMD 来实现：

```
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" ]
```

假如我们使用 `docker build -t myip .` 来构建镜像的话，如果我们需要查询当前公网 IP，只需要执行：

```
$ docker run myip
当前 IP：61.148.226.66 来自：北京市 联通
```

场景二：应用运行前的准备工作

启动容器就是启动主进程，但有些时候，启动主进程前，需要一些准备工作。比如 mysql 类的数据库，可能需要一些数据库配置、初始化的工作，这些工作要在最终的 mysql 服务器运行之前解决。

此外，可能希望避免使用 root 用户去启动服务，从而提高安全性，而在启动服务前还需要以 root 身份执行一些必要的准备工作，最后切换到服务用户身份启动服务。或者除了服务外，其它命令依旧可以使用 root 身份执行，方便调试等。

这些准备工作是和容器 CMD 无关的，无论 CMD 为什么，都需要事先进行一个预处理的工作。这种情况下，可以写一个脚本，然后放入 ENTRYPOINT 中去执行，而这个脚本会将接到的参数（也就是 ）作为命令，在脚本最后执行。比如官方镜像 redis 中就是这么做的：

```
FROM alpine:3.4
…
RUN addgroup -S redis && adduser -S -G redis redis
…
ENTRYPOINT ["docker-entrypoint.sh"]
EXPOSE 6379
CMD [ "redis-server" ]
```

可以看到其中为了 redis 服务创建了 redis 用户，并在最后指定了 ENTRYPOINT 为 dockerentrypoint.sh 脚本。

```
#!/bin/sh
...
# allow the container to be started with `--user`
if [ "$1" = 'redis-server' -a "$(id -u)" = '0' ]; then
    chown -R redis .
    exec su-exec redis "$0" "$@"
fi
exec "$@"
```

ENV 设置环境变量

ENV 设置环境变量

格式有两种：

- `ENV <key> <value>`
- `ENV <key1>=<value1> <key2>=<value2>...`

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。

```
ENV VERSION=1.0 DEBUG=on \
    NAME="Happy Feet"
```

这个例子中演示了如何换行，以及对含有空格的值用双引号括起来的办法，这和 Shell 下的行为是一致的。

定义了环境变量，那么在后续的指令中，就可以使用这个环境变量。比如在官方 node 镜像 Dockerfile 中，就有类似这样的代码：

```
ENV NODE_VERSION 7.2.0
RUN curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-linux-x64.ta
r.xz" \
    && curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
    && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION-linux-x64.tar.xz\$" SHASUMS256.txt | sha256sum -c - \
    && tar -xJf "node-v$NODE_VERSION-linux-x64.tar.xz" -C /usr/local --strip-components=1 \
    && rm "node-v$NODE_VERSION-linux-x64.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt \
    && ln -s /usr/local/bin/node /usr/local/bin/nodejs
```

在这里先定义了环境变量 NODE_VERSION ，其后的 RUN 这层里，多次使用 $NODE_VERSION 来进行操作定制。可以看到，将来升级镜像构建版本的时候，只需要更新 7.2.0 即可， Dockerfile 构建维护变得更轻松了。

下列指令可以支持环境变量展开：
ADD 、 COPY 、 ENV 、 EXPOSE 、 LABEL 、 USER 、 WORKDIR 、 VOLUME 、 STOPSIGNAL 、 ONBUILD 。

可以从这个指令列表里感觉到，环境变量可以使用的地方很多，很强大。通过环境变量，我们可以让一份 Dockerfile 制作更多的镜像，只需使用不同的环境变量即可。

ARG 构建参数

ARG 构建参数

格式： `ARG <参数名>[=<默认值>]`

构建参数和 ENV 的效果一样，都是设置环境变量。所不同的是， ARG 所设置的构建环境的环境变量，在将来容器运行时是不会存在这些环境变量的。但是不要因此就使用 ARG 保存密码之类的信息，因为 `docker history` 还是可以看到所有值的。

Dockerfile 中的 ARG 指令是定义参数名称，以及定义其默认值。该默认值可以在构建命令 docker build 中用 `--build-arg <参数名>=<值>` 来覆盖。

在 1.13 之前的版本，要求 –build-arg 中的参数名，必须在 Dockerfile 中用 ARG 定义过了，换句话说，就是 –build-arg 指定的参数，必须在 Dockerfile 中使用了。如果对应参数没有被使用，则会报错退出构建。从 1.13 开始，这种严格的限制被放开，不再报错退出，而是显示警告信息，并继续构建。这对于使用 CI 系统，用同样的构建流程构建不同的 Dockerfile 的时候比较有帮助，避免构建命令必须根据每个 Dockerfile 的内容修改。

VOLUME 定义匿名卷

VOLUME 定义匿名卷

格式为：

- `VOLUME ["<路径1>", "<路径2>"...]`
- `VOLUME <路径>`

之前我们说过，容器运行时应该尽量保持容器存储层不发生写操作，对于数据库类需要保存动态数据的应用，其数据库文件应该保存于卷(volume)中。为了防止运行时用户忘记将动态文件所保存目录挂载为卷，在 Dockerfile 中，我们可以事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

```
1 VOLUME /data
```

这里的 `/data` 目录就会在运行时自动挂载为匿名卷，任何向 /data 中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化。当然，运行时可以覆盖这个挂载设置。比如：

```
1 docker run -d -v mydata:/data xxxx
```

在这行命令中，就使用了 mydata 这个命名卷挂载到了 `/data` 这个位置，替代了 Dockerfile 中定义的匿名卷的挂载配置。

Maven结合Dockerfile示例

Maven pom.xml 的build部分节选

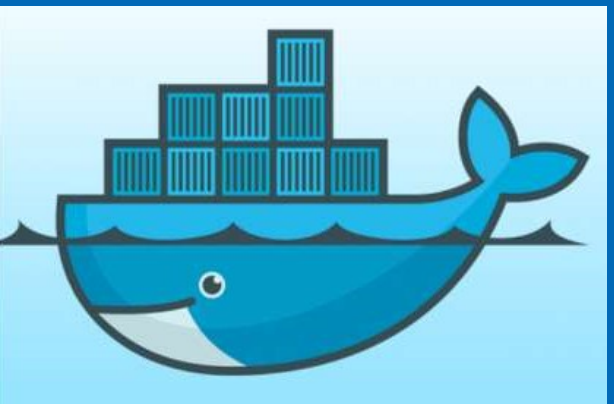
```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>1.2.0</version>
    <configuration>
        <imageName>${docker.image.prefix}/${project.name}</imageName>
        <dockerHost>http://[redacted]:2375</dockerHost>
        <!--指定Dockerfile所在的路径-->
        <dockerDirectory>${basedir}/src/main/resources/docker</dockerDirectory>
        <buildArgs>
            <!--提供参数向Dockerfile传递-->
            <WAR_FILE>${project.name}-${project.version}.war</WAR_FILE>
            <PROJECT_NAME>${project.name}.war</PROJECT_NAME>
        </buildArgs>
        <imageTags>
            <imageTag>0.0.1</imageTag>
            <!--可以指定多个标签-->
            <imageTag>latest</imageTag>
        </imageTags>
        <!--覆盖已存在的标签_镜像-->
        <forceTags>true</forceTags>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>
                <include>${project.name}-${project.version}.war</include>
            </resource>
        </resources>
    </configuration>
</plugin>
```

Dockerfile文件

```
FROM tomcat:8.5.41-jdk8

ENV CATALINA_HOME /usr/local/tomcat
ENV PATH $CATALINA_HOME/bin:$PATH

RUN set -x

RUN rm -rf ${CATALINA_HOME}/webapps/manager ; \
   rm -rf ${CATALINA_HOME}/webapps/host-manager ; \
   rm -rf ${CATALINA_HOME}/webapps/examples ; \
   rm -rf ${CATALINA_HOME}/webapps/docs;

ARG WAR_FILE
ARG PROJECT_NAME
copy ${WAR_FILE}
${CATALINA_HOME}/webapps/${PROJECT_NAME}
EXPOSE 8080 8443
CMD ["/usr/local/tomcat/bin/catalina.sh", "run"]
```

谢谢

surpass_li@aliyun.com
https://www.easyolap.cn/



作者：李在超
擅长：Java程序员, 了解Devops, 捣鼓 linux和Go