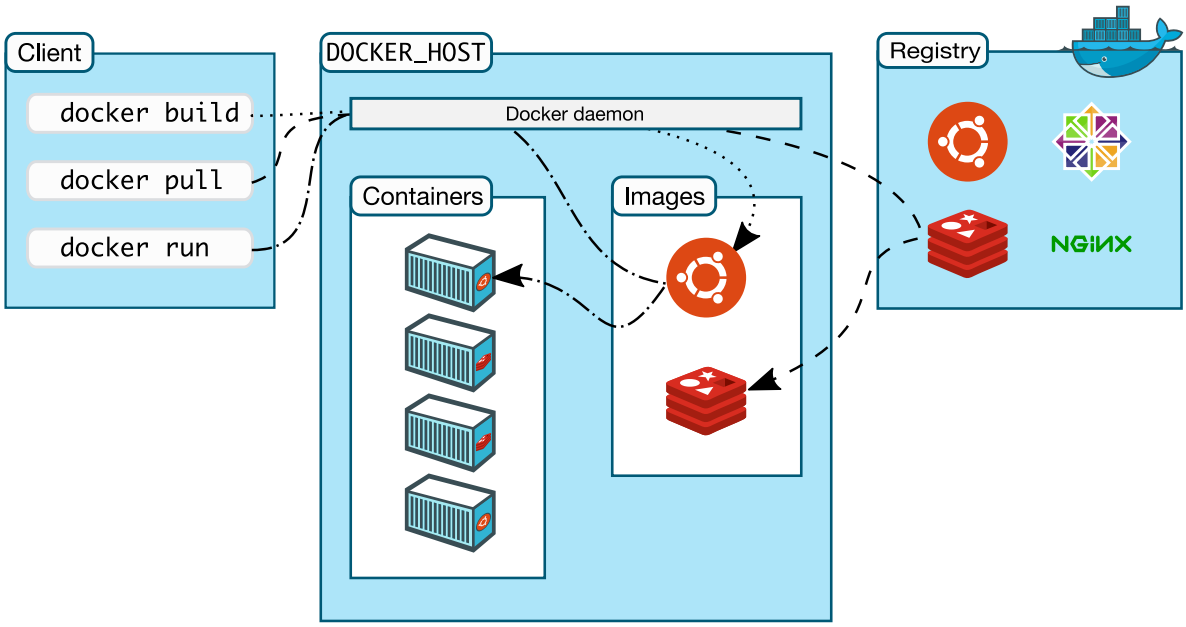
Docker实战

一、基本概念

1、Docker架构



K8S：CRI（Container Runtime Interface）

Client： 客户端；操作docker服务器的客户端（命令行或者界面）

Docker_Host：Docker主机；安装Docker服务的主机

Docker_Daemon：后台进程；运行在Docker服务器的后台进程

Containers：容器；在Docker服务器中的容器（一个容器一般是一个应用实例，容器间互相隔离）

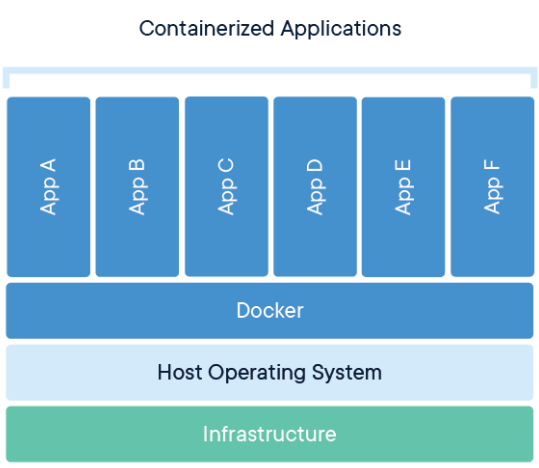
Images：镜像、映像、程序包；Image是只读模板，其中包含创建Docker容器的说明。容器是由Image运行而来，Image固定不变。

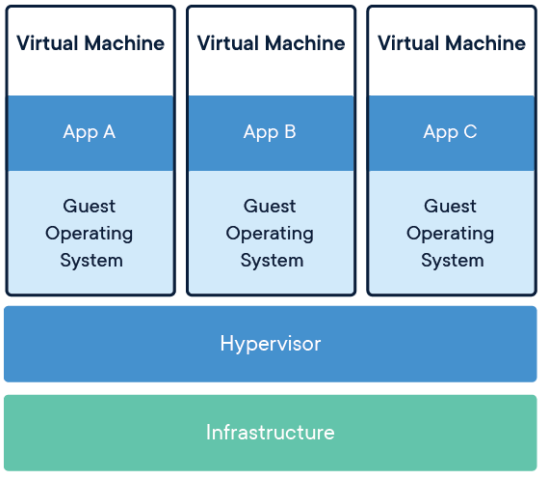
Registries：仓库；存储Docker Image的地方。官方远程仓库地址： https://hub.docker.com/search

> Docker用Go编程语言编写，并利用Linux内核的多种功能来交付其功能。 Docker使用一种称为名称空间的技术来提供容器的隔离工作区。 运行容器时，Docker会为该容器创建一组名称空间。 这些名称空间提供了一层隔离。 容器的每个方面都在单独的名称空间中运行，并且对其的访问仅限于该名称空间。

Docker	面向对象
镜像（Image）	类
容器（Container）	对象（实例）

- 容器与虚拟机





2、Docker隔离原理

- ***namespace 6项隔离***（资源隔离）

namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机和域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等
Mount	CLONE_NEWNS	挂载点(文件系统)
User	CLONE_NEWUSER	用户和用户组

- ***cgroups资源限制***（资源限制）

cgroup提供的主要功能如下：

- 资源限制：限制任务使用的资源总额，并在超过这个 配额 时发出提示
- 优先级分配：分配CPU时间片数量及磁盘IO带宽大小、控制任务运行的优先级
- 资源统计：统计系统资源使用量，如CPU使用时长、内存用量等
- 任务控制：对任务执行挂起、恢复等操作

> cgroup资源控制系统，每种子系统独立地控制一种资源。功能如下

子系统	功能
cpu	使用调度程序控制任务对CPU的使用。
cpuacct(CPU Accounting)	自动生成cgroup中任务对CPU资源使用情况的报告。
cpuset	为cgroup中的任务分配独立的CPU(多处理器系统时)和内存。
devices	开启或关闭cgroup中任务对设备的访问
freezer	挂起或恢复cgroup中的任务
memory	设定cgroup中任务对内存使用量的限定，并生成这些任务对内存资源使用情况的报告
perf_event(Linux CPU性能探测器)	使cgroup中的任务可以进行统一的性能测试
net_cls(Docker未使用)	通过等级识别符标记网络数据包，从而允许Linux流量监控程序(Traffic Controller)识别从具体cgroup中生成的数据包

3、 Docker安装

> 以下以centos为例；
> 更多其他安装方式，详细参照文档： https://docs.docker.com/engine/install/centos/

1、移除旧版本

```
sudo yum remove docker*
```

2、设置docker yum源

```
sudo yum install -y yum-utils
sudo yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
#此处可以百度 docker yum aliyun 切换为ali的yum源
```

3、安装最新docker engine

```
sudo yum install docker-ce docker-ce-cli containerd.io
```

4、安装指定版本docker engine

1、在线安装

```
#找到所有可用docker版本列表
yum list docker-ce --showduplicates | sort -r


# 安装指定版本，用上面的版本号替换<VERSION_STRING>
sudo yum install docker-ce-<VERSION_STRING>.x86_64 docker-ce-cli-<VERSION_STRING>.x86_64 containerd.io
#例如：
#yum install docker-ce-3:20.10.5-3.el7.x86_64  docker-ce-cli-3:20.10.5-3.el7.x86_64 containerd.io
#注意加上 .x86_64 大版本号
```

2、离线安装

https://download.docker.com/linux/centos/7/x86_64/stable/Packages/

```
rpm -ivh xxx.rpm

可以下载 tar
解压启动即可

```

https://docs.docker.com/engine/install/binaries/#install-daemon-and-client-binaries-on-linux

5、启动服务

```
systemctl start docker
systemctl enable docker
```

6、镜像加速

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://82m9ar63.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker

#以后docker下载直接从阿里云拉取相关镜像
```

> /etc/docker/daemon.json 是Docker的核心配置文件。

7、可视化界面-Portainer

1、什么是Portainer

https://documentation.portainer.io/

Portainer社区版2.0拥有超过50万的普通用户，是功能强大的开源工具集，可让您轻松地在Docker，Swarm，Kubernetes和Azure ACI中构建和管理容器。 Portainer的工作原理是在易于使用的GUI后面隐藏使管理容器变得困难的复杂性。通过消除用户使用CLI，编写YAML或理解清单的需求，Portainer使部署应用程序和解决问题变得如此简单，任何人都可以做到。 Portainer开发团队在这里为您的Docker之旅提供帮助；

2、安装

```
# 服务端部署
docker run -d -p 8000:8000 -p 9000:9000 --name=portainer --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce
# 访问 9000 端口即可

#agent端部署
docker run -d -p 9001:9001 --name portainer_agent --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/docker/volumes:/var/lib/docker/volumes portainer/agent
```

二、命令复习

1、常见命令

所有Docker命令手册
https://docs.docker.com/engine/reference/commandline/docker/

命令	作用
`attach`	绑定到运行中容器的 标准输入, 输出,以及错误流（这样似乎也能进入容器内容，但是一定小心，他们操作的就是控制台，控制台的退出命令会生效，比如redis,nginx...）
`build`	从一个 Dockerfile 文件构建镜像
`commit`	把容器的改变 提交创建一个新的镜像
`cp`	容器和本地文件系统间 复制 文件/文件夹
`create`	创建新容器，但并不启动（注意与docker run 的区分）需要手动启动。start\stop
`diff`	检查容器里文件系统结构的更改【A：添加文件或目录 D：文件或者目录删除 C：文件或者目录更改】
events	获取服务器的实时事件
`exec`	在运行时的容器内运行命令
`export`	导出**容器**的文件系统为一个tar文件。commit是直接提交成镜像，export是导出成文件方便传输
history	显示镜像的历史
`images`	列出所有镜像
`import`	导入tar的内容创建一个镜像，再导入进来的镜像直接启动不了容器。 /docker-entrypoint.sh nginx -g 'daemon off;' docker ps --no-trunc 看下之前的完整启动命令再用他
info	显示系统信息
`inspect`	获取docker对象的底层信息
`kill`	杀死一个或者多个容器
`load`	从 tar 文件加载镜像
`login`	登录Docker registry
`logout`	退出Docker registry
`logs`	获取容器日志；容器以前在前台控制台能输出的所有内容，都可以看到
`pause`	暂停一个或者多个容器
port	列出容器的端口映射
`ps`	列出所有容器
`pull`	从registry下载一个image 或者repository
`push`	给registry推送一个image或者repository
rename	重命名一个容器
`restart`	重启一个或者多个容器

命令	作用
`rm`	移除一个或者多个容器
`rmi`	移除一个或者多个镜像
run	创建并启动容器
`save`	把一个或者多个**镜像**保存为tar文件
search	去docker hub寻找镜像
`start`	启动一个或者多个容器
stats	显示容器资源的实时使用状态
`stop`	停止一个或者多个容器
`tag`	给源镜像创建一个新的标签，变成新的镜像
top	显示正在运行容器的进程
`unpause`	pause的反操作
update	更新一个或者多个docker容器配置
`version`	Show the Docker version information
container	管理容器
`image`	管理镜像
network	管理网络
volume	管理卷

根据正在运行的容器制作出相关的镜像：反向

根据镜像启动一个容器：正向

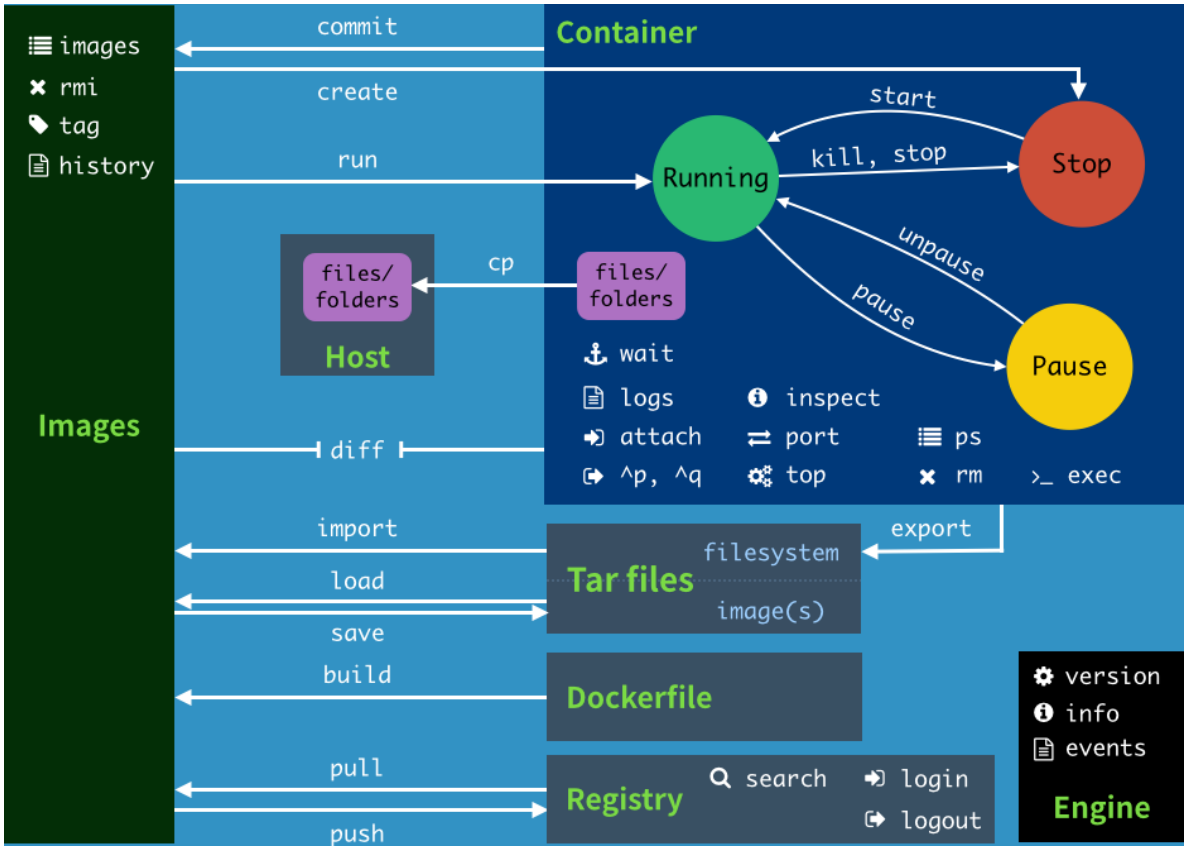
有了Docker：

1、先去软件市场搜镜像： https://registry.hub.docker.com/ docker hub

2、下载镜像 docker pull xxx

3、启动软件 docker run 镜像名；

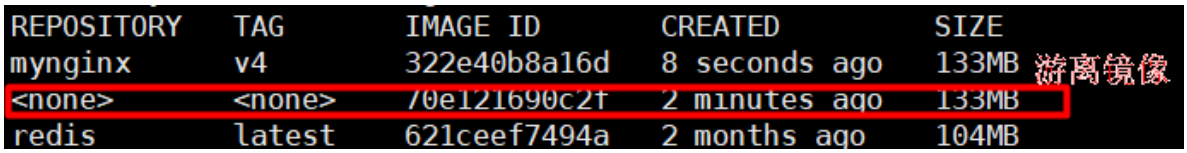
对于镜像的所有管理操作都在这一个命令：docker image --help




```
docker pull redis == docker pull redis:latest（最新版）
# 阿里云的镜像是从docker hub来的，我们配置了加速，默认是从阿里云（缓存）下载

REPOSITORY（名）    TAG （标签）         IMAGE ID（镜像id）   CREATED（镜像的创建时间）
SIZE
redis           5.0.12-alpine3.13   50ae27fed589       6 days ago
 29.3MB
redis           latest              621ceef7494a       2 months ago
104MB


# 镜像是怎么做成的。基础环境+软件
redis的完整镜像应该是：  linux系统+redis软件
alpine：超级经典版的linux 5mb；+ redis = 29.0mb
没有alpine3的：就是centos基本版

# 以后自己选择下载镜像的时候尽量使用
alpine：  slim：

docker rmi -f $(docker images -aq) #删除全部镜像
docker image prune #移除游离镜像  dangling：游离镜像（没有镜像名字的）


docker tag 原镜像:标签  新镜像名:标签  #重命名


docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
docker create [设置项] 镜像名 [启动] [启动参数...]
docker create redis: 按照redis:latest镜像启动一个容器

docker kill是强制kill -9（直接拔电源）；
docker stop可以允许优雅停机(当前正在运行中的程序处理完所有事情后再停止)

```

```
docker create --name myredis -p 6379（主机的端口）:6379（容器的端口） redis

-p  port1:port2
port1是必须唯一的，那个是没关系的。


docker run --name myredis2 -p 6379:6379  -p 8888:6379 redis  ：默认是前台启动的，一般加上-d 让他后台悄悄启动，  虚拟机的很多端口绑定容器的一个端口是允许的
docker run -d == docker create + docker start



#启动了nginx；一个容器。docker 容器里面安装了nginx，要对nginx的所有修改都要进容器
#进容器：
docker attach 绑定的是控制台. 可能导致容器停止。不要用这个

docker exec -it -u 0:0 --privileged mynginx4 /bin/bash：  0用户，以特权方式进入容器


docker container inspect  容器名 = docker inspect 容器名
docker inspect image/network/volume ....


# 一般运行中的容器会常年修改，我们要使用最终的新镜像
docker commit -a leifengyang -m "first commit" mynginx4 mynginx:v4

#把新的镜像放到远程docker hub，方便后来在其他机器下载


#---------export操作容器/import-------------------
docker export导出的文件被import导入以后变成镜像，并不能直接启动容器，需要知道之前的启动命令（docker ps --no-trunc），然后再用下面启动。
docker run -d -P mynginx:v6 /docker-entrypoint.sh nginx -g 'daemon off;'

或者docker image inspect 看之前的镜像，把 之前镜像的 Entrypoint的所有和 Cmd的连接起来就能得到启动命令

#----save/load--操作镜像--
docker save -o busybox.tar busybox:latest  把busybox镜像保存成tar文件
docker load -i busybox.tar     把压缩包里面的内容直接导成镜像

#----------
镜像为什么能长久运行
镜像启动一定得有一个阻塞的进程，一直干活，在这里代理。
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
docker run --name myredis2 -p 6379:6379  -p 8888:6379 redis
镜像启动以后做镜像里面默认规定的活。
docker run -it busybox; 交互模式进入当前镜像启动的容器

---------


#----产生镜像-----
1、基于已经存在的容器，提取成镜像
2、人家给了我tar包，导入成镜像
3、做出镜像
   -1)、准备一个文件Dockerfile
FROM busybox
```

```
CMD ping baidu.com
   -2)、编写Dockerfile
   -3)、构建镜像
   docker build -t mybusy66:v6 -f Dockerfile .


#---做redis的镜像---
FROM alpine（基础镜像）
//下载安装包
//解压
//准备配置文件
CMD  redis-server redis.conf


#----------
build  是根据一个Dockerfile构建出镜像
commit 是正在运行中的容器提交成一个镜像
```



- 容器的状态
  - Created（新建）、Up（运行中）、Pause（暂停）、Exited（退出）
- docker run的立即启动，docker create得稍后自己启动
- 推送镜像
  - 注册docker hub并登录
  - 可以创建一个仓库，选为public
  - docker push leifengyang/mynginx:tagname
  - docker hub一个完整镜像的全路径是
  - **docker.io/library/**redis:alpine3.13 我们的 docker.io/leifengyang/mynginx:tagname
  - docker images的时候镜像缩略了全名 默认官方镜像没有docker.io/library/
  - **docker.io/** `rediscommander` / `redis-commander:latest`
  - docker.io/leifengyang/mynginx:v4 我的镜像的全称
  - 登录远程docker仓库
  - 当前会话登录以后 docker login 。所有的东西都会push到这个人的仓库
  - docker push leifengyang/mynginx:tagname
  - 上面命令的完整版 docker push docker.io/leifengyang/mynginx:v4
  - 怎么知道是否登录了 cat ~/.docker/config.json 有没有 auth的值，没有就是没有登录
- docker hub太慢了，用阿里云的镜像仓库，或者以后的habor仓库
  - ```
    sudo docker tag [ImageId] registry.cn-hangzhou.aliyuncs.com/lfy/mynginx:[镜像版sudo docker push registry.cn-hangzhou.aliyuncs.com/lfy/mynginx:[镜像版本号]

    仓库网址/名称空间(lfy/leifengyang)/仓库名:版本号
    ```

2、典型命令

1、docker run

常用关键参数 `OPTIONS` 说明：
    ```

- `-d: 后台运行容器，并返回容器ID；`
- `-i: 以交互模式运行容器，通常与 -t 同时使用；`
- `-P: 随机端口映射，容器内部端口随机映射到主机的端口`
- `-p:指定端口映射，格式为：主机(宿主)端口:容器端口`
- `-t: 为容器重新分配一个伪输入终端，通常与 -i 同时使用`
- `--name="nginx-lb":为容器指定一个名称；`
- **--dns 8.8.8.8: 指定容器使用的DNS服务器，默认和宿主一致；**
- **--dns-search example.com: 指定容器DNS搜索域名，默认和宿主一致；**
- **-h "mars": 指定容器的hostname；**
- `-e username="ritchie": 设置环境变量；`
- **--env-file=[]: 从指定文件读入环境变量；**
- `--cpuset="0-2" or --cpuset="0,1,2": 绑定容器到指定CPU运行；`
- `-m :设置容器使用内存最大值；`
- `--net="bridge": 指定容器的网络连接类型，支持 bridge/host/none/container: 四种类型；`
- **--link=[]: 添加链接到另一个容器；**
- **--expose=[]: 开放一个端口或一组端口；**
- `--restart , 指定重启策略，可以写--restart=awlays 总是故障重启`
- `--volume , -v: 绑定一个卷。一般格式 主机文件或文件夹:虚拟机文件或文件夹`

> -v: 在存储章节详细解释
>
> --net: 在网络章节详细解析

0、如何使用Docker部署组件

1、先去找组件的镜像

2、查看镜像文档，了解组件的可配置内容

3、docker run进行部署

常见部署案例

1、部署Nginx

```
# 注意  外部的/nginx/conf下面的内容必须存在，否则挂载会覆盖
docker run --name nginx-app \
-v /app/nginx/html:/usr/share/nginx/html:ro \
-v /app/nginx/conf:/etc/nginx
-d nginx
```

2、部署MySQL

```
# 5.7版本
docker run -p 3306:3306 --name mysql57-app \
-v /app/mysql/log:/var/log/mysql \
-v /app/mysql/data:/var/lib/mysql \
-v /app/mysql/conf:/etc/mysql/conf.d \
-e MYSQL_ROOT_PASSWORD=123456 \
-d mysql:5.7

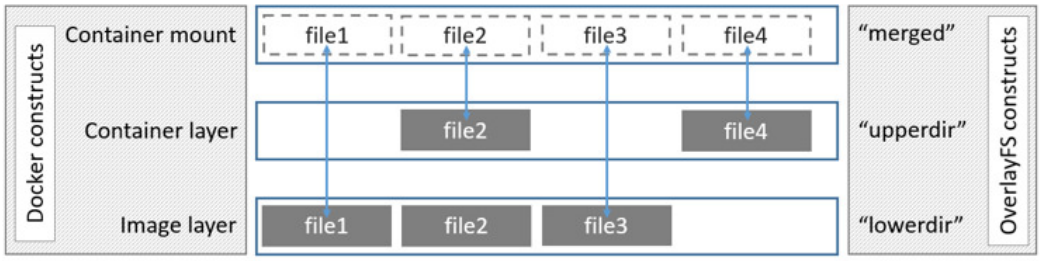
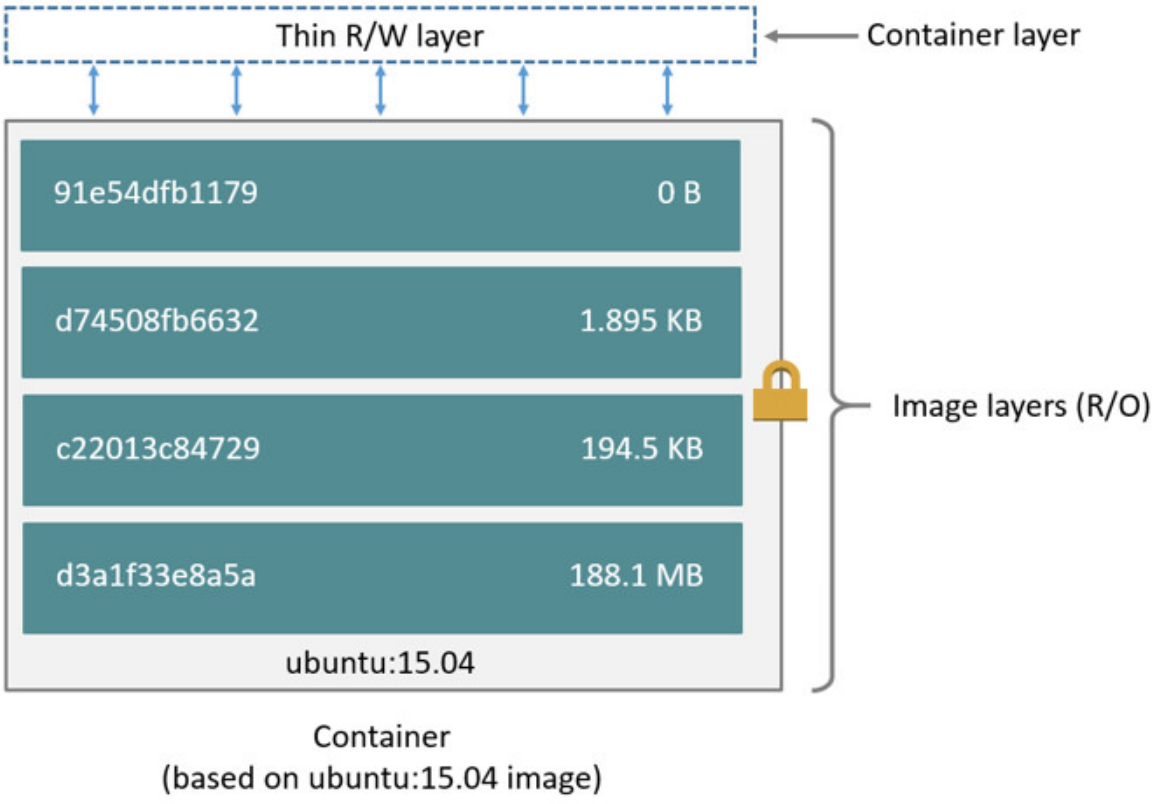
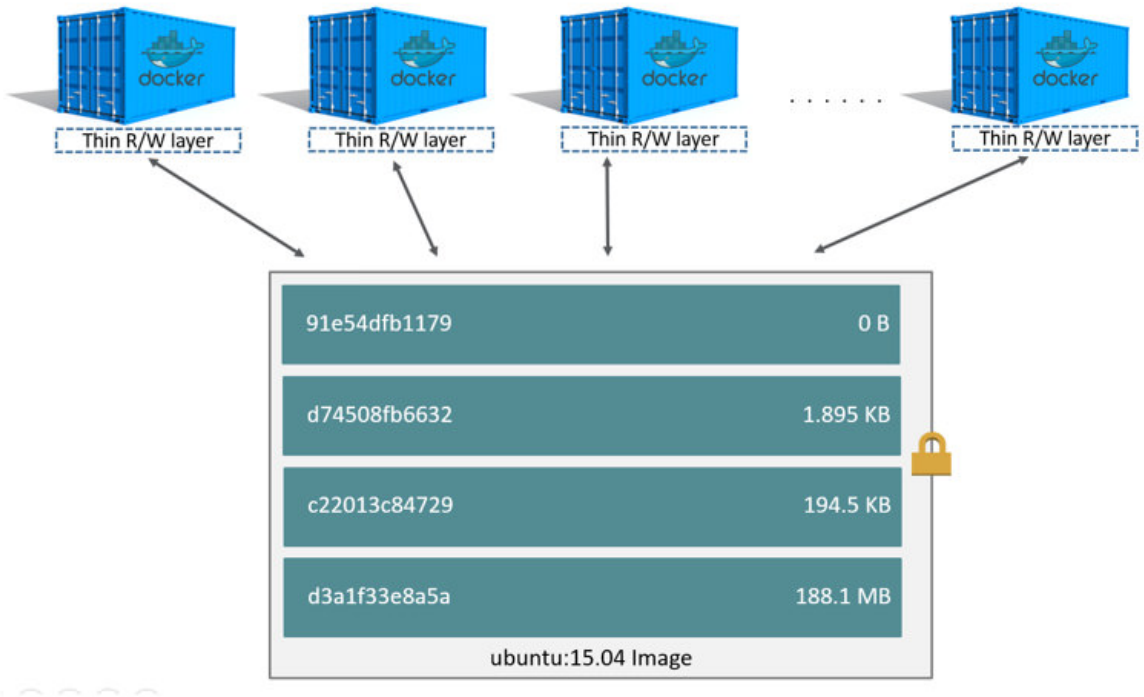
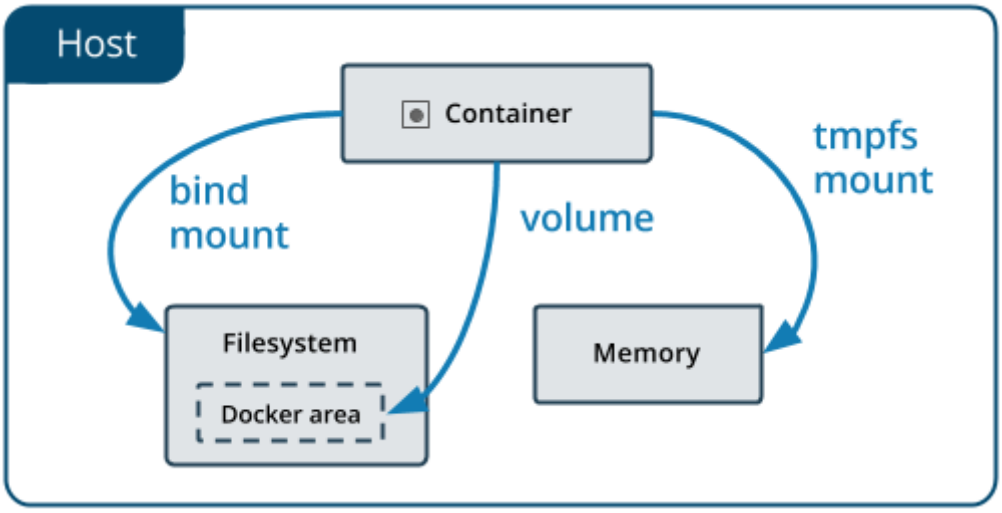
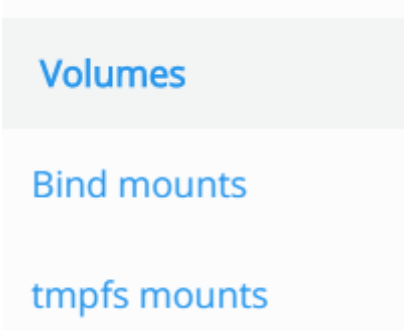
#8.x版本,引入了 secure-file-priv 机制，磁盘挂载将没有权限读写data数据，所以需要将权限透传，或者chmod -R 777 /app/mysql/data
```

```
# --privileged 特权容器，容器内使用真正的root用户
docker run -p 3306:3306 --name mysql8-app \
-v /app/mysql/conf:/etc/mysql/conf.d \
-v /app/mysql/log:/var/log/mysql \
-v /app/mysql/data:/var/lib/mysql \
-e MYSQL_ROOT_PASSWORD=123456 \
--privileged \
-d mysql
```

3、部署Redis

```
# 提前准备好redis.conf文件，创建好相应的文件夹。如：
port 6379
appendonly yes
#更多配置参照 https://raw.githubusercontent.com/redis/redis/6.0/redis.conf

docker run -p 6379:6379 --name redis \
-v /app/redis/redis.conf:/etc/redis/redis.conf \
-v /app/redis/data:/data \
-d redis:6.2.1-alpine3.13 \
redis-server /etc/redis/redis.conf --appendonly yes
```

4、部署ElasticSearch

```
#准备文件和文件夹，并chmod -R 777 xxx
#配置文件内容，参照
https://www.elastic.co/guide/en/elasticsearch/reference/7.5/node.name.html 搜索相关配置
# 考虑为什么挂载使用esconfig ...

docker run --name=elasticsearch -p 9200:9200 -p 9300:9300 \
-e "discovery.type=single-node" \
-e ES_JAVA_OPTS="-Xms300m -Xmx300m" \
-v /app/es/data:/usr/share/elasticsearch/data \
-v /app/es/plugins:/usr/shrae/elasticsearch/plugins \
-v esconfig:/usr/share/elasticsearch/config \
-d elasticsearch:7.12.0
```

5、部署Tomcat

```
# 考虑，如果我们每次 -v 都是指定磁盘路径，是不是很麻烦？
docker run --name tomcat-app -p 8080:8080 \
-v tomcatconf:/usr/local/tomcat/conf \
-v tomcatwebapp:/usr/local/tomcat/webapps \
-d tomcat:jdk8-openjdk-slim-buster
```

6、重启策略

no，默认策略，在容器退出时不重启容器
on-failure，在容器非正常退出时（退出状态非0），才会重启容器
on-failure:3，在容器非正常退出时重启容器，最多重启3次
always，在容器退出时总是重启容器

unless-stopped，在容器退出时总是重启容器，但是不考虑在Docker守护进程启动时就已经停止了的容器

2、docker exec

```
docker exec -it alpine sh
```

3、docker build

```
docker build -t imageName -f DockerfileName .
```

4、docker push

三、网络和存储原理

问题：

- 容器： 某个软件完整的运行环境；包含了一个小型的linux系统
- 宿主机里面同时4个nginx； 一个nginx运行时完整环境有20MB？
  - 4个nginx 合起来占用多少的磁盘空间
  - 80？一定会很大....

docker装的和宿主机的优缺点：

优点：docker的移植性、便捷性高于在宿主机部署、进程隔离、很方便的资源限制

缺点：docker虚拟化技术，损失不到3%的性能。

docker？原生物理机自己造docker这种东西。

镜像：容器；

镜像（Image）：固定不变的。一个镜像可以启动很多容器

容器（Container）：文件系统可能logs经常变化的，一个镜像可以启动很多容器。

docker在底层使用自己的存储驱动。来组件文件内容 storage drivers。docker 基于 AUFS（联合文件系统）；

1、Docker存储

1、镜像如何存储

0、自己探索

FROM busybox
CMD ping baidu.com

截取的nginx的分层

```
[root@lfy ~]# docker history nginx
IMAGE          CREATED        CREATED BY                                      SIZE      COMMENT
f6d0b4767a6c   2 months ago   /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon…   0B
<missing>      2 months ago   /bin/sh -c #(nop)  STOPSIGNAL SIGQUIT           0B
<missing>      2 months ago   /bin/sh -c #(nop)  EXPOSE 80                    0B
<missing>      2 months ago   /bin/sh -c #(nop)  ENTRYPOINT ["/docker-entr…   0B
<missing>      2 months ago   /bin/sh -c #(nop) COPY file:0fd5fca330dcd6a7…   1.04kB
<missing>      2 months ago   /bin/sh -c #(nop) COPY file:0b866ff3fc1ef5b0…   1.96kB
<missing>      2 months ago   /bin/sh -c #(nop) COPY file:e7e183879c35719c…   1.2kB
<missing>      2 months ago   /bin/sh -c set -x     && addgroup --system -…   63.7MB
<missing>      2 months ago   /bin/sh -c #(nop)  ENV PKG_RELEASE=1~buster      0B
<missing>      2 months ago   /bin/sh -c #(nop)  ENV NJS_VERSION=0.5.0         0B
<missing>      2 months ago   /bin/sh -c #(nop)  ENV NGINX_VERSION=1.19.6      0B
<missing>      2 months ago   /bin/sh -c #(nop)  LABEL maintainer=NGINX Do…   0B
<missing>      2 months ago   /bin/sh -c #(nop)  CMD ["bash"]                  0B
<missing>      2 months ago   /bin/sh -c #(nop) ADD file:422aca8901ae3d869…   69.2MB
```

> nginx这个镜像怎么存的
>
> 使用：docker image inspect nginx

```
        "GraphDriver": {
            "Data": {
                "LowerDir": "/var/lib/docker/overlay2/67b3802c6bdb5bcdbcccbbe7aed20faa7227d584ab37668a03f
f6952e631f7f2/diff:/var/lib/docker/overlay2/f56920fac9c356227079df41c8f4b056118c210bf4c50bd9bb077bdb4c752
4b4/diff:/var/lib/docker/overlay2/0e569a134838b8c2040339c4fdb1f3868a7118dd7f4907b40468f5fe60f055e5/diff:/
var/lib/docker/overlay2/2b51c82933078e19d78b74c248dec38164b90d80c1b42f0fdb1424953207166e/diff",
                "MergedDir": "/var/lib/docker/overlay2/e3b8bdbb0cfbe5450696c470994b3f99e8a7942078e2639a78
8027529c6278f7/merged",
                "UpperDir": "/var/lib/docker/overlay2/e3b8bdbb0cfbe5450696c470994b3f99e8a7942078e2639a788
027529c6278f7/diff",
                "WorkDir": "/var/lib/docker/overlay2/e3b8bdbb0cfbe5450696c470994b3f99e8a7942078e2639a7880
27529c6278f7/work"
            },
            "Name": "overlay2"
        },
```

指示了镜像怎么存的

- `LowerDir`：底层目录；diff（只是存储不同）；包含小型linux和装好的软件

```
/var/lib/docker/overlay2/67b3802c6bdb5bcdbcccbbe7aed20faa7227d584ab37668a03ff6952e631f7f2/diff：用户文件；

/var/lib/docker/overlay2/f56920fac9c356227079df41c8f4b056118c210bf4c50bd9bb077bdb4c7524b4/diff： nginx的启动命令放在这里

/var/lib/docker/overlay2/0e569a134838b8c2040339c4fdb1f3868a7118dd7f4907b40468f5fe60f055e5/diff： nginx的配置文件在这里

/var/lib/docker/overlay2/2b51c82933078e19d78b74c248dec38164b90d80c1b42f0fdb1424953207166e/diff： 小linux系统
```

- 倒着看

- 小linux系统（FROM apline） + Dockerfile的每一个命令可能都引起了系统的修改，所以和git一样，只记录变化
- ```
  25166226 bin    92275050 home    88080717 mnt   104857967 run    16777699 tmp
  29360668 boot   96469387 lib     92275051 opt     4194388 sbin   20971995 usr
  33554839 dev    79692108 lib64   96469388 proc    8843438 srv    29361501 var
  37749165 etc    83886446 media  100663661 root   12588332 sys
  [root@lfy diff]#
  ```
- 我们进入到这个镜像启动的容器，容器的文件系统就是镜像的；
- docker ps -s；可以看到这个容器真正用到的文件大小
- 容器会自己建立层；如果想要改东西，把改的内容复制到容器层即可 docker inspect container
  - ```
    "LowerDir": "/var/lib/docker/overlay2/41e4fa41a2ad1dca9616d4c8254a04c4d9d6a3d462c862f1e9a0562de2384dbc-init/diff:/var/lib/docker/overlay2/e3b8bdbb0cfbe5450696c470994b3f99e8a7942078e2639a788027529c6278f7/diff:/var/lib/docker/overlay2/67b3802c6bdb5bcdbcccbbe7aed20faa7227d584ab37668a03ff6952e631f7f2/diff:/var/lib/docker/overlay2/f56920fac9c356227079df41c8f4b056118c210bf4c50bd9bb077bdb4c7524b4/diff:/var/lib/docker/overlay2/0e569a134838b8c2040339c4fdb1f3868a7118dd7f4907b40468f5fe60f055e5/diff:/var/lib/docker/overlay2/2b51c82933078e19d78b74c248dec38164b90d80c1b42f0fdb1424953207166e/diff",
    "MergedDir": "/var/lib/docker/overlay2/41e4fa41a2ad1dca9616d4c8254a04c4d9d6a3d462c862f1e9a0562de2384dbc/merged",
    "UpperDir": （镜像的上层可以感知变化）"/var/lib/docker/overlay2/41e4fa41a2ad1dca9616d4c8254a04c4d9d6a3d462c862f1e9a0562de2384dbc/diff",【容器的修改后的文件，保存再宿主机哪里呀。容器删除后，那些容器目录还存在吗？一定不再】
    "WorkDir": "/var/lib/docker/overlay2/41e4fa41a2ad1dca9616d4c8254a04c4d9d6a3d462c862f1e9a0562de2384dbc/work"
    ```

- `MergedDir`：合并目录；容器最终的完整工作目录全内容都在合并层；数据卷在容器层产生；所有的增删改都在容器层；
- 

- `UpperDir`：上层目录；
- `WorkDir`：工作目录（临时层），pid；

> LowerDir（底层）\UpperDir（）\MergedDir\WorkDir(临时东西)

> docker底层的 storage driver完成了以上的目录组织结果；

哪些东西适合容器运行？

- docker启动一个MySQL，默认什么都不做？

- MySQL就会丢失数据
  - 文件挂载
  - docker commit：能提交，MySQL的容器，也能提交。100G；100G

1、Images and layers

Docker映像由一系列层组成。 每层代表图像的Dockerfile中的一条指令。 除最后一层外的每一层都是只读的。 如以下Dockerfile：

- Dockerfile文件里面几句话，镜像就有几层

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
# 每一个指令都可能会引起镜像改变，这些改变类似git的方式逐层叠加。
```

> - 该Dockerfile包含四个命令，每个命令创建一个层。
> - FROM语句从ubuntu：15.04映像创建一个图层开始。
> - COPY命令从Docker客户端的当前目录添加一些文件。
> - RUN命令使用make命令构建您的应用程序。
> - 最后，最后一层指定要在容器中运行的命令。
> - 每一层只是与上一层不同的一组。 这些层彼此堆叠。
> - 创建新容器时，可以在基础层之上添加一个新的可写层。 该层通常称为“容器层”。 对运行中的容器所做的所有更改（例如写入新文件，修改现有文件和删除文件）都将写入此薄可写容器层。



2、Container and layers

> - 容器和镜像之间的主要区别是可写顶层。
> - 在容器中添加新数据或修改现有数据的所有写操作都存储在此可写层中。
> - 删除容器后，可写层也会被删除。 基础图像保持不变。 因为每个容器都有其自己的可写容器层，并且所有更改都存储在该容器层中，所以多个容器可以共享对同一基础映像的访问，但具有自己的数据状态。

下图显示了共享同一Ubuntu 15.04映像的多个容器。



3、磁盘容量预估

```
docker ps -s
```

size：用于每个容器的可写层的数据量（在磁盘上）。

virtual size：容器使用的用于只读图像数据的数据量加上容器的可写图层大小。
多个容器可以共享部分或全部只读图像数据。
从同一图像开始的两个容器共享100%的只读数据，而具有不同图像的两个容器（具有相同的层）共享这些公共层。 因此，不能只对虚拟大小进行总计。这高估了总磁盘使用量，可能是一笔不小的数目。

4、镜像如何挑选

busybox：是一个集成了一百多个最常用Linux命令和工具的软件。linux工具里的瑞士军刀
alpine：Alpine操作系统是一个面向安全的轻型Linux发行版经典最小镜像，基于busybox，功能比Busybox完善。
slim：docker hub中有些镜像有slim标识，都是瘦身了的镜像。也要优先选择
无论是制作镜像还是下载镜像，优先选择alpine类型.

5、Copy On Write

- 写时复制是一种共享和复制文件的策略，可最大程度地提高效率。
- 如果文件或目录位于映像的较低层中，而另一层（包括可写层）需要对其进行读取访问，则它仅使用现有文件。
- 另一层第一次需要修改文件时（在构建映像或运行容器时），将文件复制到该层并进行修改。 这样可以将I / O和每个后续层的大小最小化。

2、容器如何挂载



每一个容器里面的内容，支持三种挂载方式：

1）、docker自动在外部创建文件夹自动挂载容器内部指定的文件夹内容【Dockerfile VOLUME指令的作用】

2）、自己在外部创建文件夹，手动挂载

3）、可以把数据挂载到内存中。

--mount 挂载到 linux宿主机，手动挂载（不用了）

-v 可以自动挂载，到linux'主机或者docker自动管理的这一部分区域

- Volumes(卷) ：存储在主机文件系统的一部分中，该文件系统由Docker管理（在Linux上是“ / var / lib / docker / volumes /”）。 非Docker进程不应修改文件系统的这一部分。 卷是在Docker中持久存储数据的最佳方法。
- Bind mounts(绑定挂载) 可以 *在任何地方* 存储在主机系统上。 它们甚至可能是重要的系统文件或目录。 Docker主机或Docker容器上的非Docker进程可以随时对其进行修改。
- tmpfs mounts(临时挂载) 仅存储在主机系统的内存中，并且永远不会写入主机系统的文件系统



1、volume(卷)

- 匿名卷使用

```
docker run -dP -v :/etc/nginx nginx
#docker将创建出匿名卷，并保存容器/etc/nginx下面的内容
# -v 宿主机:容器里的目录
```

- 具名卷使用

```
docker run -dP -v nginx:/etc/nginx nginx
#docker将创建出名为nginx的卷，并保存容器/etc/nginx下面的内容
```

> 如果将空卷装入存在文件或目录的容器中的目录中，则容器中的内容（复制）到该卷中。
>
> 如果启动一个容器并指定一个尚不存在的卷，则会创建一个空卷。

-v 宿主机绝对路径:Docker容器内部绝对路径：叫挂载；这个有空挂载问题

-v 不以/开头的路径:Docker容器内部绝对路径：叫绑定（docker会自动管理，docker不会把他当前目录，而把它当前卷）

以上用哪个比较好？？？？？？

- 如果自己开发测试，用 -v 绝对路径的方式
- 如果是生产环境建议用卷
- 除非特殊 /bin/docker 需要挂载主机路径的则操作 绝对路径挂载

nginx--Docker

/usr/share/nginx/html

nginx测试html挂载几种不同情况：

- 不挂载　效果：访问默认欢迎页
- -v /root/html:/usr/share/nginx/html　效果：访问forbidden
- -v html:/usr/share/nginx/html:ro　效果：访问默认欢迎页
- -v　/usr/share/nginx/html　效果：匿名卷　（什么都不写也不要加冒号，直接写容器内的目录）
- 原因：
  - -v html:/usr/share/nginx/html； docker inspect 容器的时候； docker自动管理的方式

```
# -v不以绝对路径方式：
### 1、先在docker底层创建一个你指定名字的卷（具名卷）　html
### 2、把这个卷和容器内部目录绑定
### 3、容器启动以后，目录里面的内容就在卷里面存着：

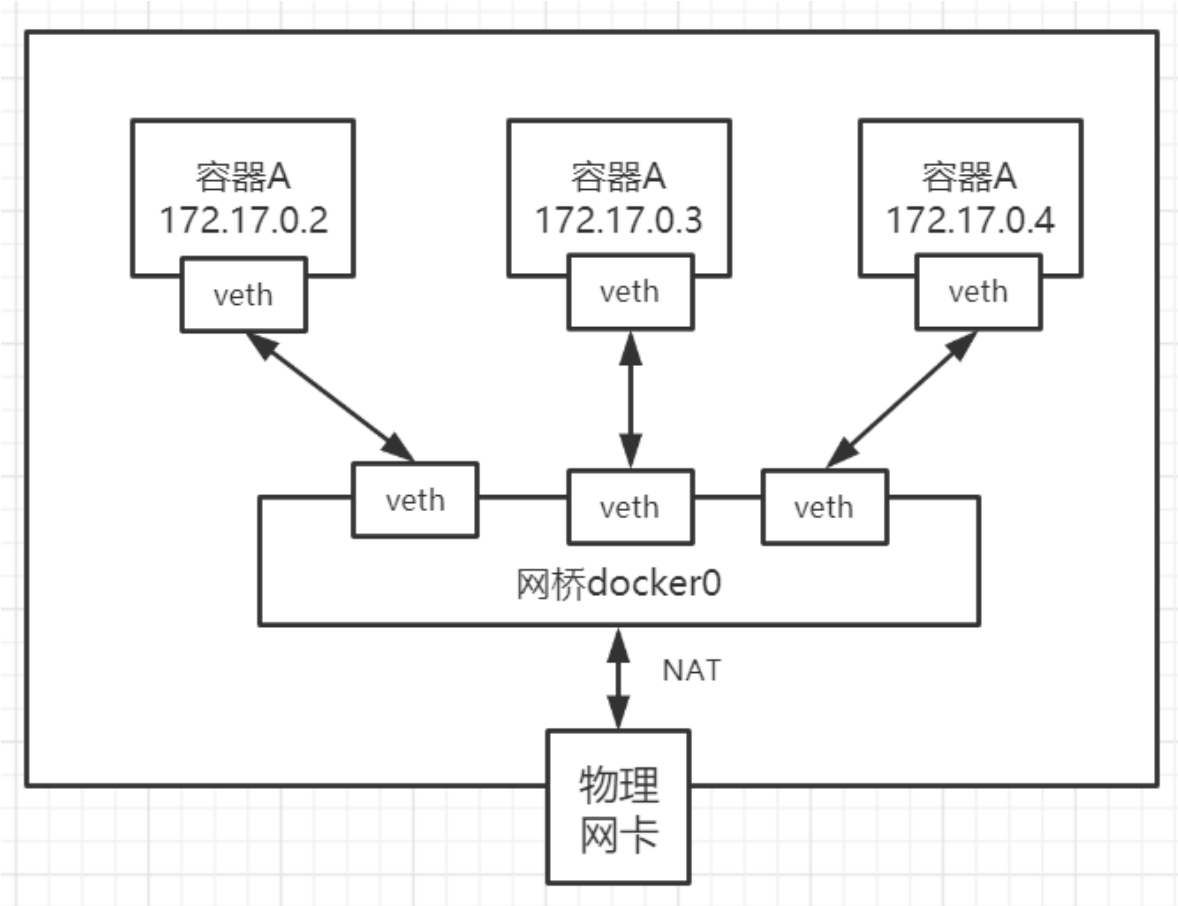
#####-v nginxhtml:/usr/share/nginx/html 也可以以下操作
## 1、 docker create volume nginxhtml　如果给卷里面就行修改，容器内部的也就改了。
## 2、 docker volume inspect nginxhtml
```

```
## 3、docker run -d -P -v nginxhtml:/usr/share/nginx/html  --name=nginx777 nginx
# 可以看到
"Mounts": [
            {
                "Type": "volume", //这是个卷
                "Name": "html",  //名字是html
                "Source": "/var/lib/docker/volumes/html/_data",  //宿主机的目录。容器里面的哪两个文件都在
                "Destination": "/usr/share/nginx/html",  //容器内部
                "Driver": "local",
                "Mode": "z",
                "RW": true,  //读写模式
                "Propagation": ""
            }
        ]
```

```
#卷：就是为了保存数据
docker volume  #可以对docker自己管理的卷目录进行操作；/var/lib/docker/volumes(卷的根目录)

```

2、bind mount

> 如果将绑定安装或非空卷安装到存在某些文件或目录的容器中的目录中，则这些文件或目录会被安装遮盖，就像您将文件保存到Linux主机上的/ mnt中一样，然后 将USB驱动器安装到/ mnt中。在卸载USB驱动器之前，/ mnt的内容将被USB驱动器的内容遮盖。 被遮盖的文件不会被删除或更改，但是在安装绑定安装或卷时将无法访问。
>
> 总结：外部目录覆盖内部容器目录内容，但不是修改。所以谨慎，外部空文件夹挂载方式也会导致容器内部是空文件夹

```
docker run -dP -v /my/nginx:/etc/nginx:ro nginx

# bind mount和 volumes 的方式写法区别在于
# 所有以/开始的都认为是 bind mount ，不以/开始的都认为是 volumes.
```

> 警惕bind mount 方式，文件挂载没有在外部准备好内容而导致的容器启动失败问题

```
# 一行命令启动nginx，并且配置文件和html页面。需要知道卷的位置才能改
docker run -d -P -v nginxconf:/etc/nginx/ -v nginxpage:/usr/share/nginx/html nginx


# 想要实现 docker run -d -P -v /root/nginxconf:/etc/nginx/  -v /root/nginxhtml:/usr/share/nginx/html --name=nginx999 nginx
### 1、提前准备好东西 目录nginxconf，目录里面的配置we年都放里面，，再调用命令
### 2、docker cp nginxdemo:/etc/nginx  /root/nginxconf   #注意/的使用
### 3、docker run -d -P -v /root/nginxconf:/etc/nginx/  -v /root/nginxhtml:/usr/share/nginx/html --name=nginx999 nginx
```

3、管理卷

```
docker volume create xxx：创建卷名
docker volume inspect xxx：查询卷详情
docker volume ls：列出所有卷
docker volume prune：移除无用卷
```

4、docker cp

cp的细节

> docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH|-：把容器里面的复制出来
>
> docker cp [OPTIONS] SRC_PATH|- CONTAINER:DEST_PATH：把外部的复制进去

- `SRC_PATH` 指定为一个文件
  - `DEST_PATH` 不存在：文件名为 `DEST_PATH`，内容为SRC的内容
  - `DEST_PATH` 不存在并且以 `/` 结尾：报错
  - `DEST_PATH` 存在并且是文件：目标文件内容被替换为SRC_PATH的文件内容。
  - `DEST_PATH` 存在并且是目录：文件复制到目录内，文件名为SRC_PATH指定的名字
- `SRC_PATH` 指定为一个目录
  - `DEST_PATH` 不存在：`DEST_PATH` 创建文件夹，复制源文件夹内的所有内容
  - `DEST_PATH` 存在是文件：报错
  - `DEST_PATH` 存在是目录
    - `SRC_PATH` 不以 `/.` 结束：源文件夹复制到目标里面
    - `SRC_PATH` 以 `/.` 结束：源文件夹里面的内容复制到目标里面

    自动创建文件夹不会做递归。把父文件夹做好

    [root@lfy ~]# docker cp index.html mynginx4:/usr/share/nginx/html
    [root@lfy ~]# docker cp mynginx4:/etc/nginx/nginx.conf nginx.conf

2、 Docker网络

1、端口映射

```
docker create -p 3306:3306 -e MYSQL_ROOT_PASSWORD=123456 --name hello-mysql
mysql:5.7

```

2、容器互联

`--link name:alias`，name连接容器的名称，alias连接的别名

场景：我们无需暴露mysql的情况下，让web应用使用mysql；

```
docker run -d -e MYSQL_ROOT_PASSWORD=123456 --name mysql01 mysql:5.7
docker run -d --link mysql01:mysql --name tomcat tomcat:7

docker exec -it tomcat bash
cat /etc/hosts
ping mysql
```

3、自定义网络

1、默认网络原理

Docker使用Linux桥接，在宿主机虚拟一个Docker容器网桥(docker0)，Docker启动一个容器时会根据Docker网桥的网段分配给容器一个IP地址，称为Container-IP，同时Docker网桥是每个容器的默认网关。因为在同一宿主机内的容器都接入同一个网桥，这样容器之间就能够通过容器的Container-IP直接通信。



Linux虚拟网络技术。

Docker容器网络就很好的利用了Linux虚拟网络技术，在本地主机和容器内分别创建一个虚拟接口，并让他们彼此联通（这样一对接口叫veth pair）；

Docker中的网络接口默认都是虚拟的接口。虚拟接口的优势就是转发效率极高（因为Linux是在内核中进行数据的复制来实现虚拟接口之间的数据转发，无需通过外部的网络设备交换），对于本地系统和容器系统来说，虚拟接口跟一个正常的以太网卡相比并没有区别，只是他的速度快很多。

原理：

1、每一个安装了Docker的linux主机都有一个docker0的虚拟网卡。桥接网卡

2、每启动一个容器linux主机多了一个虚拟网卡。

3、docker run -d -P --name tomcat --net bridge tomcat:8

2、网络模式

网络模式	配置	说明
bridge模式	--net=bridge	默认值，在Docker网桥docker0上为容器创建新的网络栈
none模式	--net=none	不配置网络，用户可以稍后进入容器，自行配置
container模式	--net=container:name/id	容器和另外一个容器共享Network namespace。kubernetes中的pod就是多个容器共享一个Network namespace。
host模式	--net=host	容器和宿主机共享Network namespace；
用户自定义	--net=mynet	用户自己使用network相关命令定义网络，创建容器的时候可以指定为自己定义的网络

3、自建网络测试

```
#1、docker0网络的特点。，
    默认、域名访问不通、--link 域名通了，但是删了又不行

#2、可以让容器创建的时候使用自定义网络，用自定义
     1、自定义创建的默认default "bridge"
     2、自定义创建一个网络网络
docker network create --driver bridge --subnet 192.168.0.0/16 --gateway 192.168.0.1 mynet
       所有东西实时维护好，直接域名ping通

docker network connect [OPTIONS] NETWORK CONTAINER

#3、跨网络连接别人就用。把tomcat加入到mynet网络
docker network connect mynet tomcat
效果：
    1、自定义网络，默认都可以用主机名访问通
    2、跨网络连接别人就用 docker network connect mynet tomcat
#4、命令
1、容器启动，指定容器ip。 docker run --ip 192.168.0.3 --net 自定义网络
```

```
2、创建子网。docker network create --subnet 指定子网范围  --driver bridge 所有东西实时维护好，直接域名ping通
3、docker compose 中的网络默认就是自定义网络方式。

docker run -d -P --network 自定义网络名(提前创建)
```

四、深入Dockerfile

Dockerfile由一行行命令语句组成，并且支持以#开头的注释行。

基础的小linux系统。jdk；

一般而言，Dockerfile可以分为四部分

`基础镜像信息` `维护者信息` `镜像操作指令` `启动时执行指令`

指令	说明
FROM	指定基础镜像
MAINTAINER	指定维护者信息，已经过时，可以使用LABEL maintainer=xxx 来替代
RUN	运行命令 v
CMD	指定启动容器时默认的命令 v
ENTRYPOINT	指定镜像的默认入口.运行命令 v
EXPOSE	声明镜像内服务监听的端口 v
ENV	指定环境变量，可以在docker run的时候使用-e改变 v；会被固化到image的config里面
ADD	复制指定的src路径下的内容到容器中的dest路径下，src可以为url会自动下载，可以为tar文件，会自动解压
COPY	复制本地主机的src路径下的内容到镜像中的dest路径下，但不会自动解压等
LABEL	指定生成镜像的元数据标签信息
VOLUME	创建数据卷挂载点
USER	指定运行容器时的用户名或UID
WORKDIR	配置工作目录，为后续的RUN、CMD、ENTRYPOINT指令配置工作目录
ARG	指定镜像内使用的参数（如版本号信息等），可以在build的时候，使用--build-args改变 v
OBBUILD	配置当创建的镜像作为其他镜像的基础镜像是，所指定的创建操作指令
STOPSIGNAL	容器退出的信号值
HEALTHCHECK	健康检查
SHELL	指定使用shell时的默认shell类型

1、FROM

FROM 指定基础镜像，最好挑一些apline，slim之类的基础小镜像.

scratch镜像是一个空镜像，常用于多阶段构建

如何确定我需要什么要的基础镜像?

- Java应用当然是java基础镜像（SpringBoot应用）或者Tomcat基础镜像（War应用）
- JS模块化应用一般用nodejs基础镜像
- 其他各种语言用自己的服务器或者基础环境镜像，如python、golang、java、php等

2、LABEL

标注镜像的一些说明信息。

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

3、RUN

- RUN指令在当前镜像层顶部的新层执行任何命令，并提交结果，生成新的镜像层。
- 生成的提交映像将用于Dockerfile中的下一步。 分层运行RUN指令并生成提交符合Docker的核心概念，就像源代码控制一样。
- exec形式可以避免破坏shell字符串，并使用不包含指定shell可执行文件的基本映像运行RUN命令。可以使用SHELL命令更改shell形式的默认shell。 在shell形式中，您可以使用\（反斜杠）将一条RUN指令继续到下一行。

> - `RUN <command>`（*shell* 形式，`/bin/sh -c` 的方式运行，避免破坏shell字符串）
> - `RUN ["executable", "param1", "param2"]`（*exec* 形式）

```
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'
#上面等于下面这种写法
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'

RUN ["/bin/bash", "-c", "echo hello"]
```

```
# 测试案例
FROM alpine
LABEL maintainer=leifengyang xx=aa

ENV msg='hello atguigu itdachang'

RUN echo $msg
RUN ["echo","$msg"]
RUN /bin/sh -c 'echo $msg'
RUN ["/bin/sh","-c","echo $msg"]

CMD sleep 10000


#总结： 由于[]不是shell形式，所以不能输出变量信息，而是输出$msg。其他任何/bin/sh -c 的形式都可以输出变量信息
```

总结：什么是shell和exec形式

```
1.  shell 是  /bin/sh -c <command>的方式，
2.  exec ["/bin/sh","-c",command] 的方式== shell方式
也就是exec 默认方式不会进行变量替换
```

4、CMD和ENTRYPOINT

0、都可以作为容器启动入口

`CMD` 的三种写法：

- `CMD ["executable","param1","param2"]` (*exec* 方式, 首选方式)
- `CMD ["param1","param2"]` (为ENTRYPOINT提供默认参数)
- `CMD command param1 param2` (*shell* 形式)

`ENTRYPOINT` 的两种写法：

- `ENTRYPOINT ["executable", "param1", "param2"]` (*exec* 方式, 首选方式)
- `ENTRYPOINT command param1 param2` (shell 形式)

```
# 一个示例
FROM alpine
LABEL maintainer=leifengyang

CMD ["1111"]
CMD ["2222"]
ENTRYPOINT ["echo"]

#构建出如上镜像后测试
docker run xxxx：效果 echo 1111
```

1、只能有一个CMD

- Dockerfile中只能有一条CMD指令。 如果您列出多个CMD，则只有最后一个CMD才会生效。
- CMD的主要目的是为执行中的容器提供默认值。 这些默认值可以包含可执行文件，也可以省略可执行文件，在这种情况下，您还必须指定ENTRYPOINT指令。

2、CMD为ENTRYPOINT提供默认参数

- 如果使用CMD为ENTRYPOINT指令提供默认参数，则CMD和ENTRYPOINT指令均应使用JSON数组格式指定。

3、组合最终效果

	无ENTRYPOINT	**ENTRYPOINT exec_entry p1_entry**	**ENTRYPOINT [“exec_entry”, “p1_entry”]**	
无CMD	*错误, 不允许的写法*； 容器没有启动命令	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry	
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd	
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	`exec_entry p1_entry p1_cmd p2_cmd`	

CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd 无ENTRYPOINT p1_cmd	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT /bin/sh -c “exec_entry p1_entry”]	
		这条竖线，总是以ENTRYPOINT的为准	这条竖线，ENTRYPOINT和CMD共同作用	

4、docker run启动参数会覆盖CMD内容

```
# 一个示例
FROM alpine
LABEL maintainer=leifengyang

CMD ["1111"]
ENTRYPOINT ["echo"]

#构建出如上镜像后测试
docker run xxxx：什么都不传则  echo 1111
docker run xxx arg1：传入arg1 则echo arg1
```

5、ARG和ENV

1、ARG

- ARG指令定义了一个变量，用户可以在构建时使用--build-arg = 传递，docker build命令会将其传递给构建器。
- --build-arg 指定参数会覆盖Dockerfile 中指定的同名参数
- 如果用户指定了 `未在Dockerfile中定义的构建参数` ，则构建会输出 `警告` 。
- ARG只在构建期有效，运行期无效
- 不建议使用构建时变量来传递诸如github密钥，用户凭据等机密。因为构建时变量值使用docker history是可见的。
- ARG变量定义从Dockerfile中定义的行开始生效。
- 使用ENV指令定义的环境变量始终会覆盖同名的ARG指令。

2、ENV

- 在构建阶段中所有后续指令的环境中使用，并且在许多情况下也可以内联替换。
- 引号和反斜杠可用于在值中包含空格。
- ENV 可以使用key value的写法，但是这种不建议使用了，后续版本可能会删除

```
ENV MY_MSG hello
ENV MY_NAME="John Doe"
ENV MY_DOG=Rex\ The\ Dog
ENV MY_CAT=fluffy
#多行写法如下
ENV MY_NAME="John Doe" MY_DOG=Rex\ The\ Dog \
    MY_CAT=fluffy
```

- docker run --env 可以修改这些值
- 容器运行时ENV值可以生效
- ENV在image阶段就会被解析并持久化（docker inspect image查看），参照下面示例。

```
FROM alpine
ENV arg=1111111
ENV runcmd=$arg
RUN echo $runcmd
CMD echo $runcmd
#ENV的固化问题： 改变arg，会不会改变 echo的值，会改变哪些值，如何修改这些值?
```

3、综合测试示例

```
FROM alpine
ARG arg1=22222
ENV arg2=1111111
ENV runcmd=$arg1
RUN echo $arg1 $arg2 $runcmd
CMD echo $arg1 $arg2 $runcmd
```

6、ADD和COPY

1、COPY

COPY的两种写法

```
COPY [--chown=<user>:<group>] <src>... <dest>
COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

- --chown功能仅在用于构建Linux容器的Dockerfiles上受支持，而在Windows容器上不起作用
- COPY指令从 `src` 复制新文件或目录，并将它们添加到容器的文件系统中，路径为 `dest` 。
- 可以指定多个 `src` 资源，但是文件和目录的路径将被解释为相对于构建上下文的源。
- 每个 `src` 都可以包含通配符，并且匹配将使用Go的 **filepath.Match** 规则进行。

```
COPY hom* /mydir/ #当前上下文，以home开始的所有资源
COPY hom?.txt /mydir/ # ?匹配单个字符
COPY test.txt relativeDir/ # 目标路径如果设置为相对路径，则相对与 WORKDIR 开始
# 把 “test.txt” 添加到 <WORKDIR>/relativeDir/

COPY test.txt /absoluteDir/ #也可以使用绝对路径，复制到容器指定位置


#所有复制的新文件都是uid(0)/gid(0)的用户，可以使用--chown改变
COPY --chown=55:mygroup files* /somedir/
COPY --chown=bin files* /somedir/
COPY --chown=1 files* /somedir/
COPY --chown=10:11 files* /somedir/
```

2、ADD

同COPY用法，不过 ADD拥有自动下载远程文件和解压的功能。

注意：

- `src` 路径必须在构建的上下文中； 不能使用 `../something /something` 这种方式，因为docker构建的第一步是将上下文目录（和子目录）发送到docker守护程序。
- 如果 `src` 是URL，并且 `dest` 不以斜杠结尾，则从URL下载文件并将其复制到 `dest` 。
  - 如果 `dest` 以斜杠结尾，将自动推断出url的名字（保留最后一部分），保存到 `dest`
- 如果 `src` 是目录，则将复制目录的整个内容，包括文件系统元数据。

7、WORKDIR和VOLUME

1、WORKDIR

- WORKDIR指令为Dockerfile中跟随它的所有 `RUN，CMD，ENTRYPOINT，COPY，ADD` 指令设置工作目录。 如果WORKDIR不存在，即使以后的Dockerfile指令中未使用它也将被创建。
- WORKDIR指令可在Dockerfile中多次使用。 如果提供了相对路径，则它将相对于上一个WORKDIR指令的路径。 例如：

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

- 也可以用到环境变量

```
ENV DIRPATH=/path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
#结果 /path/$DIRNAME
```

2、VOLUME

作用：把容器的某些文件夹映射到主机外部

写法：

```
VOLUME ["/var/log/"]  #可以是JSON数组
VOLUME /var/log  #可以直接写
VOLUME /var/log /var/db #可以空格分割多个
```

注意：

用 VOLUME 声明了卷，那么以后对于卷内容的修改会被丢弃，所以， `一定在volume声明之前修改内容` ；

8、USER

写法：

```
USER <user>[:<group>]
USER <UID>[:<GID>]
```

- USER指令设置运行映像时要使用的用户名（或UID）以及可选的用户组（或GID），以及Dockerfile中USER后面所有RUN，CMD和ENTRYPOINT指令。

9、EXPOSE

- EXPOSE指令通知Docker容器在运行时在指定的网络端口上进行侦听。 可以指定端口是侦听TCP还是UDP，如果未指定协议，则默认值为TCP。
- EXPOSE指令实际上不会发布端口。 它充当构建映像的人员和运行容器的人员之间的一种文档，即有关打算发布哪些端口的信息。 要在运行容器时实际发布端口，请在docker run上使用-p标志发布并映射一个或多个端口，或使用-P标志发布所有公开的端口并将其映射到高阶端口。

```
EXPOSE <port> [<port>/<protocol>...]
EXPOSE [80,443]
EXPOSE 80/tcp
EXPOSE 80/udp

```

11、multi-stage builds

多阶段构建

1、使用

https://docs.docker.com/develop/develop-images/multistage-build/

解决：如何让一个镜像变得更小; 多阶段构建的典型示例

```
### 我们如何打包一个Java镜像
FROM maven
WORKDIR /app
COPY . .
RUN mvn clean package
COPY /app/target/*.jar /app/app.jar
ENTRYPOINT  java -jar app.jar

## 这样的镜像有多大?

## 我们最小做到多大? ?
```

2、生产示例

```
#以下所有前提 保证Dockerfile和项目在同一个文件夹
# 第一阶段：环境构建； 用这个也可以
FROM maven:3.5.0-jdk-8-alpine AS builder
WORKDIR /app
ADD ./ /app
RUN mvn clean package -Dmaven.test.skip=true

# 第二阶段，最小运行时环境，只需要jre；第二阶段并不会有第一阶段哪些没用的层
#基础镜像没有 jmap； jdk  springboot-actutor（jdk）
FROM openjdk:8-jre-alpine

LABEL maintainer="534096094@qq.com"
# 从上一个阶段复制内容
COPY --from=builder /app/target/*.jar /app.jar

# 修改时区
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && echo
'Asia/Shanghai' >/etc/timezone && touch /app.jar


ENV JAVA_OPTS=""
ENV PARAMS=""
# 运行jar包
ENTRYPOINT [ "sh", "-c", "java -Djava.security.egd=file:/dev/./urandom
$JAVA_OPTS -jar /app.jar $PARAMS" ]
```

```
<!--为了加速下载需要在pom文件中复制如下 -->
    <repositories>
        <repository>
            <id>aliyun</id>
            <name>Nexus Snapshot Repository</name>
            <url>https://maven.aliyun.com/repository/public</url>
            <layout>default</layout>
            <releases>
                <enabled>true</enabled>
            </releases>
            <!--snapshots默认是关闭的,需要开启  -->
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>aliyun</id>
            <name>Nexus Snapshot Repository</name>
            <url>https://maven.aliyun.com/repository/public</url>
            <layout>default</layout>
            <releases>
                <enabled>true</enabled>
            </releases>
```

```
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>
```

```
######小细节

RUN /bin/cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && echo
'Asia/Shanghai' >/etc/timezone
或者
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && echo
'Asia/Shanghai' >/etc/timezone
可以让镜像时间同步。


## 容器同步系统时间 CST（China Shanghai Timezone）
-v /etc/localtime:/etc/localtime:ro
#已经不同步的如何同步？
docker cp /etc/localtime 容器id:/etc/
```

docker build --build-arg url="git address" -t demo:test . ： 自动拉代码并构建镜像

```


FROM  maven:3.6.1-jdk-8-alpine AS buildapp
#第二阶段，把克隆到的项目源码拿过来
# COPY --from=gitclone *  /app/
WORKDIR /app
COPY pom.xml .
COPY src .

RUN mvn clean package -Dmaven.test.skip=true

# /app 下面有 target
RUN pwd && ls -l

RUN cp /app/target/*.jar  /app.jar
RUN ls -l
### 以上第一阶段结束，我们得到了一个 app.jar


## 只要一个JRE
# FROM openjdk:8-jre-alpine
FROM openjdk:8u282-slim
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && echo
'Asia/Shanghai' >/etc/timezone
LABEL maintainer="534096094@qq.com"
# 把上一个阶段的东西复制过来
COPY --from=buildapp /app.jar  /app.jar

# docker run -e JAVA_OPTS="-Xmx512m -Xms33 -" -e PARAMS="--spring.profiles=dev -
-server.port=8080" -jar /app/app.jar
# 启动java的命令
```

```
ENV JAVA_OPTS=""
ENV PARAMS=""
ENTRYPOINT [ "sh", "-c", "java -Djava.security.egd=file:/dev/./urandom
$JAVA_OPTS -jar /app.jar $PARAMS" ]
```

自己写一个多阶段构建

1、自动从git下载指定的项目

2、把项目自动打包生成镜像

3、我们只需要运行镜像即可

12、Images瘦身实践

- 选择最小的基础镜像
- 合并RUN环节的所有指令，少生成一些层
- RUN期间可能安装其他程序会生成临时缓存，要自行删除。如：
-

```
# 开发期间，逐层验证正确的
RUN xxx
RUN xxx
RUN aaa \
aaa \
vvv \




#生产环境
RUN apt-get update && apt-get install -y \
  bzr \
  cvs \
  git \
  mercurial \
  subversion \
  && rm -rf /var/lib/apt/lists/*
```

- 使用 `.dockerignore` **文件**，排除上下文中无需参与构建的资源
- 使用多阶段构建
- 合理使用构建缓存加速构建。[--no-cache]

学习更多Dockerfile的写法： https://github.com/docker-library/

13、springboot java 最终写法

```
FROM openjdk:8-jre-alpine
LABEL maintainer="534096094@qq.com"

COPY target/*.jar /app.jar
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && echo
'Asia/Shanghai' >/etc/timezone && touch /app.jar

ENV JAVA_OPTS=""
ENV PARAMS=""

ENTRYPOINT [ "sh", "-c", "java -Djava.security.egd=file:/dev/./urandom
$JAVA_OPTS -jar /app.jar $PARAMS" ]

# 运行命令 docker run -e JAVA_OPTS="-Xmx512m -Xms33 -" -e PARAMS="--
spring.profiles=dev --server.port=8080" -jar /app/app.jar
```

五、熟悉docker-compose

https://docs.docker.com/compose/install/

场景： 一个复杂的应用，不是启动一个容器就能完成的

app = nginx + web + mysql +redis

以后只要启动app，创建app。都得run 4个，保证网络畅通+挂载ok

docker-compose

不懂yaml？？？？？？？？？？？

application.yaml

可以写一个yaml文件。指定所有需要启动的内容。docker-compose up/down

1、基础

安装

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.0/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

2、yaml语法

```
{
    "name": "itdachang",
    "url": "http://www.itdachang.com",
    "page": 88,
    "address": {
        "street": "宏福科技园",
        "city": "北京市昌平区",
        "country": "中国"
    },
    "links": [
        {
            "name": "Google",
            "url": "http://www.google.com"
        },
        {
            "name": "Baidu",
            "url": "http://www.baidu.com"
        }
    ]
}
```

以上JSON转为Yaml为

```
# 这是yaml注释
# key: value  基本键值对写法；  yaml使用缩进控制层次。
```

3、compose语法

app == wordpress （个人博客） web ---- mysql（存储层）

4、compose示例

hello-world

- ```
  mkdir composetest
  ```
- 创建app.py
- 其他.,..,

- 编写compose文件[compose.yaml]
- compose文件名 docker-compose.yml, docker-compose.yaml, compose.yml, compose.yaml

```
version: "3.9" #指定版本号;查看文档https://docs.docker.com/compose/compose-file/
services:  #所有需要启动的服务
  web:   #第一个服务的名字
    build:   #docker build -t xxx -f Dockerfile .
      dockerfile: Dockerfile
      context: .
    image: 'hello:py'
    ports: #指定启动容器暴露的端口
      - "5000:5000"
  redis: #第二个服务的名字
    image: "redis:alpine"
#  mysqlserver: #第三个服务


#怎么执行的
Creating network "composetest_default" with the default driver
Building web
Sending build context to Docker daemon  5.632kB

Step 1/10 : FROM python:3.7-alpine
。。。。。。
Step 10/10 : CMD ["flask", "run"]
 ---> Running in 01e36491132c
Removing intermediate container 01e36491132c
 ---> 47d09826ac6f
Successfully built 47d09826ac6f
Successfully tagged hello:py
======web镜像名 hello:py===


WARNING: Image for service web was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.
Pulling redis (redis:alpine)...
.......
Status: Downloaded newer image for redis:alpine
##下载成功


Creating composetest_redis_1 ... done
Creating composetest_web_1   ... done
Attaching to composetest_web_1, composetest_redis_1
redis_1  | 1:C 15 Apr 2021 13:55:27.693 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1  | 1:C 15 Apr 2021 13:55:27.693 # Redis version=6.2.1, bits=64, commit=00000000, modified=0, pid=1, just started
```
```

```
redis_1  | 1:C 15 Apr 2021 13:55:27.693 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
redis_1  | 1:M 15 Apr 2021 13:55:27.694 * monotonic clock: POSIX clock_gettime
redis_1  | 1:M 15 Apr 2021 13:55:27.695 * Running mode=standalone, port=6379.
redis_1  | 1:M 15 Apr 2021 13:55:27.695 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1  | 1:M 15 Apr 2021 13:55:27.695 # Server initialized
redis_1  | 1:M 15 Apr 2021 13:55:27.695 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
redis_1  | 1:M 15 Apr 2021 13:55:27.695 * Ready to accept connections
web_1    |  * Serving Flask app "app.py"
web_1    |  * Environment: production
web_1    |    WARNING: This is a development server. Do not use it in a production deployment.
web_1    |    Use a production WSGI server instead.
web_1    |  * Debug mode: off
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

#因为compose创建了自定义网络，大家都能通
```

一句话启动这个 docker-compose up -d

```
version: "3.9" #指定版本号;查看文档https://docs.docker.com/compose/compose-file/
services:  #所有需要启动的服务
  web:   #第一个服务的名字
    build:   #docker build -t xxx -f Dockerfile .
      dockerfile: Dockerfile
      context: .
    image: 'hello:py'
    ports: #指定启动容器暴露的端口
      - "5000:5000"
  redis: #第二个服务的名字
    image: "redis:alpine"
  mysql:
    image: "mysql"
 其他volumes,networks等

compose+docker swarm == 集群部署
```

```
version: "3.7"

services:
  app:
    image: node:12-alpine
    command: sh -c "yarn install && yarn run dev"
    ports:
      - 3000:3000
```

```
      working_dir: /app
      volumes:
        - ./:/app
      environment:
        MYSQL_HOST: mysql
        MYSQL_USER: root
        MYSQL_PASSWORD: secret
        MYSQL_DB: todos
      networks:
        - hello
        - world
      deploy: #安装docker swarm
        replicas: 6 #指定副本：处于不同的服务器（负载均衡+高可用）


    mysql:  #可以代表一个容器，ping 服务名 mysql 可以访问
      image: mysql:5.7  #负载均衡下，数据一致怎么做？？？主从同步，读写分离
      volumes:
        - todo-mysql-data:/var/lib/mysql
      environment:
        MYSQL_ROOT_PASSWORD: secret
        MYSQL_DATABASE: todos
      networks: #这个服务加入那个自定义网络
        - hello
      deploy: #安装docker swarm
        replicas: 6 #指定副本：处于不同的服务器（负载均衡+高可用）


    redis:
      image: redis
      networks:
        - world


  volumes:
    todo-mysql-data:
  networks:
     hello:
     world:
```

六、docker swarm

两句：

- docker swarm init （创建一个master 节点）
  - 控制台打印
  - docker swarm join --token SWMTKN-1-1i0biktih9tfn7mrj6asn27em4vydg8pp00u930nrycpgct1ww-7ecs32nl5f5y8qx6e5lp4f064 10.120.82.4:2377
- 其他和本机（master）能互通的机器 把上面的命令运行，加入集群

- docker swarm join --token SWMTKN-1-1i0biktih9tfn7mrj6asn27em4vydg8pp00u930nrycpgct1ww-7ecs32nl5f5y8qx6e5lp4f064 10.120.82.4:2377

k8s怎么解决： helm把整个应用的部署打成应用包，helm install mysql（主从）

在青云开的三个机器默认不通，想办法让他们处于同一子网。（创建vpc）

参照 **https://workshop.pek3a.qingstor.com/CloudOperation_100P001C201908_%E8%99%9A%E6%8B%9F%E4%B8%BB%E6%9C%BA%E7%9A%84%E5%88%9B%E5%BB%BA%E5%92%8C%E4%BD%BF%E7%94%A8_v2.mp4**

也就是 **https://www.qingcloud.com/products/instances/** 下面的视频