目录

[Docker学习 1](#_Toc3395)

[一、 docker概述 1](#_Toc23916)

[1． Docker 为什么出现？ 1](#_Toc2085)

[2． docker的历史 2](#_Toc23693)

[3． docker能干嘛 3](#_Toc12039)

[二、 docker安装 5](#_Toc8243)

[1. docker的基本组成 5](#_Toc19412)

[2. 安装docker 6](#_Toc24758)

[3. 阿里云镜像加速 10](#_Toc23344)

[4. 回归HelloWorld流程 11](#_Toc32531)

[5. 底层原理 12](#_Toc23913)

[三、 docker的常用命令 13](#_Toc6296)

[1. 帮助命令 13](#_Toc2020)

[2. 镜像命令 13](#_Toc11555)

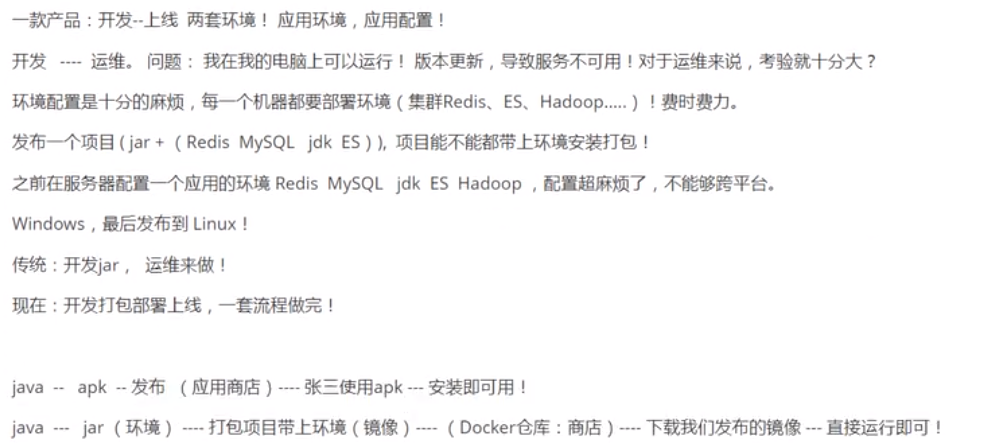
[3. 容器命令 16](#_Toc8942)

[4. 常用其他命令 17](#_Toc5160)

# Docker学习

## Docker概述

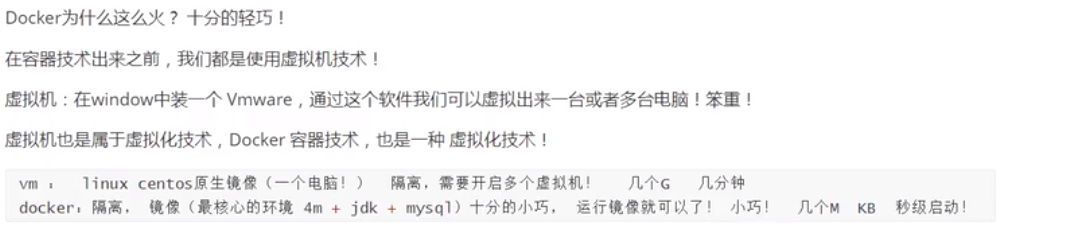
### Docker 为什么出现？





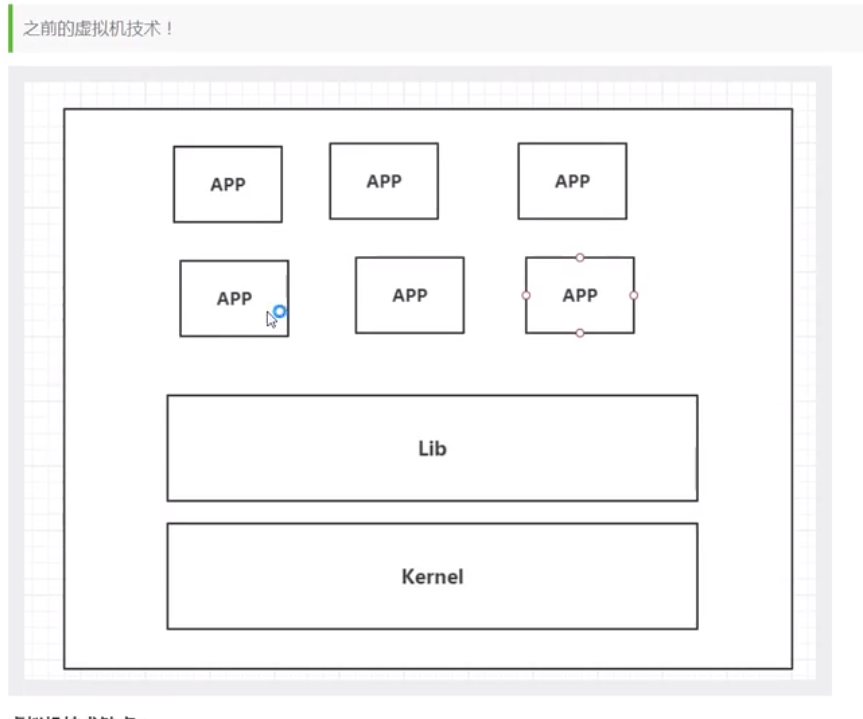
### docker的历史

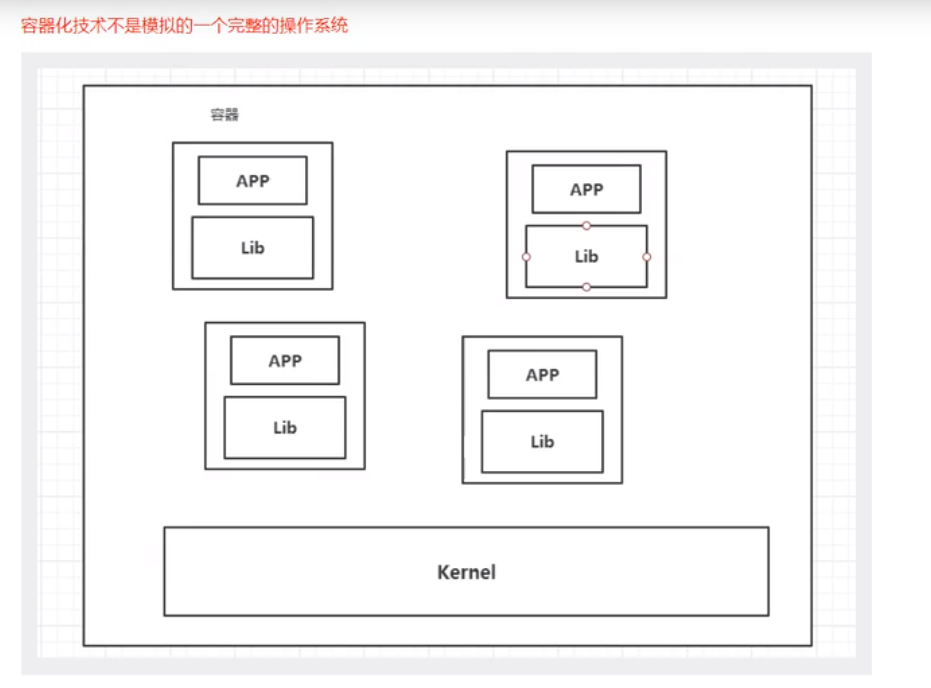


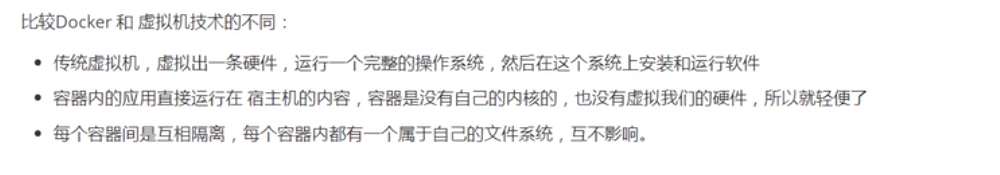




### docker能干嘛



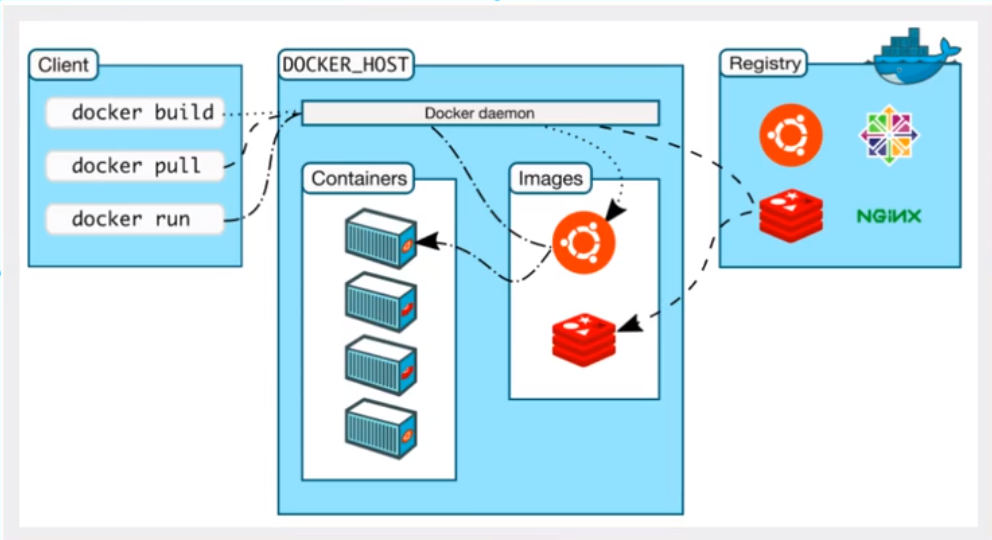


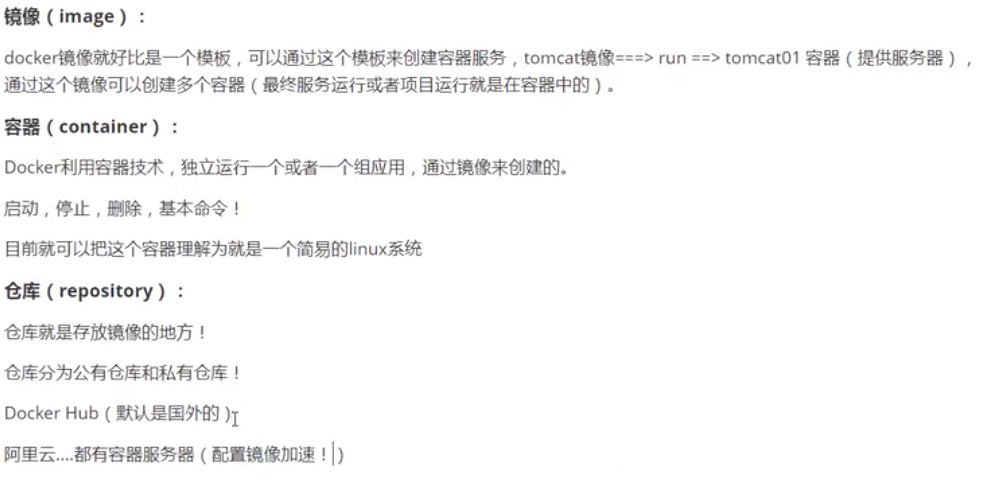




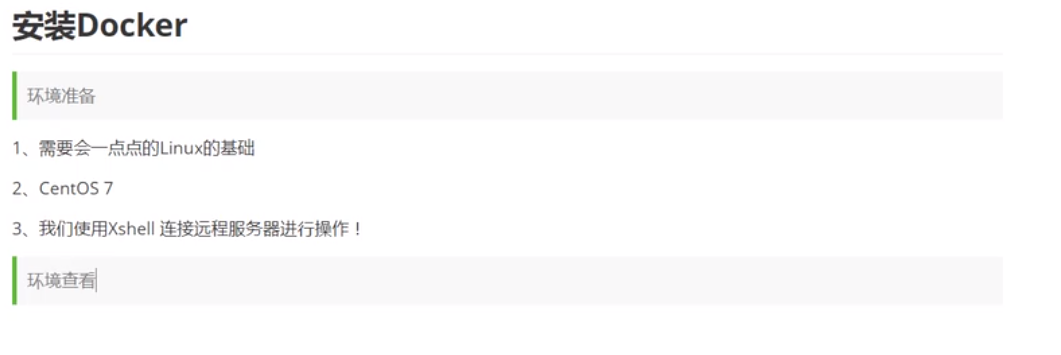
## Docker安装

### docker的基本组成



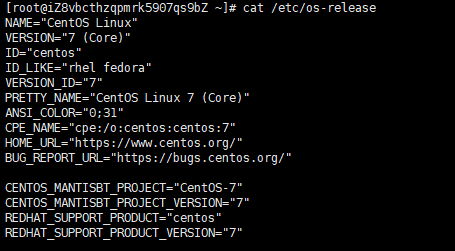


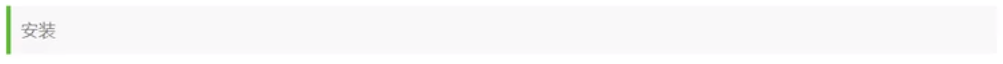
### 安装docker





系统内核是3.10





帮助文档：

1. 卸载旧版本

$ yum remove docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-engine

1. 需要的安装包

$ yum install -y yum-utils

1. 设置镜像的仓库

yum-config-manager \

--add-repo \

https://download.docker.com/linux/centos/docker-ce.repo

这里默认是国外的

yum-config-manager \

--add-repo \

http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

1. 更新yum软件包索引

yum makecache fast

1. 安装docker相关的内容

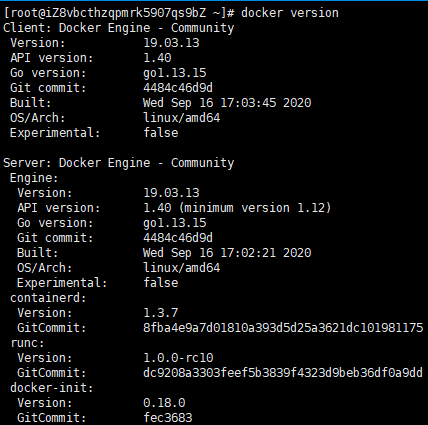
yum install docker-ce docker-ce-cli containerd.io

docker-ce 社区版 ee 是企业版

1. 启动docker

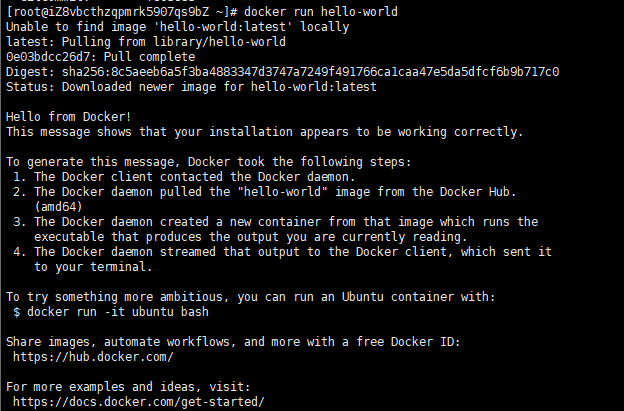
systemctl start docker

docker version 查看是否安装成功



1. 测试hello -world image

docker run hello-world



1. 查看一下下载的这个hello-world镜像

docker images

了解：卸载docker

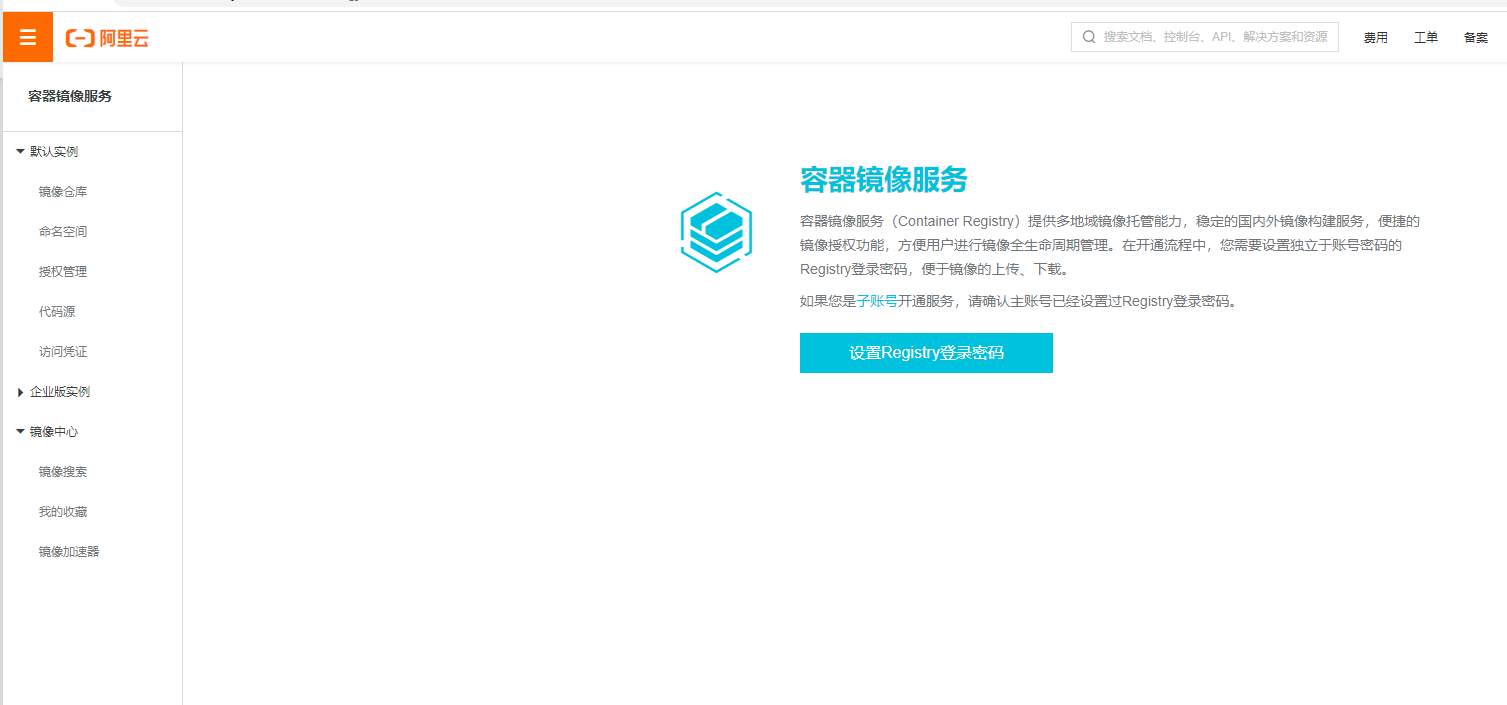
1. 卸载依赖

yum remove docker-ce docker-ce-cli containerd.io

1. 删除资源

rm -rf /var/lib/docker docker的默认工作路径

### 阿里云镜像加速





sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'

{

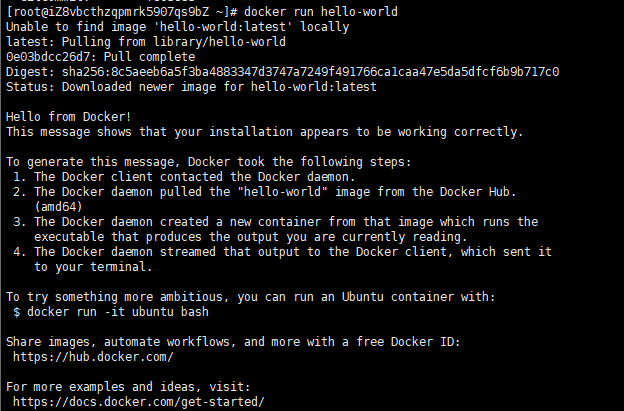
"registry-mirrors": ["https://1yj6g1jy.mirror.aliyuncs.com"]

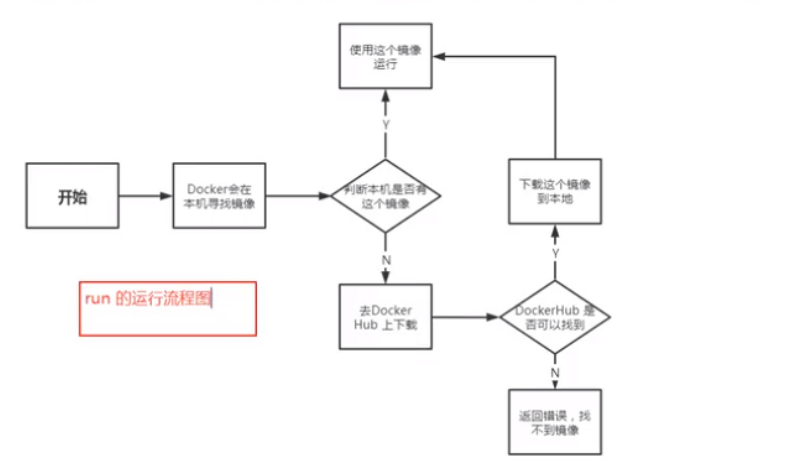
} EOF

sudo systemctl daemon-reload

sudo systemctl restart docker

### 回归HelloWorld流程

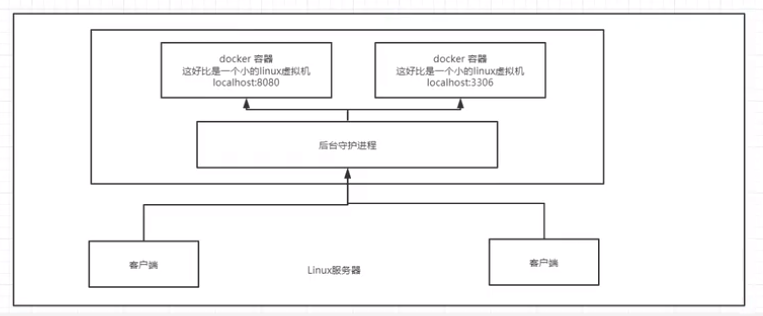




### 底层原理

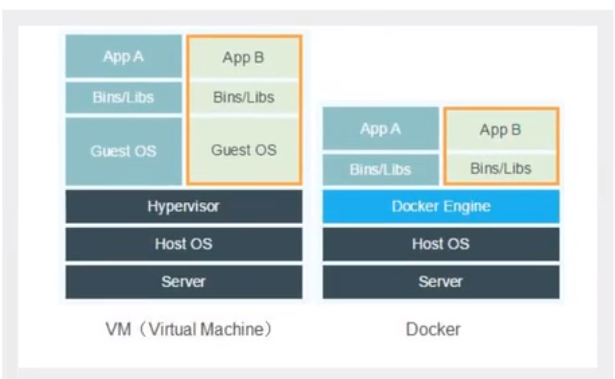
docker是怎么工作的？

* docker是一个C/S结构的系统，Docker的守护进程运行在主机上，通过Socket从客户端访问。dockerServer接收到Docker-Client的指令，就会执行这个命令。



Docker为什么比VM快？

* docker有着比虚拟机更少的抽象层
* docker利用的是宿主机的内核，VM需要是GuestOS



新建一个容器的时候，docker不需要像虚拟机一样重新加载一个操作系统内核，避免引导。虚拟机是加载GuestOS，分钟级别的，而docker是利用宿主机的操作系统，省略了这个复杂的过程，秒级。

## Docker的常用命令

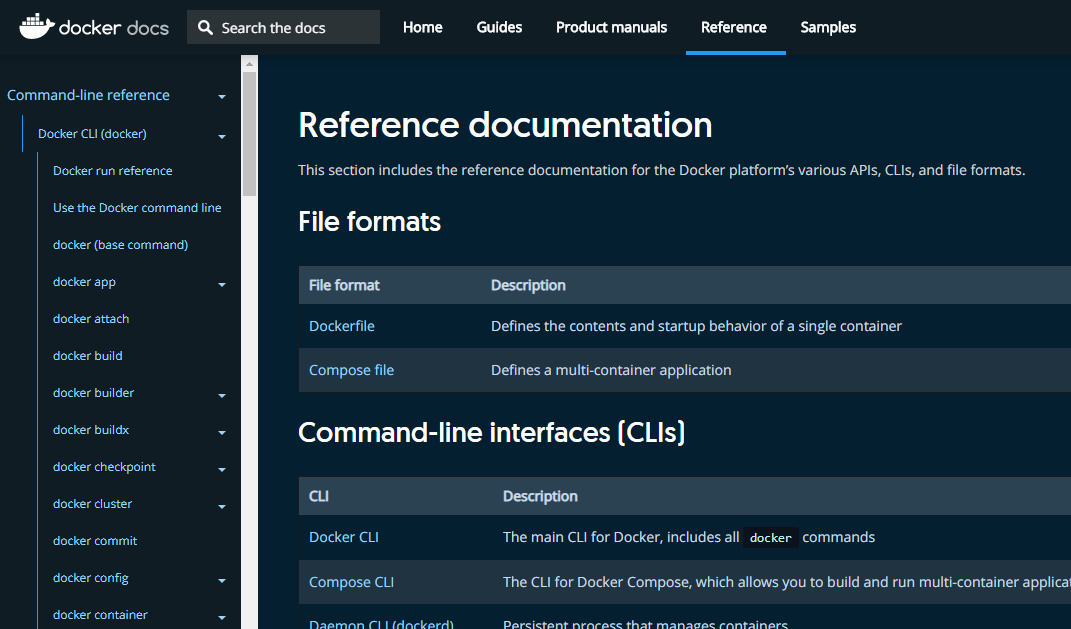
### 帮助命令

docker version # 显示docker信息

docker info # 显示docker系统信息，包括镜像和容器数量

docker --help # 帮助命令

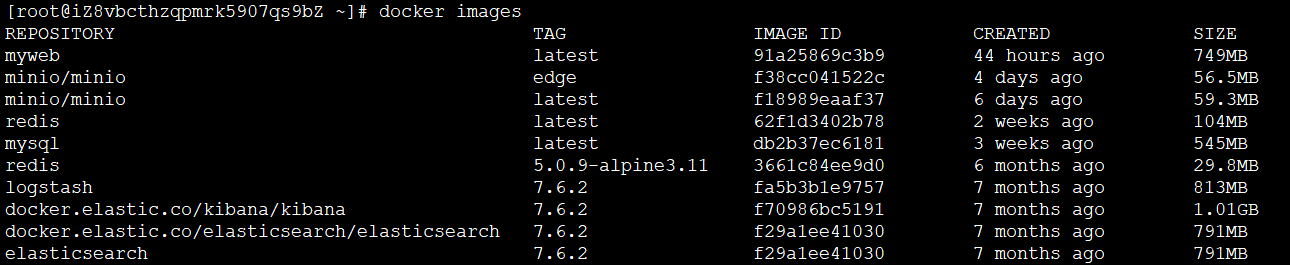
官方文档命令位置



### 镜像命令

#### docker images

docker images # 查看所有本地的主机上的镜像



# 解释

REPOSITORY 镜像的仓库源

TAG 镜像的标签

IMAGE ID 镜像的id

CREATED 镜像的创建时间

SIZE 镜像的大小

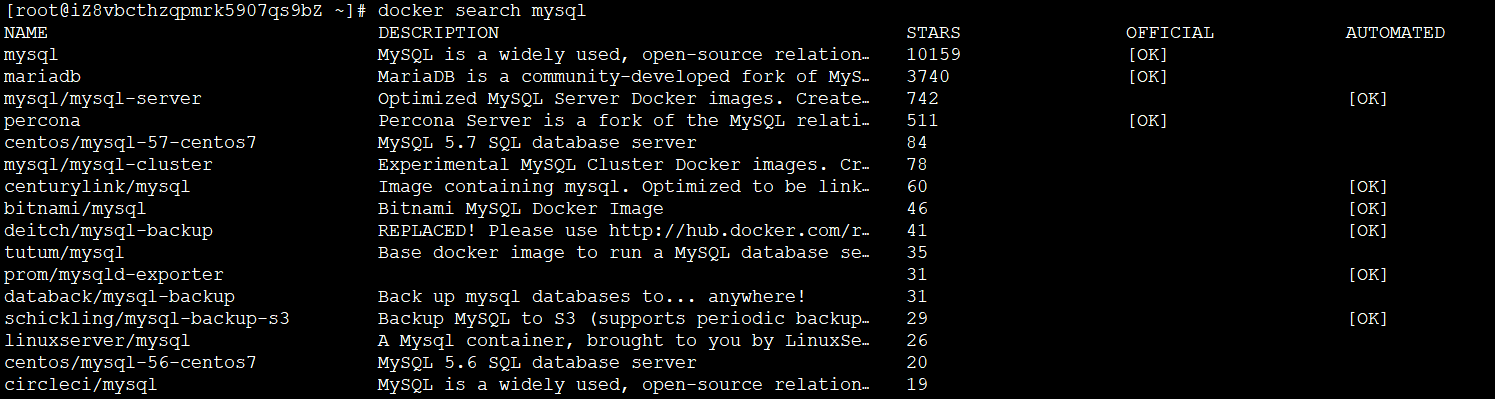
# 可选项

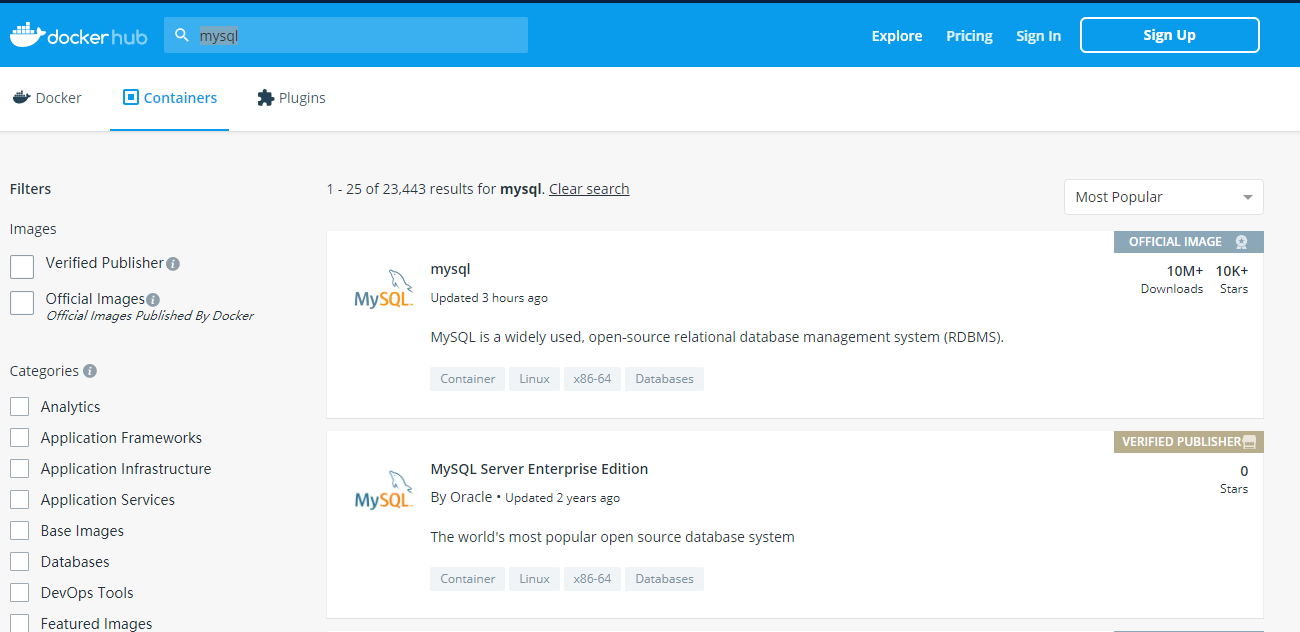
-a, --all #列出所有镜像

-q, --quiet #只显示镜像的id

#### 2.2.docker search

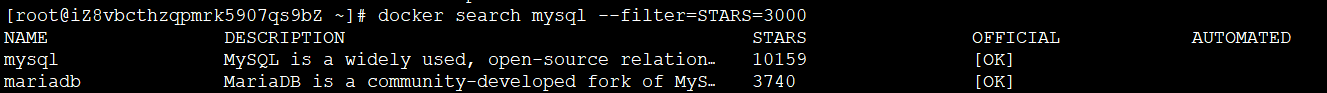
docker search # 搜索镜像

和网站https://hub.docker.com里搜索mysql出来的列表是对应的



# 可选项，通过收藏来过滤

--filter=STARS=3000 搜索出来的镜像就是STAR大于3000的



#### 2.3.docker pull

docker pull #下载镜像 docker pull 镜像名[:tag]

[root@iZ8vbcthzqpmrk5907qs9bZ ~]# docker pull mysql

Using default tag: latest #如果不写tag，就默认最新

latest: Pulling from library/mysql

bb79b6b2107f: Pull complete # 分层下载，docker image核心 联合文件系统

49e22f6fb9f7: Pull complete

842b1255668c: Pull complete

9f48d1f43000: Pull complete

c693f0615bce: Pull complete

8a621b9dbed2: Pull complete

0807d32aef13: Pull complete

a56aca0feb17: Pull complete

de9d45fd0f07: Pull complete

1d68a49161cc: Pull complete

d16d318b774e: Pull complete

49e112c55976: Pull complete

#签名信息

Digest: sha256:8c17271df53ee3b843d6e16d46cff13f22c9c04d6982eb15a9a47bd5c9ac7e2d

Status: Downloaded newer image for mysql:latest

#真实地址 等价于 docker pull docker.io/library/mysql:latest

docker.io/library/mysql:latest

# 指定版本下载 docker pull mysql:5.7

[root@iZ8vbcthzqpmrk5907qs9bZ ~]# docker pull mysql:5.7

5.7: Pulling from library/mysql

bb79b6b2107f: Already exists

49e22f6fb9f7: Already exists

842b1255668c: Already exists

9f48d1f43000: Already exists

c693f0615bce: Already exists

8a621b9dbed2: Already exists

0807d32aef13: Already exists

f15d42f48bd9: Pull complete

098ceecc0c8d: Pull complete

b6fead9737bc: Pull complete

351d223d3d76: Pull complete

Digest: sha256:4d2b34e99c14edb99cdd95ddad4d9aa7ea3f2c4405ff0c3509a29dc40bcb10ef

Status: Downloaded newer image for mysql:5.7

docker.io/library/mysql:5.7

#### 2.4.docker rmi

docker rmi -f 镜像id # 删除指定的镜像

docker rmi -f 镜像id 镜像id 镜像id #删除多个指定的镜像

docker rmi -f $(docker images -aq) #删除全部镜像

### 容器命令

说明：有了镜像才能创建容器，linux,下载一个centos镜像来测试学习

1. 下载一个centos镜像

docker pull centos

1. 启动容器

docker run [可选参数] image

# 参数说明

--name=”Name” #容器名字 tomcat01这种来区分容器

-d #后台方式运行

-it #使用交互方式运行，进入容器查看内容

-p #指定容器端口 -p 8080:8080

-P #随机指定端口

#运行容器并进入 docker run -it [REPOSITORY] /bin/bash

[root@iZ8vbcthzqpmrk5907qs9bZ ~]# docker run -it centos /bin/bash

[root@f88ca6ed1607 /]# ls

bin dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var

[root@f88ca6ed1607 /]#

1. 退出容器

exit #直接容器停止并退出

ctrl + p + q #容器不停止退出

1. 列出所有运行的容器

docker ps # 查看运行中的容器

docker ps -a # 查看曾经运行过的容器和现在在运行的容器

docker ps -n=? # 显示最近创建的容器,n=数量

docker ps -q # 只显示容器的编号

1. 删除容器

docker rm 容器id #删除指定的容器，不能删除正在运行的容 器，如果要强制，rm -f

docker rm -f $(docker ps -aq) #删除所有的容器

docker ps -a -q|xargs docker rm #删除所有的容器

1. 启动和停止容器的操作

docker start 容器id # 启动容器

docker restart 容器id # 重启容器

docker stop 容器id # 停止容器

docker kill 容器id # 杀死容器

### 常用其他命令

1. 后台启动容器

docker run -d 镜像名

# 用docker ps ,发现centos 停止了

# 常见的坑：docker容器后台运行，就必须要有一个前台进程，docker发现没有应用，就会自动停止，容器启动后，发现自己没有提供服务，就会立刻停止，就是没有程序了。

1. 查看日志

docker logs 镜像id #查看某镜像的日志

# 自己编写一段shell脚本

docker run -d centos /bin/sh -c “while true;do echo xzw;sleep 1;done”

# 显示日志

-tf # 显示日志

--tail number #显示日志条数

docker logs -tf --tail 10 [containerid]

1. 查看容器内部进程信息

docker top [containerid] # 查看容器内部进程信息

1. 查看容器信息

docker inspect [containerid] # 查看容器信息

1. 进入当前正在进行的容器

# 我们通常容器都是使用后台方式运行，需要进入容器，修改一些配置

docker exec -it 容器id /bin/bash

docker attach 容器id

# exec 进入容器后开启一个新的终端，可以在里面操作（常用）

# attach 进入容器正在执行的终端，不会启动新的进程

1. 从容器内拷贝文件到主机上

docker cp 容器id:容器内路径 主机路径

### 小结

attach # 当前shell下attach连接指定运行镜像  
build # 通过Dockerfile定制镜像  
commit # 提交当前容器为新的镜像  
cp # 从容器中拷贝指定文件或目录到宿主机中  
create # 创建一个新的容器，同 run 但不启动容器  
diff # 查看 docker 容器变化  
events # 从docker服务器获取容器实时事件  
exec # 在已存在的容器上运行命令  
export # 导出容器的内容流作为一个 tar 归档文件【对应 import】  
history # 展示一个镜像形成历史  
images # 列出系统当前镜像  
import # 从tar包中的内容创建一个新的文件系统映像【对应 export】  
info # 显示系统相关信息  
inspect # 查看容器详细信息  
kill # kill 指定的容器  
load # 从一个 tar 包中加载一个镜像【对应 save】  
login # 注册或者登录一个 docker 源服务器  
logout # 从当前 Docekr registry 退出  
logs # 输出当前容器日志信息

stats # 查看cpu状态

### 练习

#### 5.1练习一

docker 安装nginx

# 1、搜索镜像 （可以docker命令搜索，建议网站搜索）

docker search nginx

# 2、下载镜像 （自动选择最新版）

docker pull nginx

# 3、查看当前镜像

docker images

# 4、后台运行

docker run -d --name nginx01 -p 8002:80 nginx

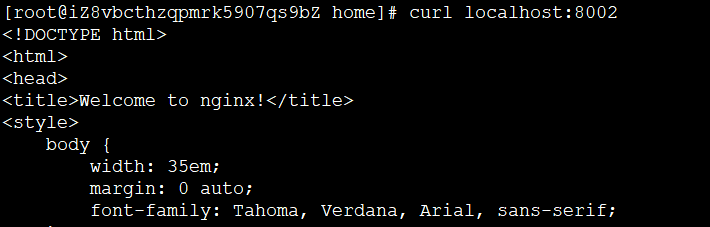
-d 为后台运行

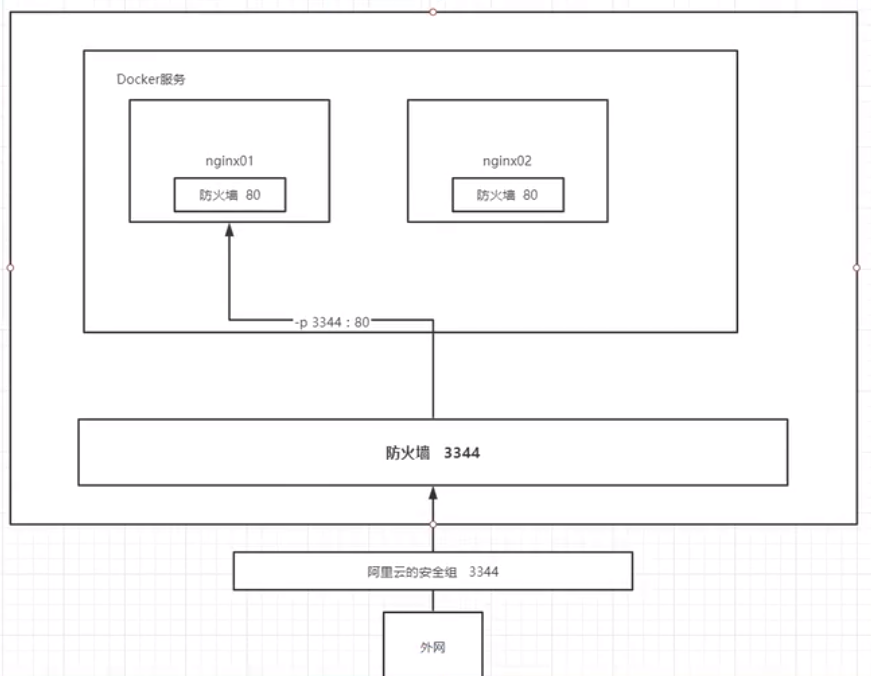
--name指定一个名字，不指定就是默认

-p为向外暴露的端口:nginx端口

# 5、测试

curl localhost:8002





#### 5.2练习二

docker 安装tomcat

# 官方的使用

docker run -it --rm tomcat:9.0

# 我们之前的启动都是后台，停止了容器之后，容器还是可以查到

# docker run -it --rm 一般用来测试，用完就删除

# 进入容器后，发现

# 1. linux命令少了

# 2. 没有webapps

# 原因：阿里云镜像的原因。默认是最小的镜像，所有不必要的都剔除掉。保证最小可运

# 行环境

#### 5.3练习三

部署es + kibana

# es暴露的端口很多！

# es十分的耗内存

# es的数据一般需要放置到安全目录，挂载

# 官方：

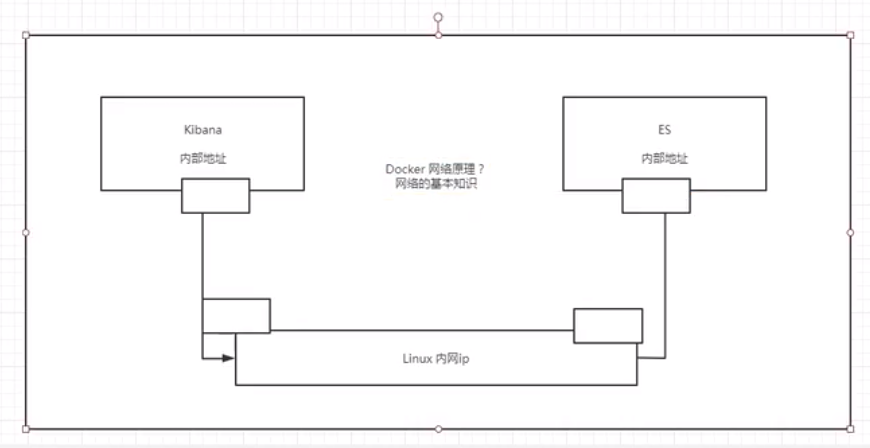
docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" elasticsearch:7.6.2

# es增加内存限制，修改配置文件 -e 环境配置修改

docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" -e ES\_JAVA\_OPTS=”-Xms64m -Xmx512m” elasticsearch:7.6.2

# 安装kibana

# 网络连接原理



### 可视化

# portainer

# Rancher（CI/CD 持续集成和持续部署）

# 使用portainer

docker run -d -p 8088:9000 --restart=always -v /var/run/docker.sock:/var/run/docker.sock --privileged=true portainer/portainer

# 什么是portainer

Docker图形化界面管理工具，提供一个后台面板供我们操作。平时不会使用

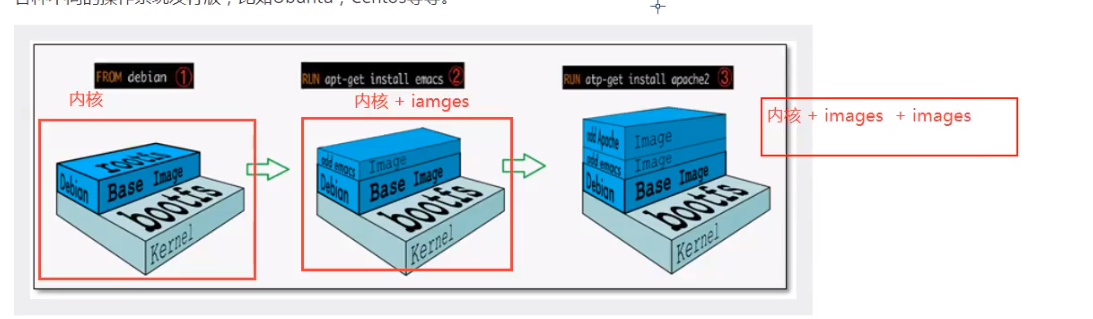
## Docker镜像讲解

### 镜像是什么

* 镜像是一种轻量级、可执行的独立软件包，用来打包软件运行环境和基于运行环境开发的软件，它包含运行某个软件所需的所有内容，包括代码、运行时、库、环境变量和配置文件。
* 所有的应用，直接打包docker镜像，就可以直接跑起来。
* 如何得到镜像：
  + 从远程仓库下载
  + 朋友拷贝
  + 自己制作一个镜像

### docker镜像加载原理





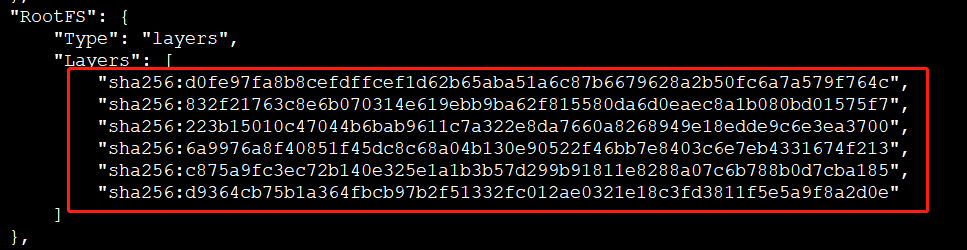


### 分层理解

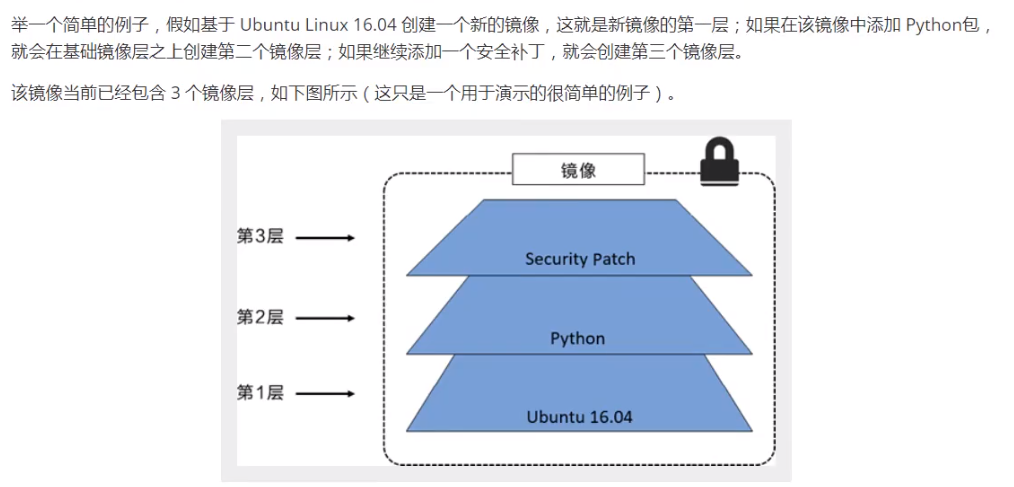
当下载一个镜像时，可以看出，是一层层下载的。

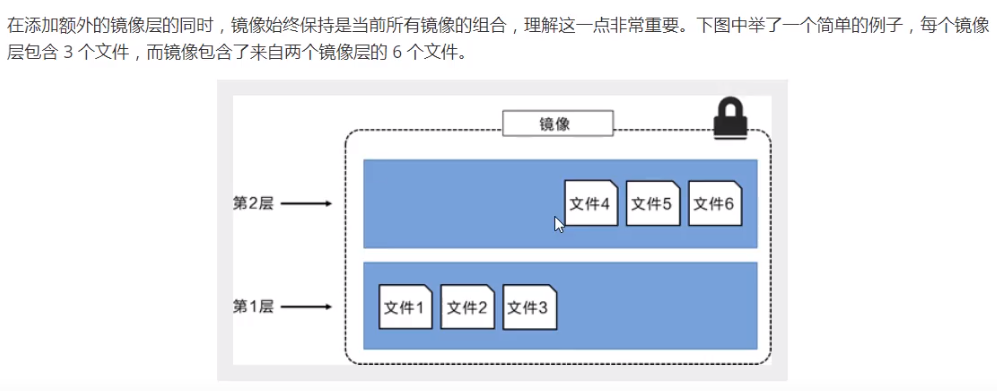


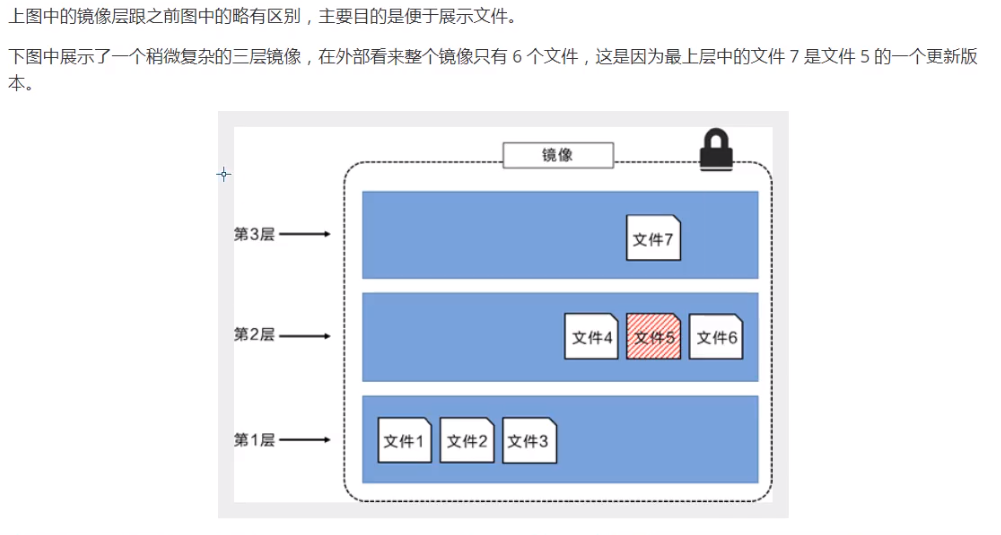
已经下载过的就不需要再下载。

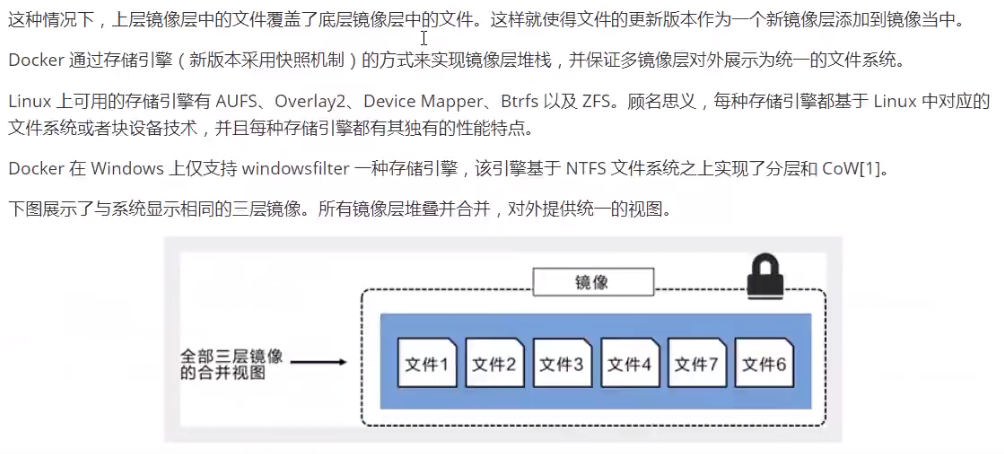


所有的Docker镜像都起始于一个基础镜像层，当进行修改或增加新内容时，就会在当前镜像层之上，创建新的镜像层。



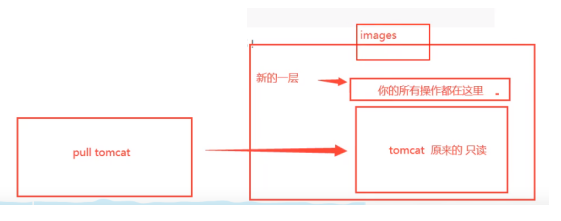






特点：

docker镜像都是只读的，当容器启动时，一个新的可写层被加载到镜像的顶部。这一层就是我们通常说的容器层，容器之下的都叫镜像层。



### commit镜像

docker commit # 提交容器称为一个新的副本

# 命令与git命令类似

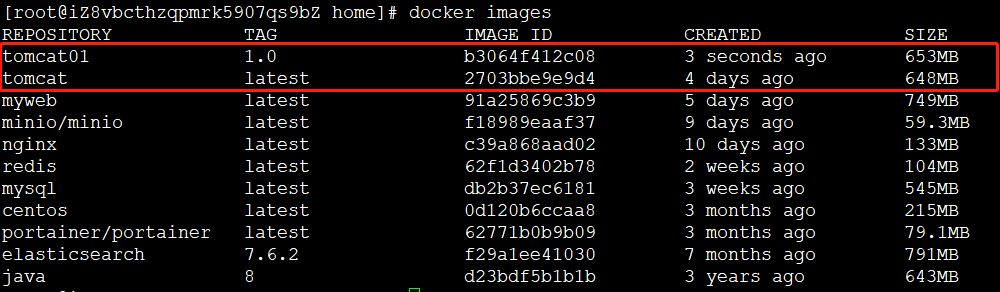
docker commit -m=”提交的描述信息” -a=”作者” 容器id 目标镜像名:[TAG]

# 启动一个默认的tomcat

# 该tomcat里是没有webapps应用

# 拷贝进webapps

# 将操作过的容器通过commit提交为一个新的镜像，相当于加了一层可写层



相当于保存当前容器状态。

## 容器数据卷

### 什么是容器数据卷

docker是将应用和环境打包成一个镜像，而数据是在容器中，如果删除容器，数据也会丢失。我们希望数据能持久化。

容器之间可以有一个数据共享的技术。Docker容器中产生的数据，同步到本地。

这就是卷技术。即目录的挂载，将我们容器内的目录，挂载到linux上。



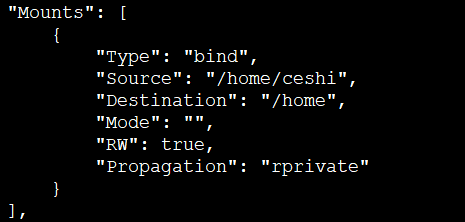
实现容器的持久化和同步操作，容器间也可以共享。

### **使用数据卷**

# 方式一：使用命令挂载 -v

docker run -it -v 主机目录:容器目录 image /bin/bash

# 通过inspect 查看挂载信息



# 测试是否挂载成功

### **配置mysql**

# 1.将mysql目录挂载出来，启动参考hub里的文档。

docker run -d -p 3310:3306 -v /home/mysql/conf:/etc/mysql/conf.d -v /home/mysql/data:/var/lib/mysql -e MYSQL\_ROOT\_PASSWORD=123456 --name mysql01 mysql:latest

# 启动成功后可以用navicat测试登录

# 再查看本地是否有数据

# 假设容器删除，但是挂载在本地的数据卷依旧没有丢失

### **具名挂载和匿名挂载**

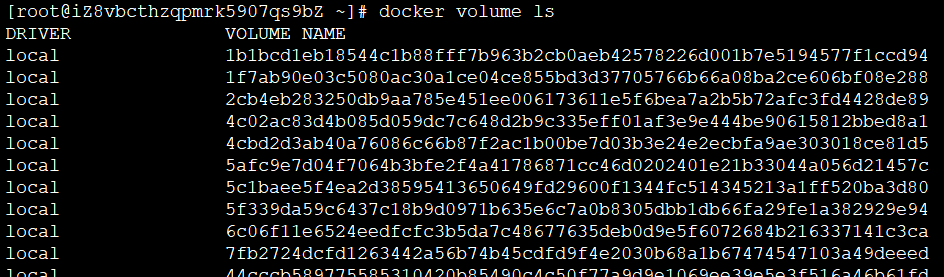
# 匿名挂载 -v 容器内路径

docker run -d -P --name nginx01 -v /etc/nginx nginx

#查看所有的卷的名字

docker volume ls

# 这种就是匿名挂载，只写了容器内的路径，没有写容器外的路径



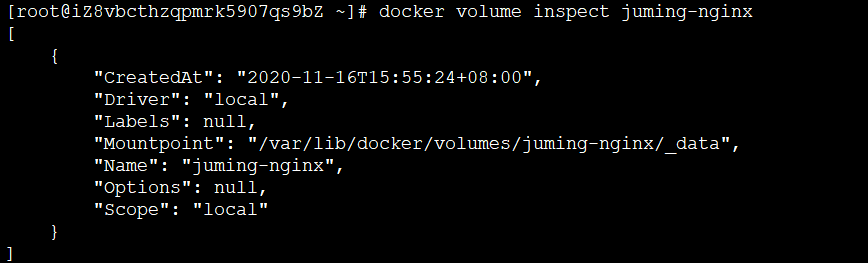
# 具名挂载，通过-v 卷名：容器内路径

docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx nginx



# 查看卷的路径

docker volume inspect juming-nginx



所有的docker容器内的卷，没有指定目录的情况下都在该目录

可以通过具名挂载可以方便的找到卷

# 如何确定具名挂载还是匿名挂载，还是指定路径挂载

-v 容器内路径 #匿名挂载

-v 卷名：容器内路径 #具名挂载

-v /宿主机路径：容器内路径 #指定路径挂载

# 拓展

# 通过 -v 容器内路径：ro rw

# 这两个是 readonly 和 readwrite

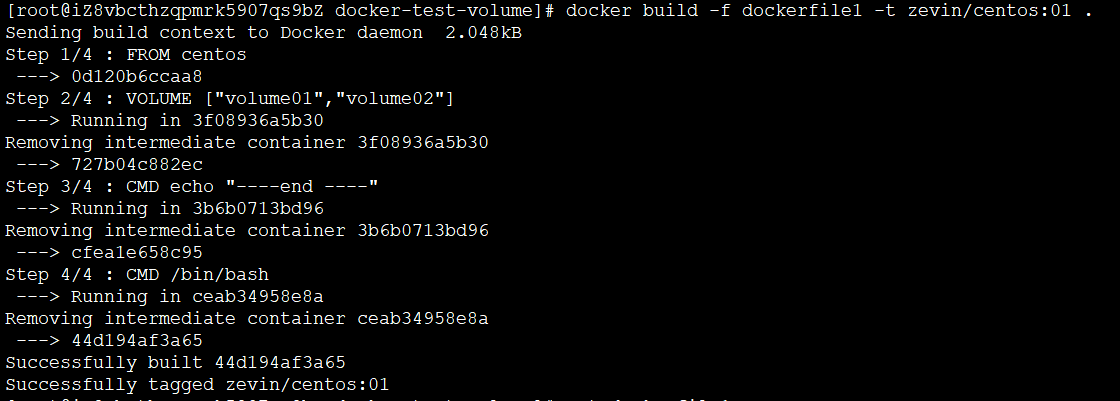
# 一旦设置了容器权限，容器对我们挂载出来的内容就有限定了。

docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx:ro nginx

docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx:rw nginx

### **初识Dockerfile**

挂载方式二：Dockerfile就是用来构建docker镜像的构建文件，就是一个命令脚本。通过这个脚本可以生成一个镜像。镜像是一层一层的，脚本一个个的命令，每个命令都是一层。



# 编写dockerfile文件，名字随意，建议Dockerfile。

# 文件中的内容，指令（大写） 参数

FROM centos

VOLUME ["volume01","volume02"]

CMD echo "----end ----"

CMD /bin/bash

# 使用docker build 命令挂载

docker build -f dockerfile1 -t zevin/centos:01 .

# 启动自己生成的容器

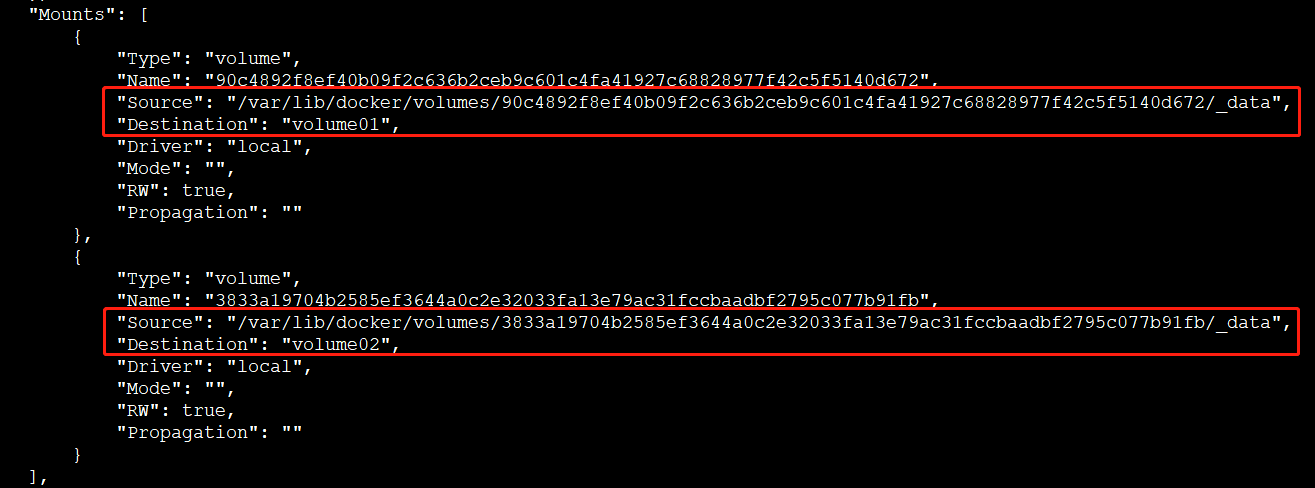


这个目录就是我们生成镜像的时候自动挂载的，数据卷目录。

这个卷和外部一定有个同步的目录。

# 查看卷挂载路径

docker inspect 容器名

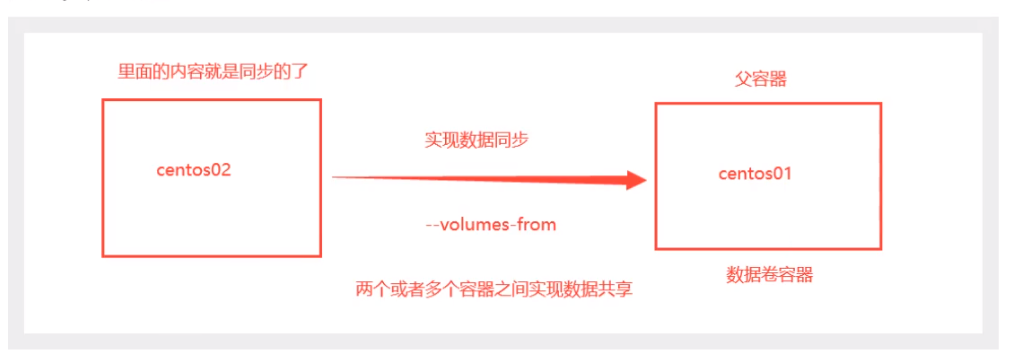


这种方式以后使用的非常多，因为通常会构建自己的镜像

假设构建镜像的时候没有挂载卷，要手动镜像挂载 -v 卷名：容器内路径

### **数据卷容器**

多个mysql同步数据



测试：启动三个容器

# 创建docker01和02两个容器

docker run -it --name docker01 zevin/centos:01

# 创建docker02容器并且将01和02实现挂载

docker run -it --name docker02 --volumes-from docker01 zevin/centos:01

# 向docker01里添加文件做测试

结论：

容器之间配置信息的传递，数据卷容器的生命周期一直持续到没有容器使用为止。

但是一旦持久化到本地，本地的数据是不会删除的。

## DockerFile

### dockerfile介绍

dockerfile是用来构建docker镜像的文件 命令 参数 脚本

构建步骤：

# 1.编写一个dockerfile文件

# 2.docker build构建成为一个镜像

# 3.docker run 运行镜像

# 4.docker push 发布镜像（Dockerhub、阿里云）

### **dockerfile构建过程**

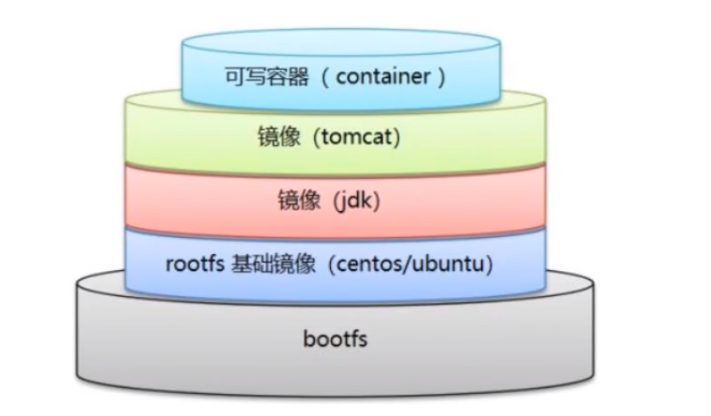
# 基础知识：

# 1.每个保留关键字（指令）都是必须是大写字母

# 2.执行从上到下顺序执行

# 3.#表示注释

# 4.每一个指令都会创建提交一个新的镜像层，并提交



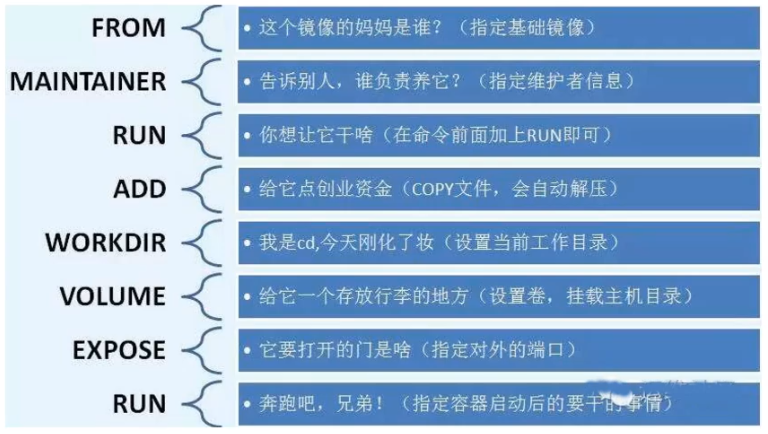
dockerfile是面向开发的，我们以后要发布项目，做镜像，就需要编写dockerfile文件，这个文件十分简单。docker镜像逐渐成为企业交付的标准，必须要掌握。

# dockerfile：构建文件，定义了一切的步骤，源代码

# dockerimages:通过dockerfile构建生成的镜像，最终发布和运行的产品，原来是jar,war

# docker容器：容器就是镜像运行起来提供服务。

### **dockerfile指令**



FROM #基础镜像，一切从这里开始构建

MAINTAINER #镜像是谁写的，姓名+邮箱

RUN #镜像构建的时候需要运行的命令

ADD #步骤，tomcat镜像，这个tomcat压缩包，添加。压缩包会 自动解压

WORKDIR #镜像的工作目录

VOLUME #挂载的目录

EXPOST #保留端口配置

CMD #指定容器启动的时候要运行的命令，只有最后一个会生效，可 被替代

ENTRYPOINT #指定容器启动的时候要运行的命令，可以追加命令。

ONBUILD #当构建一个被继承Dockerfile，这个时候回运行ONBUILD 指令，触发指令

COPY #类似ADD，将我们的文件拷贝到镜像中

ENV #构建的时候设置环境变量

# CMD和ENTRYPOINT

比如CMD里写的ls -a， 命令写docker run -l，那么就会替换成

docker run ls -a

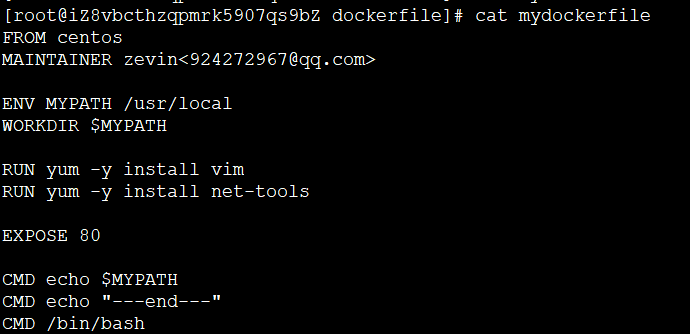
而ENTRYPOINT则后面追加

docker run -l ls -a

### **练习指令**

创建一个自己的centos

# 1、编写dockerfile的文件



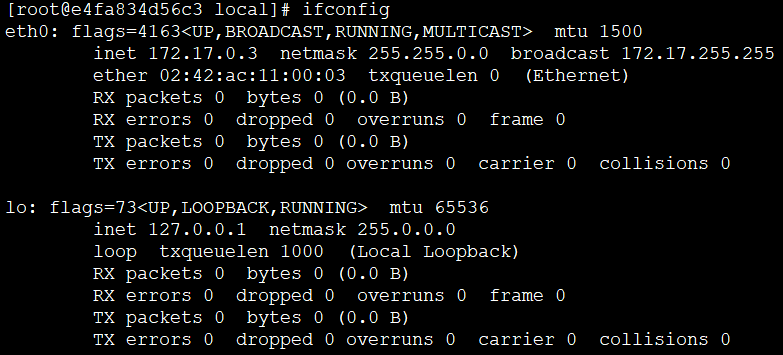
# 2、通过build进行镜像构建

docker build -f dockerfile文件路径 -t 镜像名:[TAG] .

docker build -f mydockerfile -t mycentos:0.1 .

# 3、测试运行

docker run -it mycentos:0.1

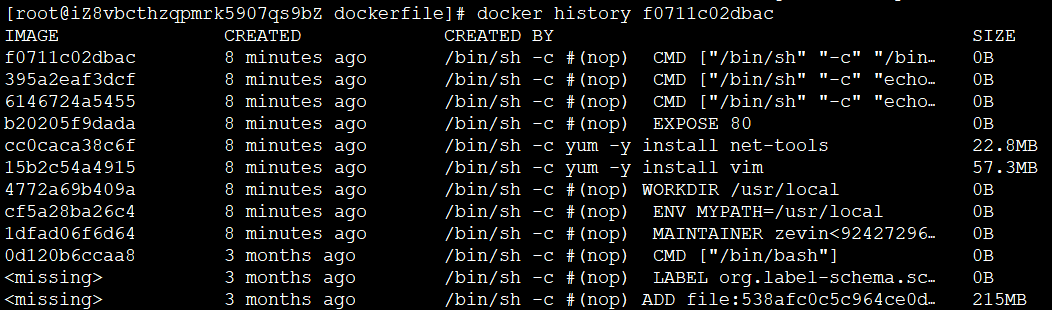


[root@iZ8vbcthzqpmrk5907qs9bZ dockerfile]# docker run -it mycentos:0.1

[root@e4fa834d56c3 local]#

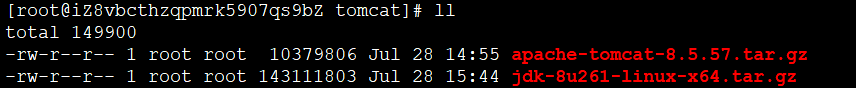
工作目录也默认为/usr/local

#可以通过docker history imageid 来列出本地的变更历史



做一个tomcat镜像

# 1.准备镜像文件，需要jdk压缩包



# 2.编写dockerfile文件，使用官方命名Dockerfile，build会自动寻找这个文件

FROM centos

MAINTAINER zevin<924272967@qq.com>

COPY readme.txt /usr/local/readme.txt

ADD jdk-8u261-linux-x64.tar.gz /usr/local/

ADD apache-tomcat-8.5.57.tar.gz /usr/local/

RUN yum -y install vim

ENV MYPATH /usr/local

WORKDIR $MYPATH

ENV JAVA\_HOME /usr/local/jdk1.8.0\_11

ENV CLASSPATH $JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

ENV CATALINA\_HOME /usr/local/apache-tomcat-8.5.57

ENV CATALINA\_BASH /usr/local/apache-tomcat-8.5.57

ENV PATH $PATH:$JAVA\_HOME/bin:$CATALINA\_HOME/lib:$CATALINA\_BASH/bin

EXPOSE 8080

CMD /usr/local/apache-tomcat-8.5.57/bin/startup.sh && tail -F /url/local/apache-tomcat-8.5.57/bin/logs/catalina.out

# 3.构建镜像

docker build -t diytomcat .

# 4.启动容器

docker run -d -p 9092:8080 --name zevintomcat -v /home/zevin/build/tomcat/test/:/url/local/apache-tomcat-8.5.57/webapps/test -v /home/zevin/build/tomcat/tomcatlogs/:/url/local/apache-tomcat-8.5.57/logs diytomcat

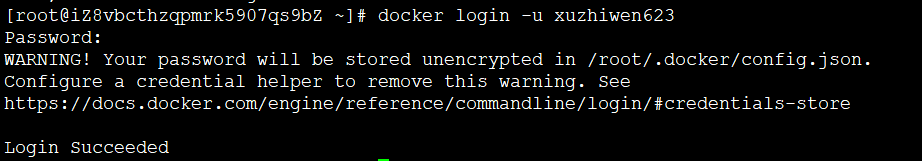
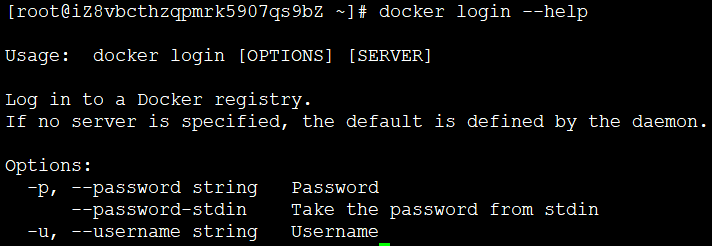
# 5.通过挂载发布项目

### **发布镜像**

# 1.在https://hub.docker.com/ 上注册自己的账号

# 2.确定这个账号可以登录

# 3.在服务器上提交自己的镜像，先登录 docker login



# 4.提交镜像 docker push

docker push xuzhiwen623/diytomcat:1.0

出现问题？

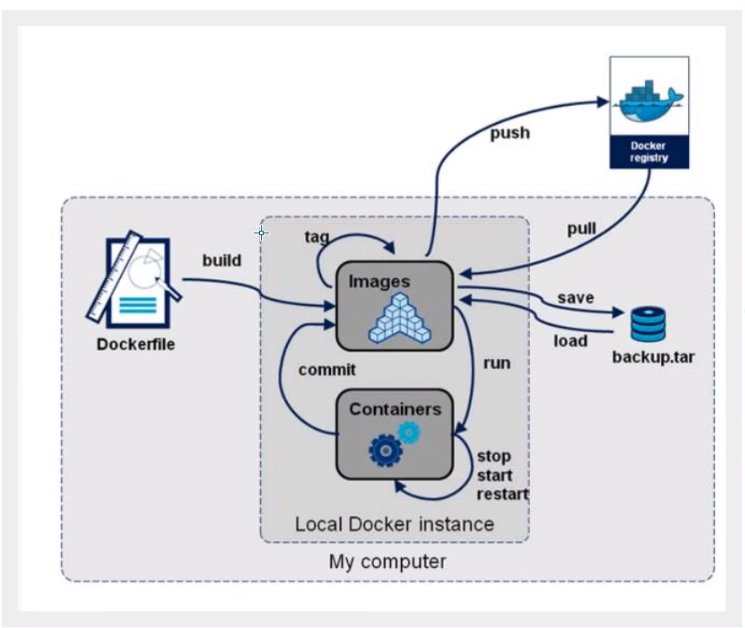
The push refers to repository [docker.io/xuzhiwen623/diytomcat]

# 5.解决，增加一个tag，再提交

docker tag 96a9278aee65 xuzhiwen623/diytomcat:1.0

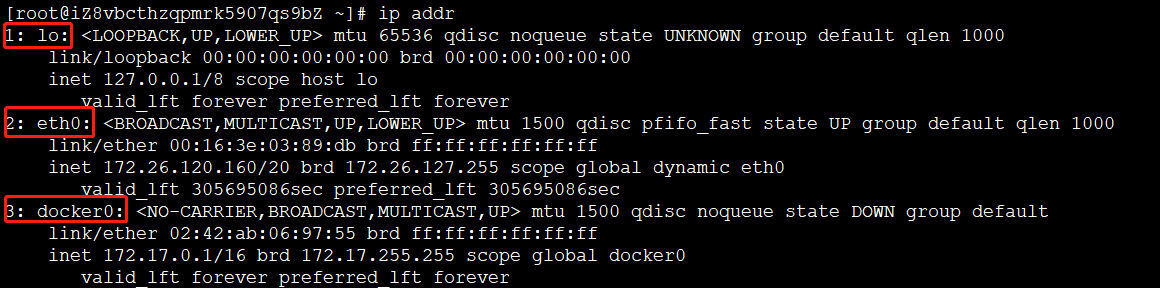
发布到阿里云上，参考阿里云官方文档

### **总结**



## Docker网络

### 理解docker0



lo: 本机回环地址

eth0: 阿里云内网地址

docker0: docker生成的网卡

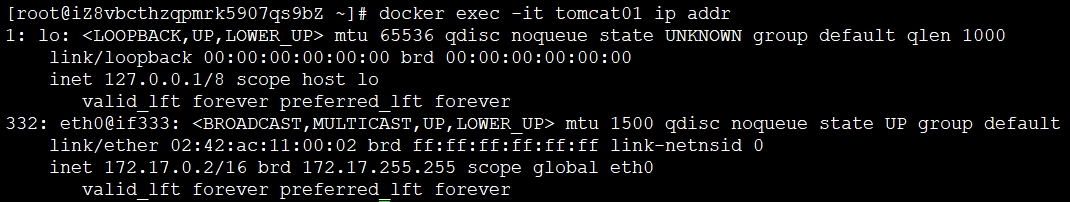
# 问题：docker如何处理容器网络访问的？

做个测试

# 运行一个容器

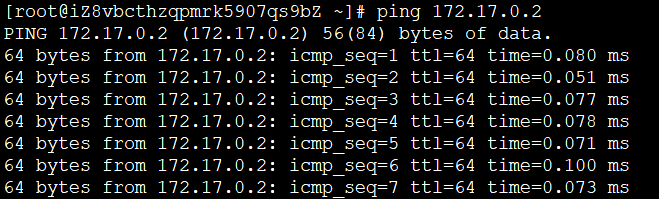
docker run -d -P --name tomcat01 tomcat

# 查看容器内部网络地址



发现容器启动的时候会得到一个eth0@if333的ip地址，这个就是docker分配的。

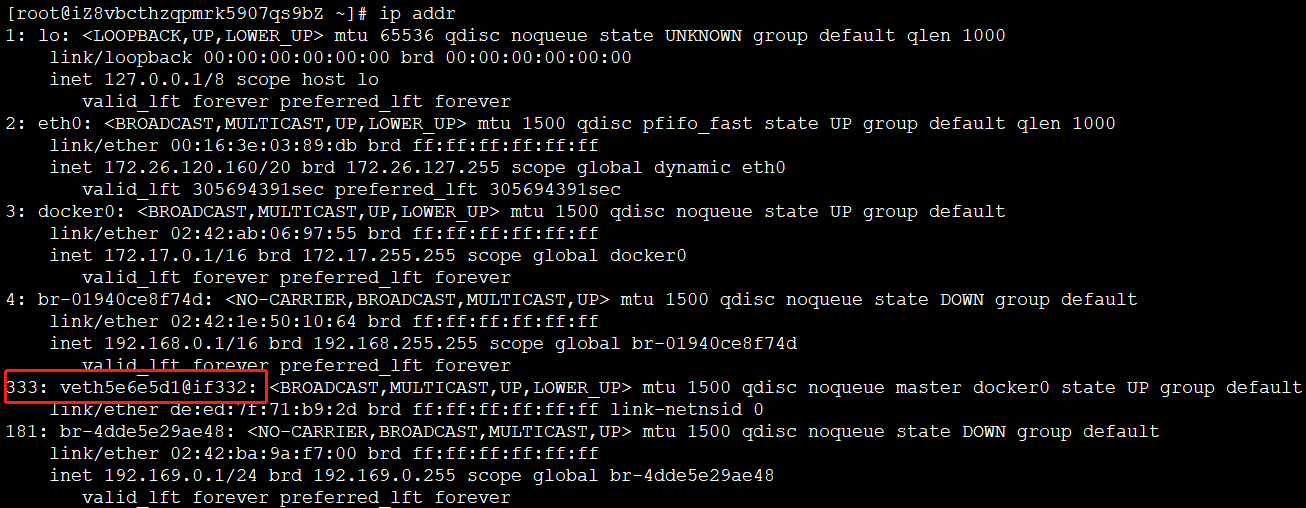
# 思考：linux能不能ping通容器内部？



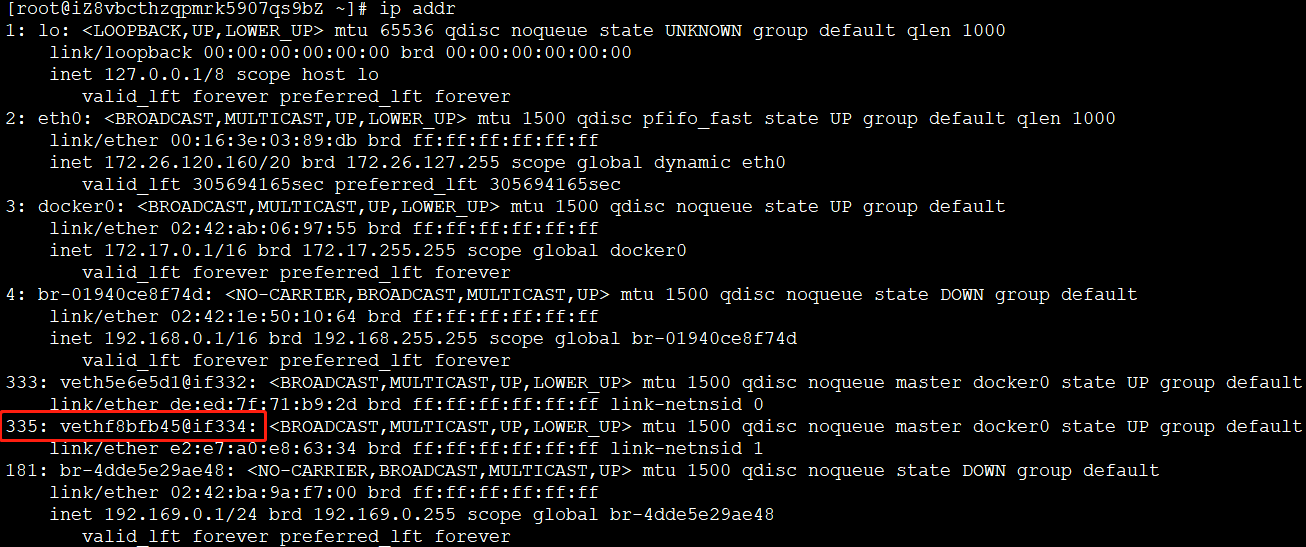
发现可以Ping通。

我们每启动一个docker容器，docker就会给docker容器分配一个ip，我们只要安装了docker ，就会有一个网卡docker0，这个是桥接模式，使用的技术是evth-pair技术。

# 再次测试ip addr 出现了一个新的333，后面绑定的332，前面332绑定333

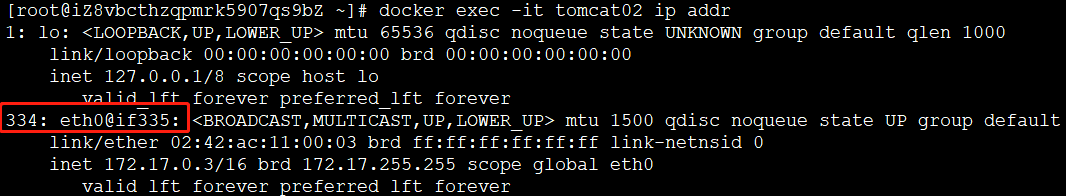


# 再启动一个容器测试



发现又多了一对网卡335，334

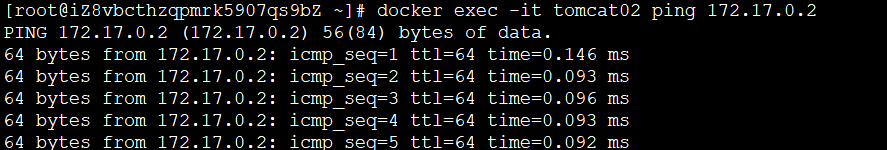
容器内部网卡也是对应的



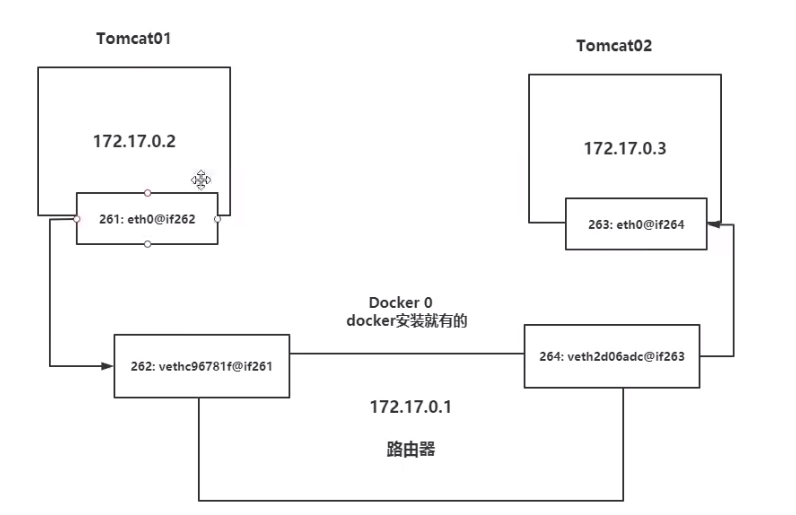
# 这个容器带来的网卡都是一对一对的，这个技术就是evth-pair技术

evth-pair 就是一对的虚拟设备接口，他们都是成对出现的，一段连着协议，一段彼此相连。正因为有这个特性，evth-pair一般充当一个桥梁，连接各种虚拟网络设备。

# 测试两个容器能不能直接ping通，可以ping通

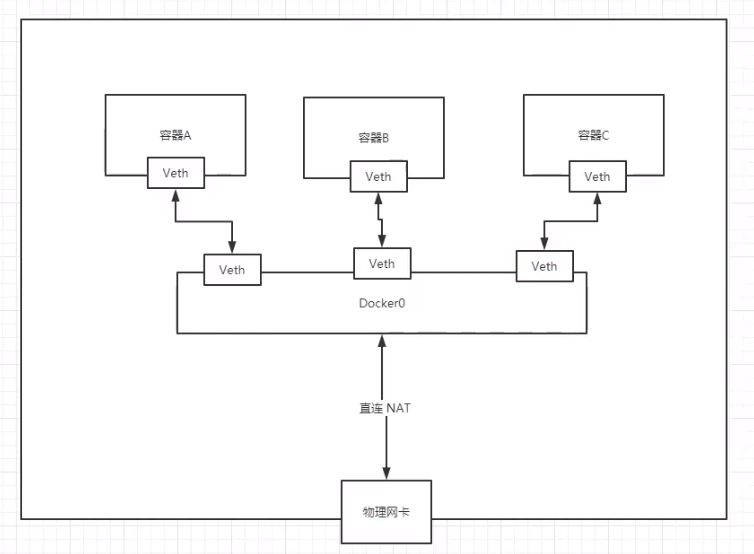


# 容器之间是可以互相ping通的



所有的容器不指定网络的情况下，都是docker0路由的，docker会给我们的容器分配一个默认的可用ip

Docker使用的是linux的桥接，宿主机中是一个Docker容器的网桥docker0



Docker中的所有网络接口都是虚拟的。虚拟的转发效率高

只要容器删除，对应网桥一对就没了

### --link

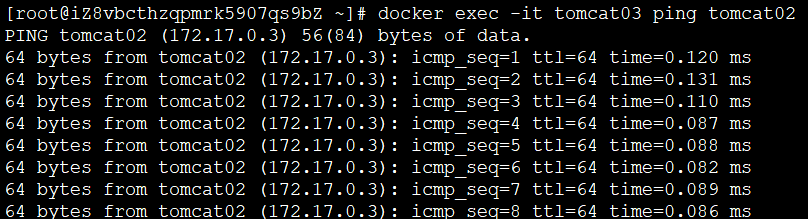
# 思考：编写了一个微服务，database url = ip:，项目不重启，数据库ip换掉了，希望可以通过名字来访问容器

# 测试，直接ping看行不行



# 解决：--link来指定要连通的容器

docker run -d -P --name tomcat03 --link tomcat02 tomcat



发现可以ping通，可以通过服务名来ping通

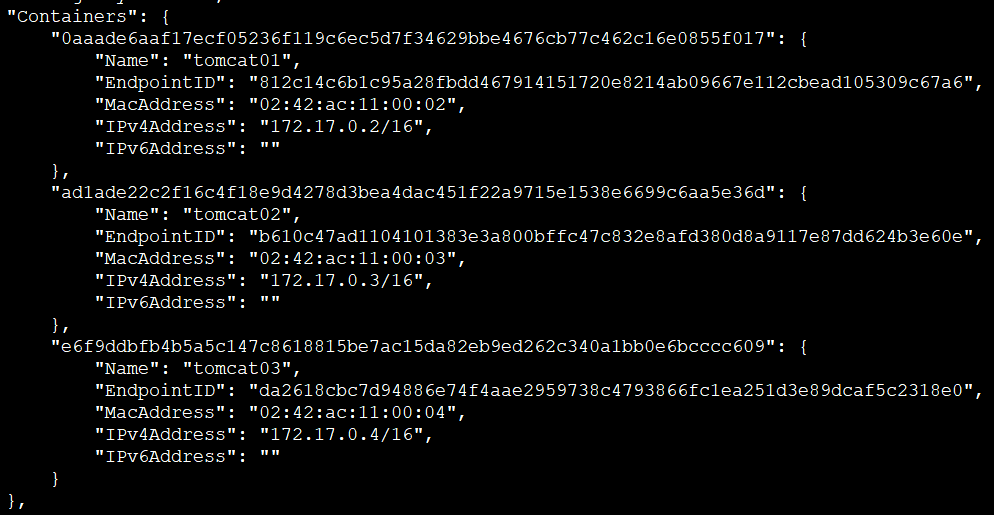
# 思考，反向能ping通吗？



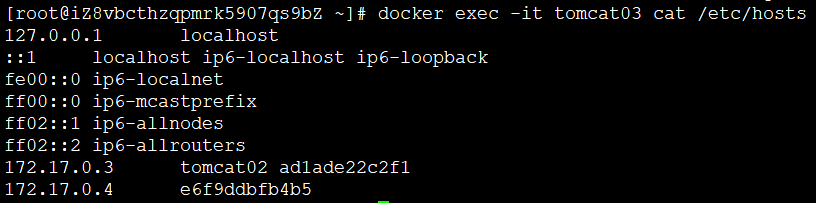
发现需要配置才能ping通

# 使用docker network 来查看网络信息

docker network inspect



# 通过查看hosts配置发现，其实这个tomcat03就是在本地配置了tomcat02的配置



# 总结

--link 就是我们在hosts配置中增加了一个绑定

现在在开发中不建议使用--link

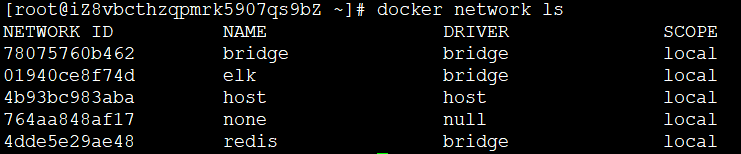
需要自定义网络，不是docker0

docker0不支持容器名连接访问

### 自定义网络

# 查看所有docker网络

docker network ls



# 网络模式

bridge #桥接 docker

none #不配置网络

host #和宿主机共享网络

container #容器内可以网络连通（用的少）

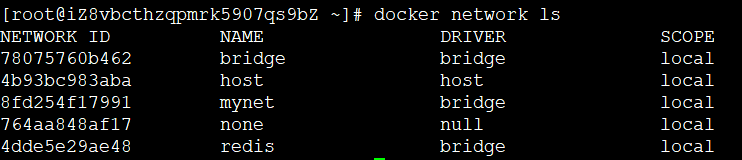
# 测试 默认 --net bridge，这个就是docker0

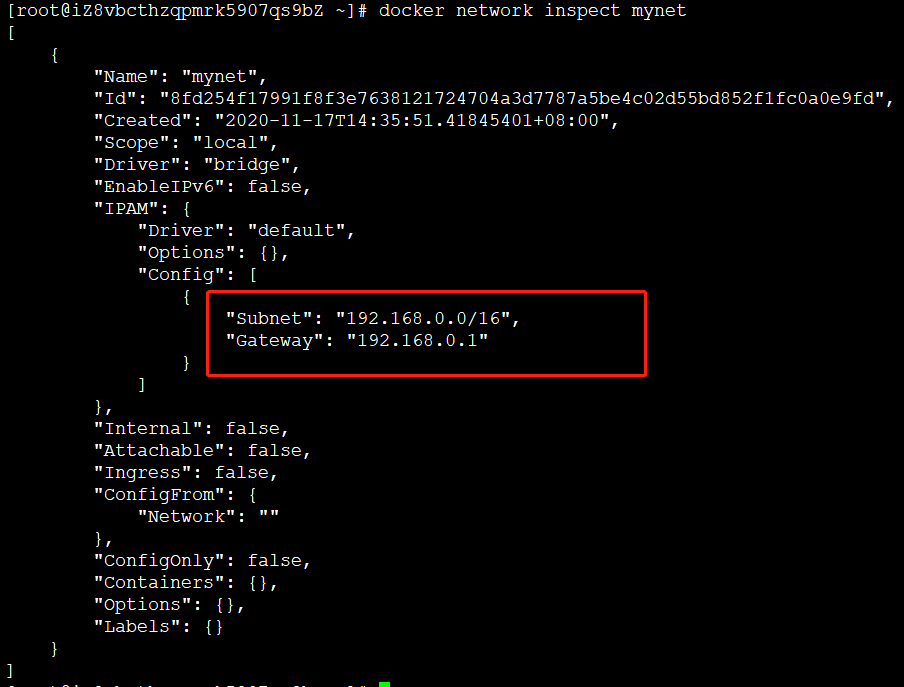
docker run -d -P --name tomcat01 --net bridge tomcat

# 自定义网络

docker network create --driver bridge --subnet 192.168.0.0/16 --gateway 192.168.0.1 mynet

--driver

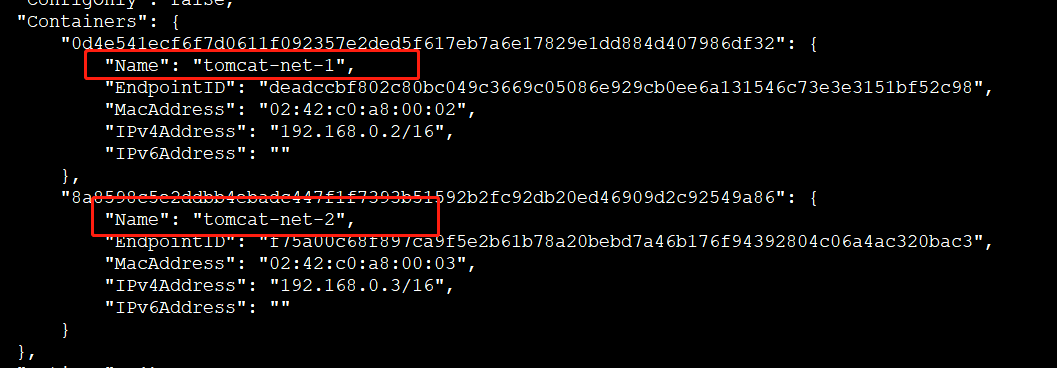




# 将tomcat放入自己的网络

docker run -d -P --name tomcat-net-1 --net mynet tomcat

docker run -d -P --name tomcat-net-2 --net mynet tomcat



可以看到自己的网络里有两个tomcat容器了。

测试Ping，两边都可以互相ping通

# 好处

保证不同的集群使用不同的网络，保证集群是安全和健康的。

### 网络连通

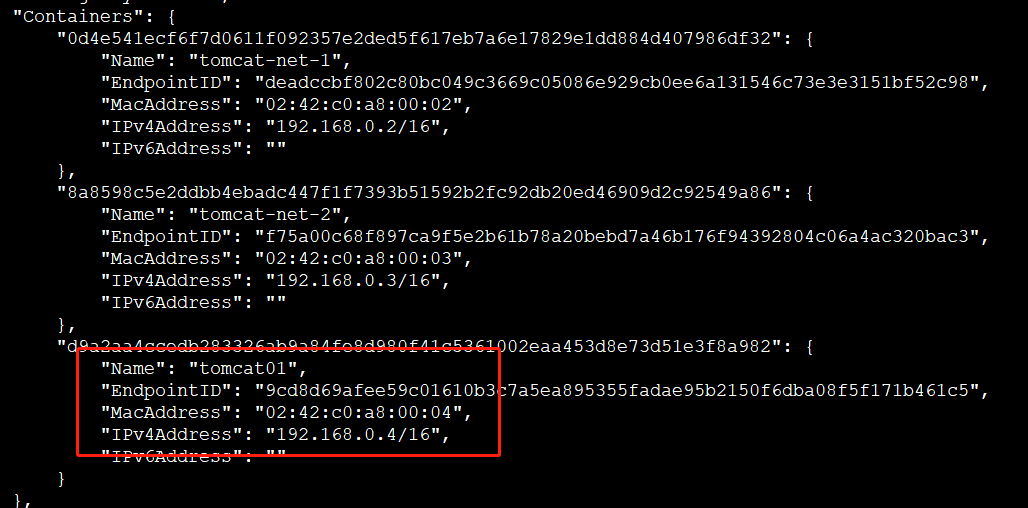
# 不同网段的容器怎么互通？

使用 docker network connect

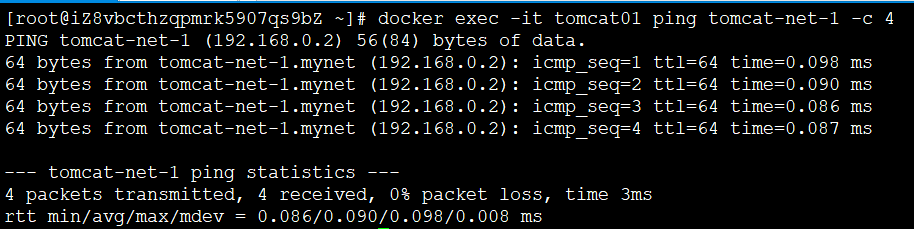
# 将一个容器连在某网段上

docker network connect mynet tomcat01

# 通过 docker network inspect 看到连到该网段上了



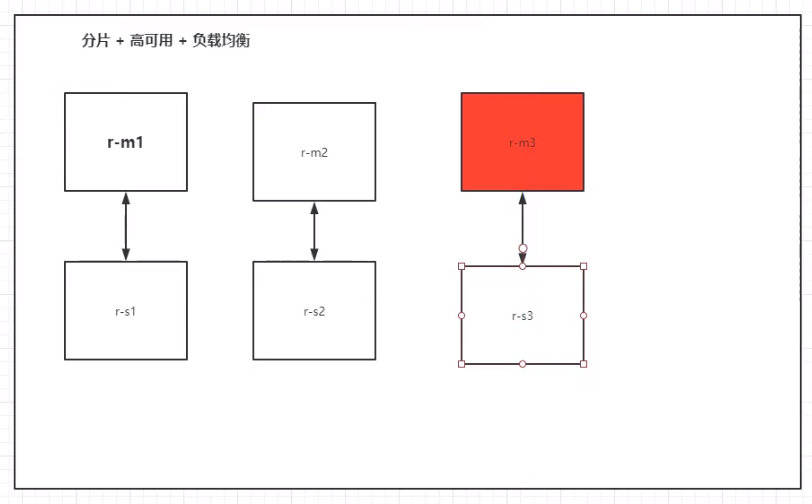
这样一个容器就有两个ip地址



# 结论

假设要跨网络操作别人，就要使用docker network connect来打通

### 部署redis集群



# 创建redis网卡

docker network create redis --subnet 172.38.0.0/16

# 编写shell 脚本

for port in $(seq 1 6); \

do \

mkdir -p /mydata/redis/node-${port}/conf

touch /mydata/redis/node-${port}/conf/redis.conf

cat<<EOF>/mydata/redis/node-${port}/conf/redis.conf

port 6379

bind 0.0.0.0

cluster-enabled yes

cluster-config-file nodes.conf

cluster-node-timeout 5000

cluster-announce-ip 172.38.0.1${port}

cluster-announce-port 6379

cluster-announce-bus-port 16379

appendonly yes

EOF

done

# 启动容器

docker run -p 6371:6379 -p 16371:16379 --name redis-1 -v /mydata/redis/node-1/data:/data -v /mydata/redis/node-1/conf/redis.conf:/etc/redis/redis.conf -d --net redis --ip 172.38.0.11 redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf

docker run -p 6372:6379 -p 16372:16379 --name redis-2 -v /mydata/redis/node-2/data:/data -v /mydata/redis/node-2/conf/redis.conf:/etc/redis/redis.conf -d --net redis --ip 172.38.0.12 redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf

docker run -p 6373:6379 -p 16373:16379 --name redis-3 -v /mydata/redis/node-3/data:/data -v /mydata/redis/node-3/conf/redis.conf:/etc/redis/redis.conf -d --net redis --ip 172.38.0.13 redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf

docker run -p 6374:6379 -p 16374:16379 --name redis-4 -v /mydata/redis/node-4/data:/data -v /mydata/redis/node-4/conf/redis.conf:/etc/redis/redis.conf -d --net redis --ip 172.38.0.14 redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf

docker run -p 6375:6379 -p 16375:16379 --name redis-5 -v /mydata/redis/node-5/data:/data -v /mydata/redis/node-5/conf/redis.conf:/etc/redis/redis.conf -d --net redis --ip 172.38.0.15 redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf

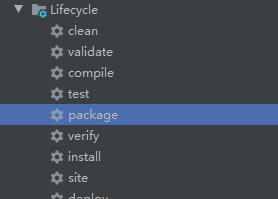
docker run -p 6376:6379 -p 16376:16379 --name redis-6 -v /mydata/redis/node-6/data:/data -v /mydata/redis/node-6/conf/redis.conf:/etc/redis/redis.conf -d --net redis --ip 172.38.0.16 redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf

# 创建集群，随便进一个容器

redis-cli --cluster create 172.38.0.11:6379 172.38.0.12:6379 172.38.0.13:6379 172.38.0.14:6379 172.38.0.15:6379 172.38.0.16:6379 --cluster-replicas 1

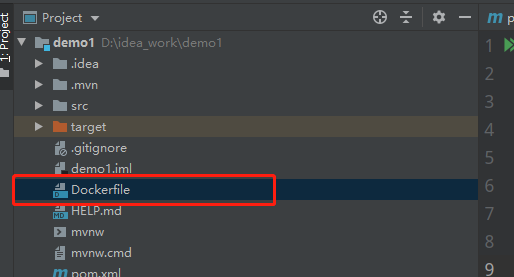
## springboot微服务打包docker镜像

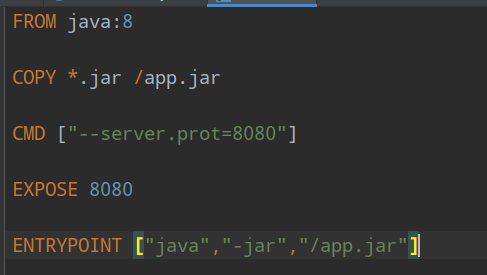
# 1.打包项目



# 2.编写dockerfile文件

创建Dockerfile文件，编写文件





# 3.将jar包和dockerfile放在同一目录下



# 4.构建docker镜像

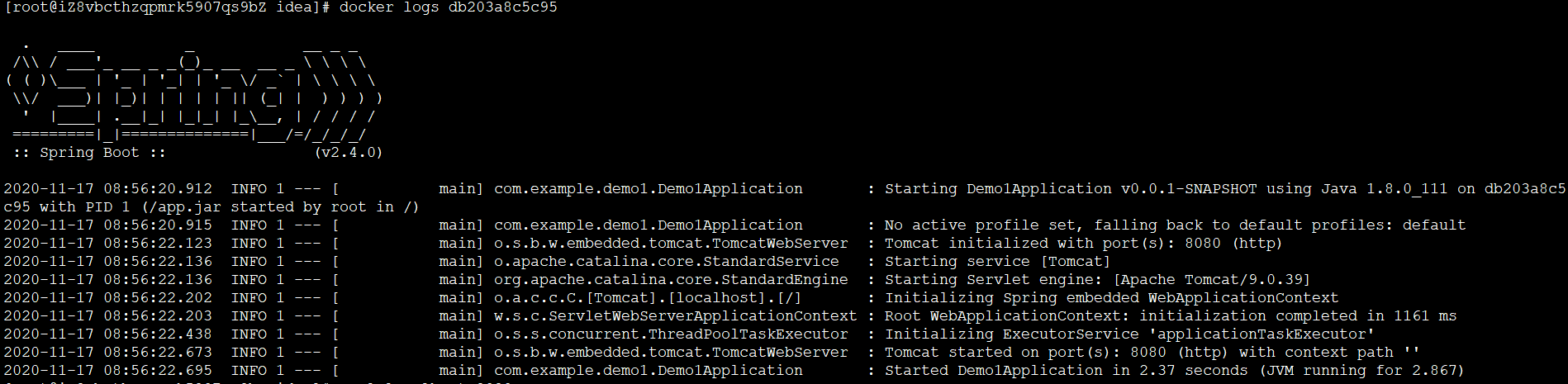
docker build -t zevinproject .

# 5.运行容器

docker run -d -p 8080:8080 --name idea zevinproject

# 6.可以通过查看日志查看是否启动成功

docker logs



以后使用docker，给别人交付的就是一个镜像

## Docker Compose

### 1.简介

之前，是一个服务一个服务手动操作，Dockerfile build run 这样操作。

假设现在微服务项目里有很多服务，单独操作很麻烦，同时依赖关系很复杂。

Docker Compose就是定义/编排运行多个容器。

# 官方文档介绍

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see [the list of features](https://docs.docker.com/compose/" \l "features).

Compose works in all environments: production, staging, development, testing, as well as CI workflows. You can learn more about each case in [Common Use Cases](https://docs.docker.com/compose/" \l "common-use-cases).

Using Compose is basically a three-step process:

1.Define your app’s environment with a Dockerfile so it can be reproduced anywhere.

2.Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.

3.Run docker-compose up and Compose starts and runs your entire app.

# 理解

# 1.Compose是docker官方的开源项目。需要安装

# 2.Dockerfile让程序在任何地方运行，但是每个容器都单独build run太麻烦。

# 3.Compose文件

version: "3.8"

services:

web:

build: .

ports:

- "5000:5000"

volumes:

- .:/code

- logvolume01:/var/log

links:

- redis

redis:

image: redisvolumes:

logvolume01: {}

# Compose:重要的概念

# 1.服务services,容器。应用（web ,redis,mysql..）

# 2.项目project，一组关联的容器。

### 安装

# 参考官方文档

# 1.下载

curl -L "https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

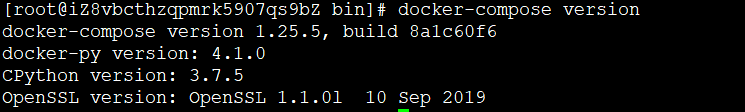
上面这个可能很慢

curl -L https://get.daocloud.io/docker/compose/releases/download/1.25.5/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose



# 2.授权

chmod +x /usr/local/bin/docker-compose



### 体验

官网getting started

### yaml规则

version:”” #版本

service: #服务

服务1:web

服务配置

images

build

network

....

服务2:redis

# 其他配置 网络/卷、全局规则

volumes:

networks:

configs:

# 注意depends\_on 下面那个指的是web项目需要redis和db，就会先启动db和redis

version: "3.8"services:

web:

build: .

depends\_on:

- db

- redis

redis:

image: redis

db:

image: postgres