DOCKER

DOCKER学习：

- DOCKER概述;
- DOCKER安装
- DOCKER命令
  - 镜像命令
  - 容器命令
  - 操作命令
  - 。。。
- Docker镜像
- 容器的数据卷
- DockerFile
- Docker网络原理
- IDEA整合DOCKER
- Docker Compose
- Docker Swarm
- CI\CD jenkins

知道的越多不知道的越多

Docker概述

Docker为什么会出现?

一款产品，开发-----上线，两套环境！应用环境！应用配置！

开发-------运维。 问题：我在我的电脑上可以使用！版本跟新导导致服务不可用！对于运维来说考验十分大

开发即运维！！！

环境配置十分麻烦，每一个机器都要配置环境！！ 费时费力！！

发布一个项目jar+(REdis Mysql jdk ES),项目带上环境安装打包！！

之前在服务器配置一个应用的环境（REdis Mysql jdk ES Hadoop）,配置超麻烦，不能够跨平台！！

windows，连最后发布到linux

传统：开发jar打包， 运维来做

现在：开发打包部署上线，一套流程做完

java------apk-----发布(应用商店)----张三使用apk--------安装即可用

java------jar(环境)-------打包项目带环境(镜像！)--------Docker仓库：商店----------下载我么发布的镜像直接运行即可

Docker对于以上问题产生了一个解决方案

docker思想来自一个集装箱

JRE ----多个应用（端口冲突）--------因为原来都是交叉的

隔离：Docker核心思想！打包装箱！每个箱子都是互相隔离的

水果　　生化武器

Docker通过隔离机制，可以将服务器利用到极致

本质：所有的技术都是因为出现一些问题才会出现的；

Docker的历史：

2010 ，几个搞IT的年轻人在美国成立了一家公司dotcloud

做一些pass的云计算服务,LXC有关的容器技术

他们将自己的容器化技术命名就是Docker！

刚诞生的时候，没有引起行业的注意！！dotcloub，就活不下去！！

开源

2013年，Docker开源；

Docker越来越多的人发现了Docker的优点，每个月都会更新

2014年4月9日，Docker1.0发布

Docker这么火的原因?

它相对虚拟机十分轻巧

在容器技术出现之前，我们都是使用虚拟机

虚拟机运行：在windows或者mac中安装一个虚拟机软件vmware，可以虚拟一台或多台电脑！占用内存太大！笨重！

虚拟机也是属于虚拟化技术，Docker容器技术，也是一种虚拟化技术！

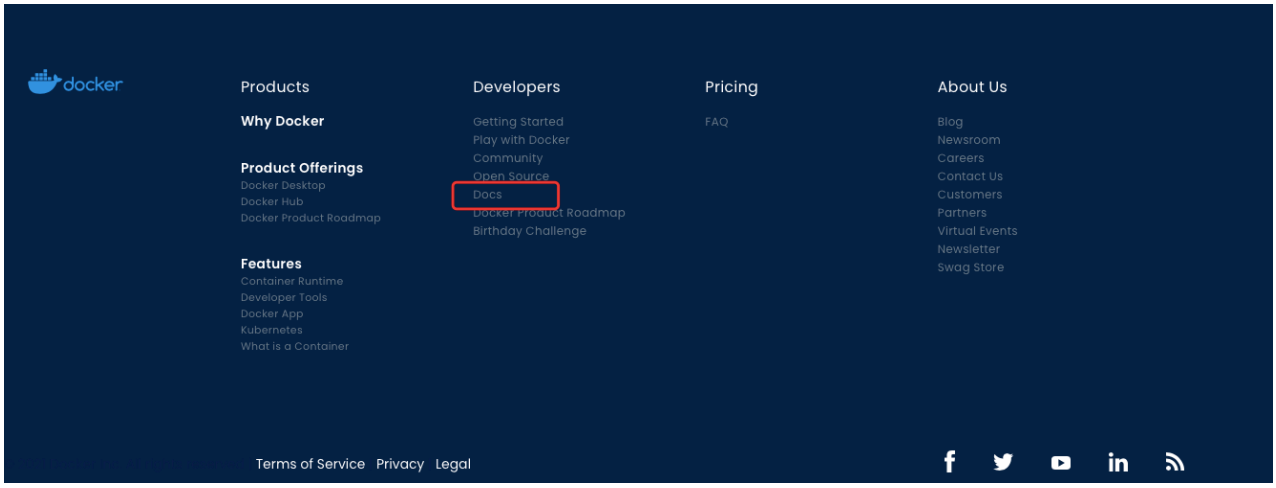
```
vm . linux centos(原生镜像)，隔离，就需要开启多个虚拟机 ！几个G       启动需要几分钟

Docker，隔离，镜像(最核心的环境 4m，jdk，mysql )十分的小巧，运行镜像就可以了！小巧！几M
  KB     秒级启动
```

到现在所有开发人员必须会

> 聊聊Docker

Docker是基于GO语言开发的！ 开源项目

官网：https://www.docker.com

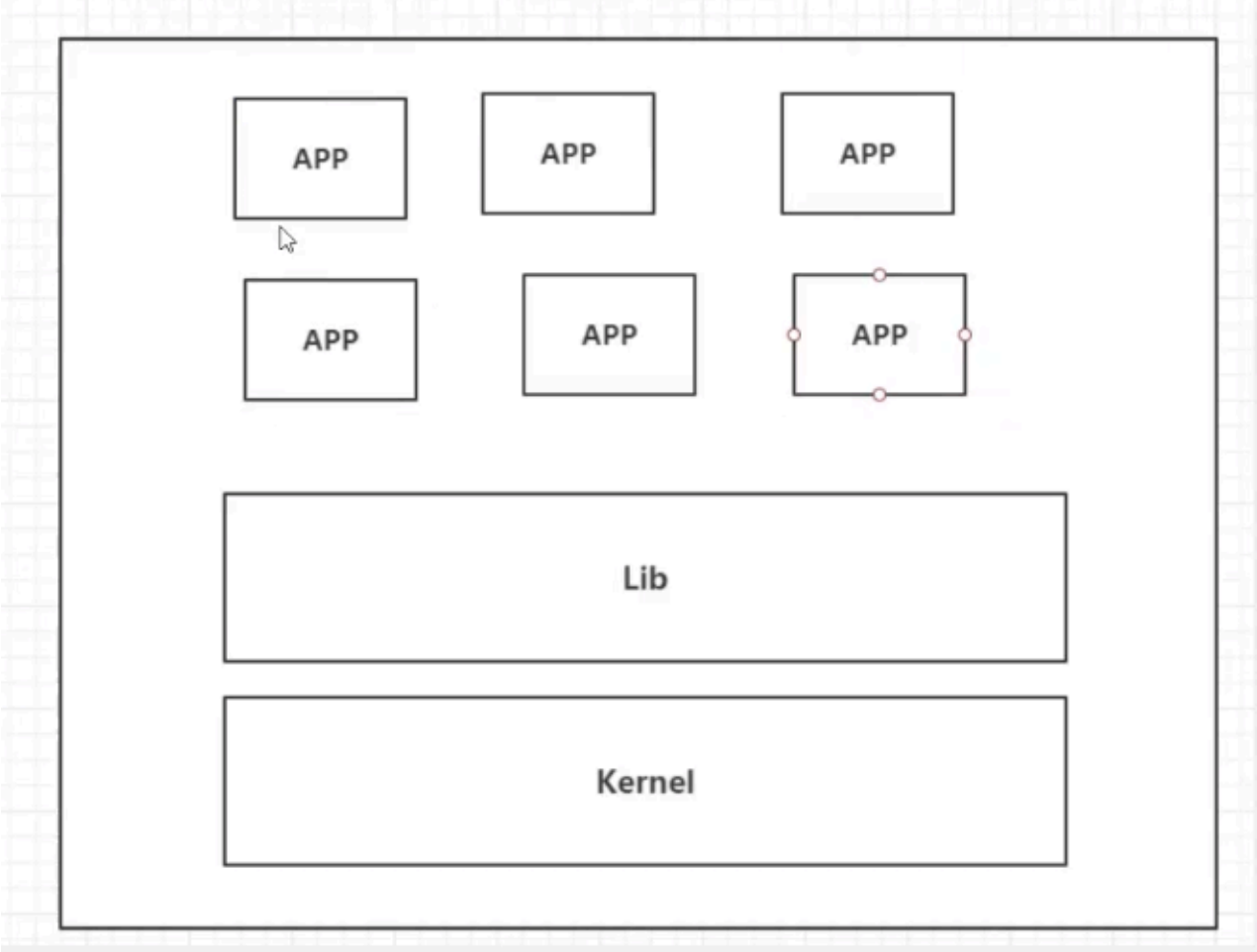


文档地址：https://docs.docker.com Docker的文档超级详细的！

仓库地址：https://registry.hub.docker.com

Docker能干嘛?

> 之前的虚拟机技术

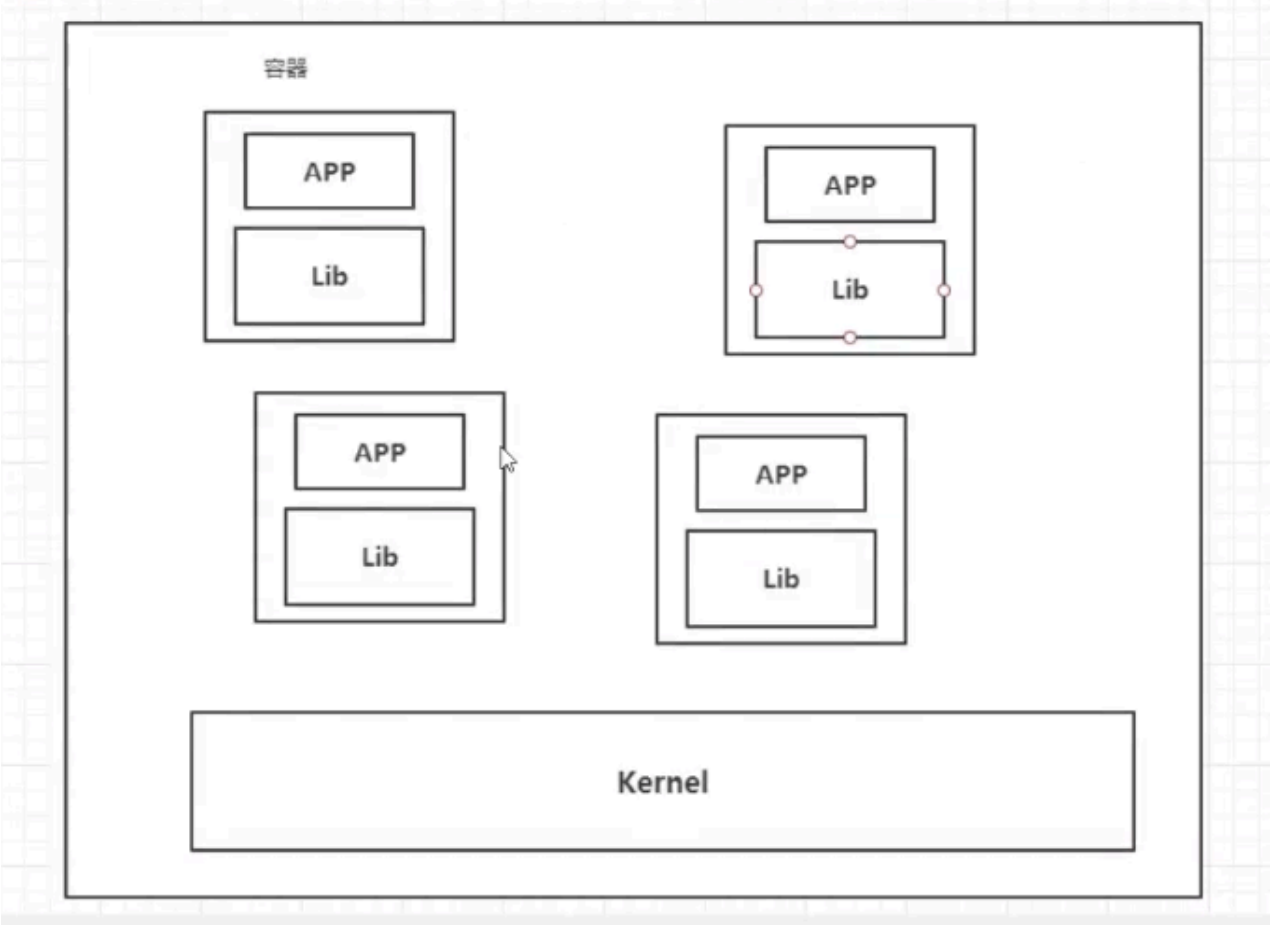


虚拟机技术的缺点：

1. 占用资源十分多
2. 冗余步骤十分多
3. 启动很慢

> 容器化技术

容器化技术不是模拟的完整的操作系统



比较Docker和虚拟机技术的不同：

- 传统虚拟机，虚拟出一个硬件 ，运行一个完整的操作系统，然后在这个系统上安装和运行软件
- 容器的应用直接运行在宿主机的内容，容器是没有自己的内核，也没有虚拟我们的硬件，所以就轻便了
- 每个容器件恶事互相隔离的，每个容器内都有一个属于自己的文件系统

> DebOps(开发，运维)

更快速的交付和部署

传统：一堆的帮助文档，安装程序

Docker：打包镜像，发布测试，一键运行

更便捷的升级和扩缩容

使用Docker之后，我们部署应用就和搭积木一样

项目打包为一个镜像，扩展　服务器A

更简单的系统运维

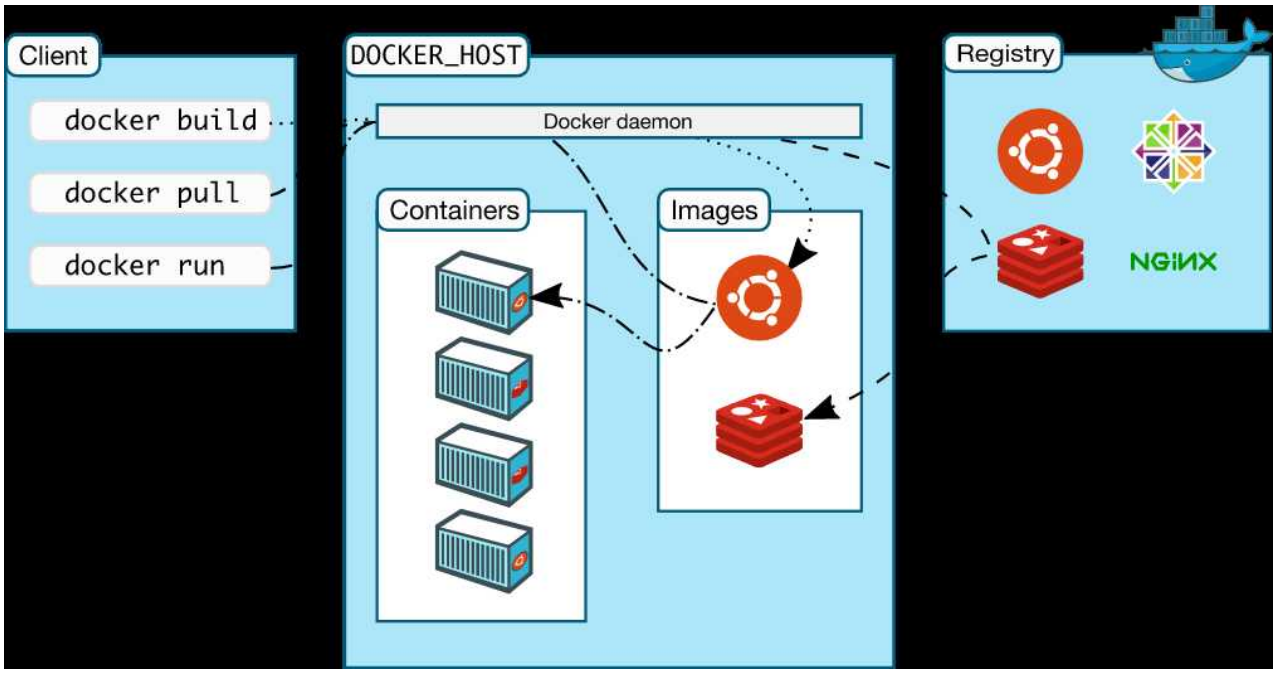
在容器化之后，我们的开发或测试环境都是高度一致的

更高效的计算机资源利用

Docker 是内核级别的虚拟化 ， 可以在物理机上运行很多的容器实例，服务器的性能可以被压榨到极致

Docker的安装：

Docker的基本组成：



镜像(image)：

docker镜像就好比是一个模版，可以通过这个模版来创建一个容器服务，tomcat=====>run(运行起来)=====>tomcat01容器(提供服务)，通过这个镜像可以创建多个容器(最终服务器运行，或者项目运行就是在容器中的)

容器（container）：

Docker利用容器技术，独立运行一个或者一组应用，通过镜像创建的

启动，停止，删除，基本命令！

目前就可以把容器理解为基本的linux系统

仓库（repository）：

存放镜像的地方！

仓库分为公有仓库和私有仓库：

Docker Hub（默认是国外的）

阿里云。。。。都有容器服务（配置镜像加速）

安装Docker：

> 环境准备

1. 需要会一点点linux基础
2. Centos
3. 使用xshell连接远程服务器操作(我用的mac终端)

> 环境查看

```
# 系统内核是3.10以上的
[root@localhost ~]# uname -r
4.18.0-240.el8.x86_64
```

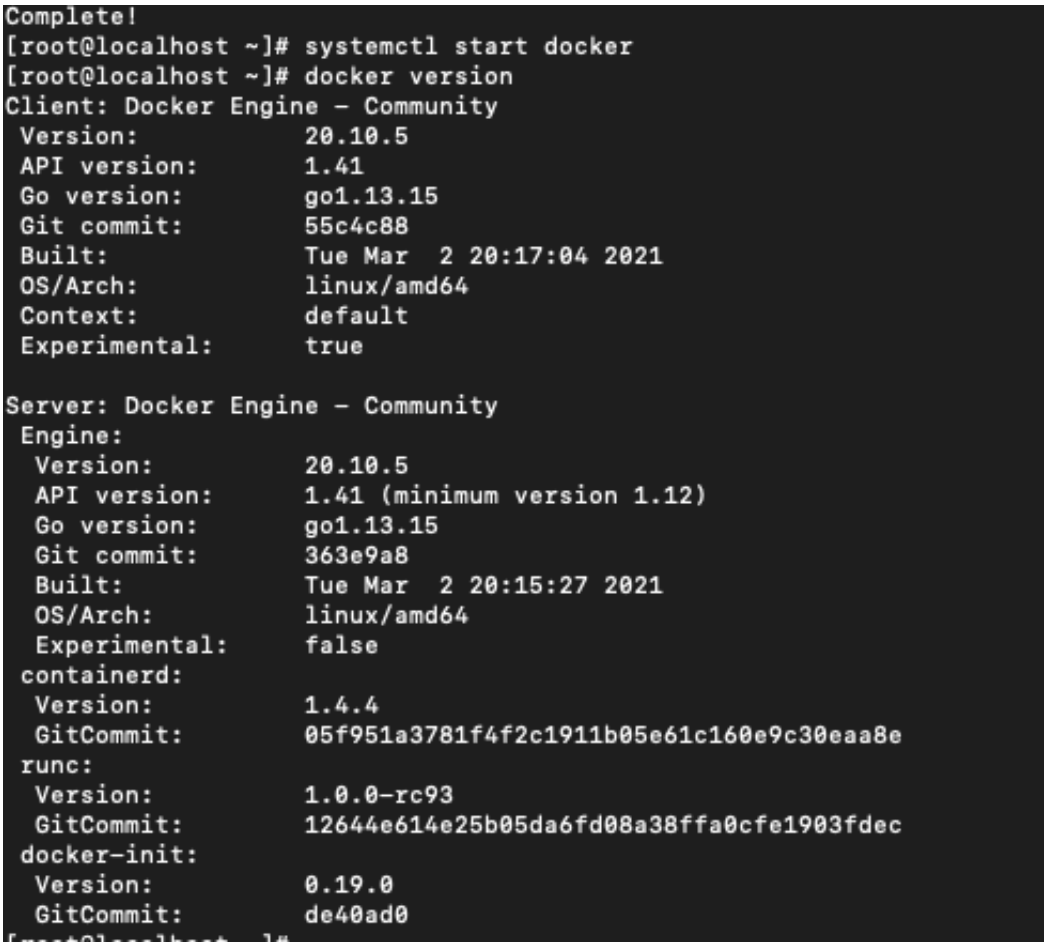
```
[root@localhost ~]# uname -r
4.18.0-240.el8.x86_64
[root@localhost ~]# cat /etc/os-release
NAME="CentOS Linux"
VERSION="8"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="8"
PLATFORM_ID="platform:el8"
PRETTY_NAME="CentOS Linux 8"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:8"
HOME_URL="https://centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"
CENTOS_MANTISBT_PROJECT="CentOS-8"
CENTOS_MANTISBT_PROJECT_VERSION="8"
```

> 安装:

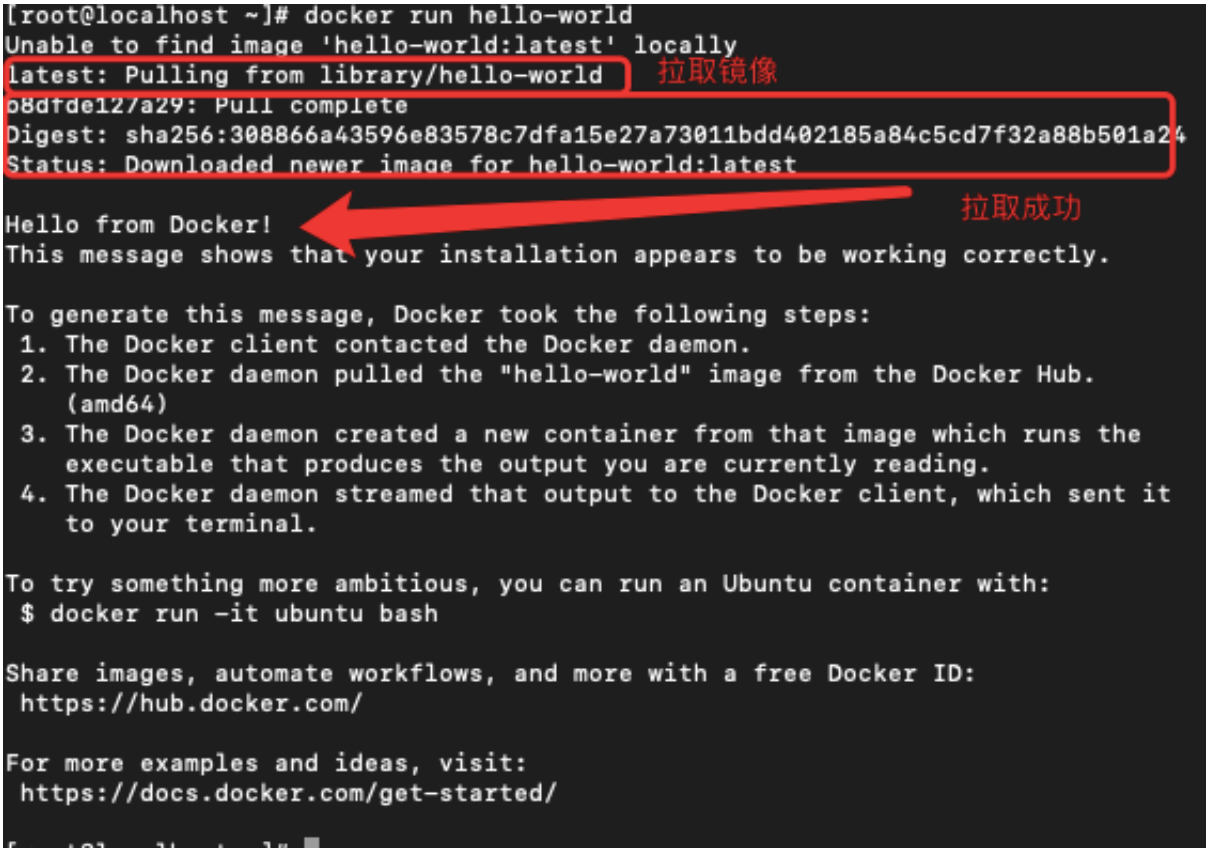
帮助文档:

```
# 1.卸载旧的版本
 sudo yum remove docker \
                  docker-client \
                  docker-client-latest \
                  docker-common \
                  docker-latest \
                  docker-latest-logrotate \
                  docker-logrotate \
                  docker-engine

# 2.需要的安装包
yum install -y yum-utils

# 3.设置镜像的仓库
yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo #默认是从国外的!
```

```
yum-config-manager \
    --add-repo \
    http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo #推荐使用阿里云

# 更新yum软件包索引:
[root@localhost ~]# yum makecache
# 4.安装Docker相关的内容 docker-ce  社区   ee  企业版
yum install docker-ce docker-ce-cli containerd.io

# 5.启动Docker
systemctl start docker

# 6.使用docker version查看是否安装成功
```




```
# 7.使用hello world
docker run hello-world
```



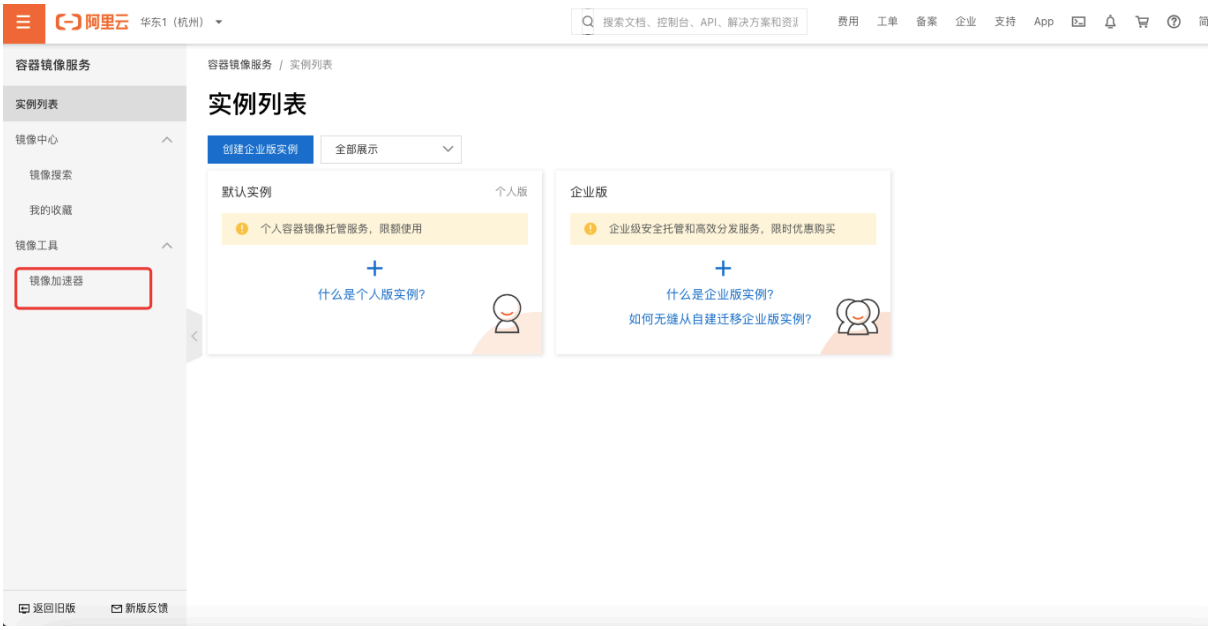

```
# 8.查看下载的hello-world镜像
[root@localhost ~]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    d1165f221234   4 weeks ago    13.3kB
```

了解：卸载docker

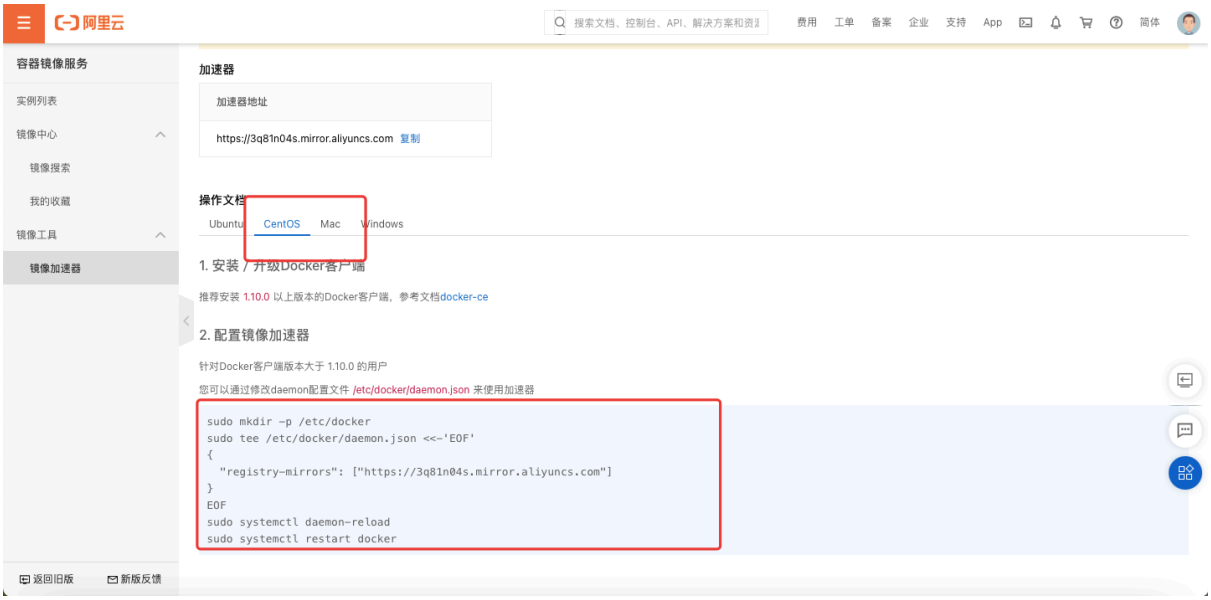
```
# 1.卸载依赖
yum remove docker-ce docker-ce-cli containerd.io

# 2.删除资源
rm -rf /var/lib/docker
rm -rf /var/lib/containerd
# /var/lib/docker docker的默认工作路径
```

阿里云镜像加速

1. 登陆阿里云找到容器服务



2. 找到镜像加速地址



3. 配置使用

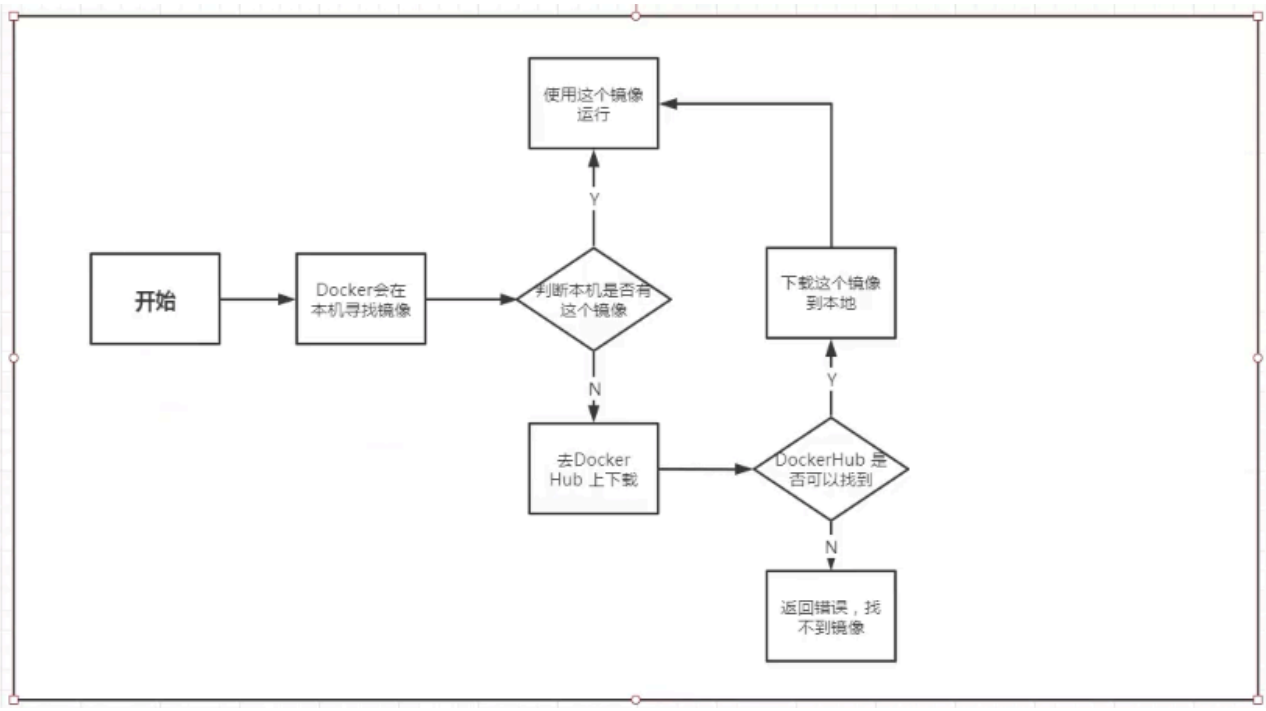
```
sudo mkdir -p /etc/docker

sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://3q81n04s.mirror.aliyuncs.com"]
}
EOF

sudo systemctl daemon-reload

sudo systemctl restart docker
```

回顾HelloWorld流程

```
[root@localhost ~]# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:308866a43596e83578c7dfa15e27a73011bdd402185a84c5cd7f32a88b501a24
Status: Downloaded newer image for hello-world:latest

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

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

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

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

拉取镜像

拉取成功

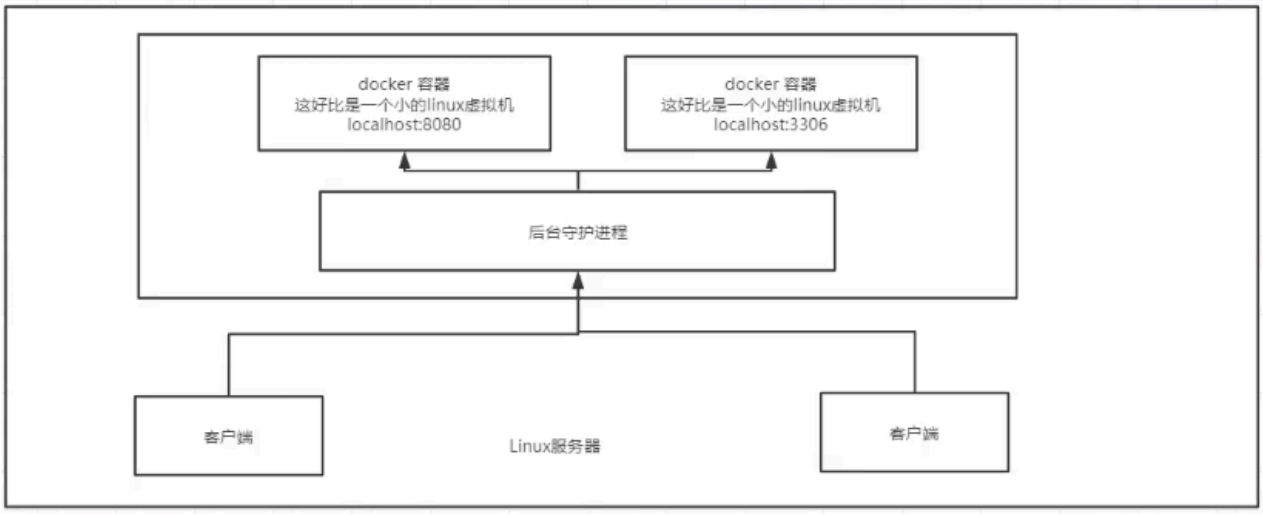


底层原理：

Docker是怎么工作的?

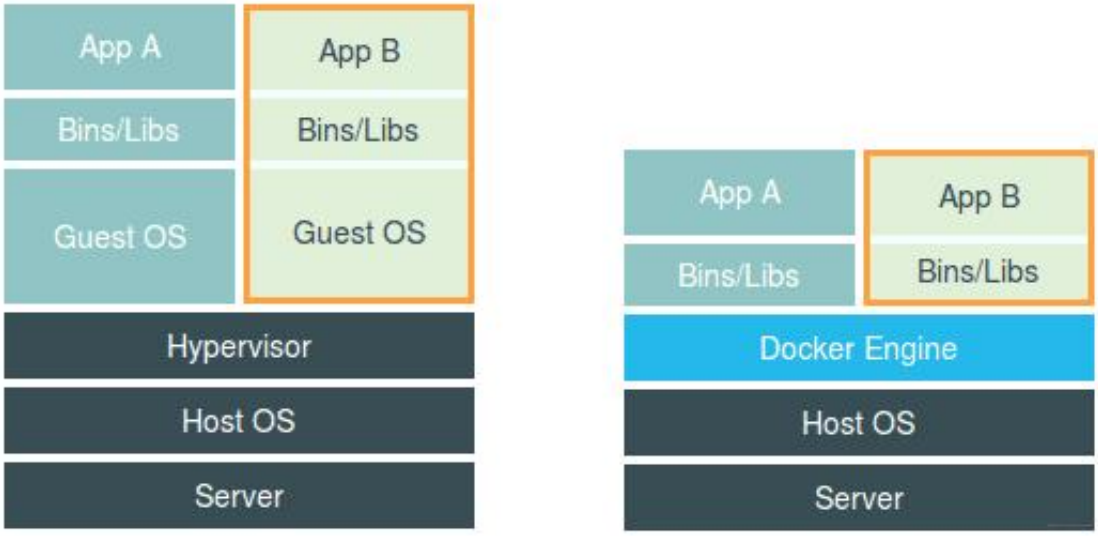
Docker是一个Client-Server结构的系统，Docker的守护进程运行在主机上，通过Socket从客户端访问！

Docker-server接收到Docker-Client的指令，就会执行！



Docker为什么比vm快

1. Docker有着比虚拟机更少的网络进程
2. Docker利用的是宿主机的内核，vm需要的是GuestOS；



所以说，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，避免引导型操作，虚拟机加载GuestOS，分钟级别的 ，Docker是利用宿主机的操作系统，省略了复杂的过程，秒级！

	虚拟机	docker容器
操作系统	宿主机上运行虚拟机OS	共享宿主机OS
存储	镜像较大（GB）	镜像小（MB）
性能	操作系统额外的cpu、内存消耗	几乎无性能损耗
移植性	笨重、与虚拟化技术耦合度高	轻量、灵活迁移
隔离性	完全隔离	安全隔离
部署	慢、分钟级	快速、秒级
运行密度	一般几十个	单机支持上千容器

	Docker容器	LXC	VM
虚拟化类型	OS虚拟化	OS虚拟化	硬件虚拟化
性能	=物理机性能	=物理机性能	5%-20%损耗
隔离性	NS 隔离	NS 隔离	强
QoS	Cgroup 弱	Cgroup 弱	强
安全性	中	差	强
GuestOS	只支持Linux	只支持Linux	全部
可迁移性	强	弱	

Docker常用命令

帮助命令：

```
docker version          # 显示docker的版本信息
docker info             # 显示docker的系统信息，包括镜像和容器的数量
docker 命令 --help       # 万能命令帮助
```

帮助文档地址：https://docs.docker.com/reference/

镜像命令：

docker images 查看本地所有镜像

```
[root@localhost ~]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED       SIZE
hello-world   latest    d1165f221234   4 weeks ago   13.3kB

#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
Options:
  -a, --all             列出所有镜像
  -q, --quiet           只显示镜像的ID
```

docker search　　搜索镜像

```
[root@localhost ~]# docker search mysql
NAME                              DESCRIPTION
  STARS     OFFICIAL   AUTOMATED
mysql                             MySQL is a widely used, open-source relation…
  10702     [OK]
mariadb                           MariaDB Server is a high performing open sou…
  4023      [OK]

#可选项，通过收藏过滤
--filter=STARS=3000   #搜索出来的镜像就是STARS大于3000的

[root@localhost ~]# docker search mysql --filter=STARS=3000
NAME      DESCRIPTION                                     STARS     OFFICIAL
AUTOMATED
mysql     MySQL is a widely used, open-source relation…   10702     [OK]
mariadb   MariaDB Server is a high performing open sou…   4023      [OK]
[root@localhost ~]# docker search mysql --filter=STARS=5000
NAME      DESCRIPTION                                     STARS     OFFICIAL
AUTOMATED
mysql     MySQL is a widely used, open-source relation…   10702     [OK]
```

docker pull 下载镜像

```
#下载镜像  docker pull 镜像名[:tag版本]
[root@localhost ~]# docker pull mysql
Using default tag: latest                       #如果不写tag 默认就是latest
latest: Pulling from library/mysql
a076a628af6f: Pull complete                     #分层下载，docker image的核心  联合文
件系统
f6c208f3f991: Pull complete
88a9455a9165: Pull complete
406c9b8427c6: Pull complete
7c88599c0b25: Pull complete
25b5c6debdaf: Pull complete
43a5816f1617: Pull complete
1a8c919e89bf: Pull complete
9f3cf4bd1a07: Pull complete
80539cea118d: Pull complete
201b3cad54ce: Pull complete
944ba37e1c06: Pull complete
Digest: sha256:feada149cb8ff54eade1336da7c1d080c4a1c7ed82b5e320efb5beebed85ae8c
#签名(防伪)
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest    #真实地址
[root@localhost ~]#

#等价
docker pull mysql
```

```
docker pull docker.io/library/mysql:latest

#指定版本下载
[root@localhost ~]# docker pull mysql:5.7
5.7: Pulling from library/mysql
a076a628af6f: Already exists
f6c208f3f991: Already exists
88a9455a9165: Already exists
406c9b8427c6: Already exists
7c88599c0b25: Already exists
25b5c6debdaf: Already exists
43a5816f1617: Already exists
1831ac1245f4: Pull complete
37677b8c1f79: Pull complete
27e4ac3b0f6e: Pull complete
7227baa8c445: Pull complete
Digest: sha256:b3d1eff023f698cd433695c9506171f0d08a8f92a0c8063c1a4d9db9a55808df
Status: Downloaded newer image for mysql:5.7
docker.io/library/mysql:5.7
```

```
[[root@localhost ~]# docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    d1165f221234   4 weeks ago    13.3kB
mysql         5.7       a70d36bc331a   2 months ago   449MB
mysql         latest    c8562eaf9d81   2 months ago   546MB
```

docker rmi 删除镜像!

```
[root@localhost ~]# docker rmi -f a70d36bc331a          #删除指定的容器

[root@localhost ~]# docker rmi -f 镜像id 镜像id 镜像id    #删除多个容器

[root@localhost ~]# docker rmi -f $(docker images -aq) #删除全部的容器
```

容器命令:

说明：有了镜像才可以创建容器，linux，下载一个centos镜像来测试学习

```
docker pull centos
```

新建容器并启动

```
docker run [可选参数] images

#参数说明
--name=="Name"       容器名字    tomcat01    tomacat02，用来区分容器
-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                  指定容器的端口   -p          8080:8080
#      -p       ip:主机端口:容器端口 （常用）
#      -p       主机端口:容器端口 （常用）
#      -p       容器端口
-P                  随机指定端口


#测试，启动并进入容器
[root@localhost ~]# docker run -it centos /bin/bash
[root@80138667de4f /]#

[root@80138667de4f /]# ls   查看容器内的centos，基础版本，很多命令都是不完善的
bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run
sbin  srv  sys  tmp  usr  var

#从容器中退回到主机
[root@80138667de4f /]# exit
exit
[root@localhost ~]# ls
anaconda-ks.cfg  Desktop  Documents  Downloads  initial-setup-ks.cfg  Music
Pictures  Public  Templates  Videos
```

列出当前正在运行的容器

```
#docker ps 命令
      #显示正在运行的容器
-a    # 列出当前正在运行的容器+带出历史运行过的容器
-n=?  #显示最近创作的容器
-q    # 只显示容器的编号

[root@localhost ~]# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
[root@localhost ~]# docker ps -a
CONTAINER ID   IMAGE          COMMAND       CREATED          STATUS
        PORTS     NAMES
80138667de4f   centos         "/bin/bash"   11 minutes ago   Exited (0) 6
minutes ago             loving_lederberg
ddfd509b66f4   d1165f221234   "/hello"      14 hours ago     Exited (0) 14
hours ago               romantic_shirley
```

退出容器

```
exit    #直接容器停止并退出
ctrl + P + Q  #容器不停止退出
```

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
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
```

```
[root@localhost ~]# docker ps -a
CONTAINER ID   IMAGE          COMMAND       CREATED          STATUS                      PORTS     NAMES
7e38df502dfc   centos         "/bin/bash"   9 minutes ago    Up 9 minutes                          zealous_swanson
80138667de4f   centos         "/bin/bash"   29 minutes ago   Exited (0) 24 minutes ago             loving_lederberg
ddfd509b66f4   d1165f221234   "/hello"      14 hours ago     Exited (0) 14 hours ago               romantic_shirley
[root@localhost ~]# docker rm 80138667de4f
80138667de4f
[root@localhost ~]# docker rm 7e38df502dfc
Error response from daemon: You cannot remove a running container 7e38df502dfcec4bdad1952a66be10634ed67864aab193eb37e3f86f90e32d1. Stop the container before attempting removal or force remove
[root@localhost ~]#  docker ps -aq
7e38df502dfc
ddfd509b66f4
[root@localhost ~]#  docker rm -f $(docker ps -aq)
7e38df502dfc
ddfd509b66f4
[root@localhost ~]# docker ps -1
unknown shorthand flag: '1' in -1
See 'docker ps --help'.
[root@localhost ~]# docker ps -q
[root@localhost ~]#
```


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           #停止当前正在运行的容器

docker kill 容器id           #强制停止当前容器
```

```
[root@localhost ~]# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
[root@localhost ~]# docker ps -a
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS                       PORTS     NAMES
258776d8db41   centos    "/bin/bash"   35 seconds ago   Exited (127) 24 seconds ago            festive_montalcini
3968f6815574   centos    "/bin.bash"   49 seconds ago   Created                                interesting_pike
[root@localhost ~]#  docker start 258776d8db41
258776d8db41
[root@localhost ~]# docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED              STATUS         PORTS     NAMES
258776d8db41   centos    "/bin/bash"   About a minute ago   Up 5 seconds             festive_montalcini
[root@localhost ~]# docke stop
bash: docke: command not found...
[root@localhost ~]# ^C
[root@localhost ~]# docker stop 258776d8db41
258776d8db41
[root@localhost ~]# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

常用的其他命令:

后台启动容器

```
# 命令 docker run -d 镜像名
[root@localhost ~]# docker run -d centos

# docker ps 发现 centos停止了

# 常见的坑，docker容器使用后台运行就必须要有一个前台进程
# nginx，容器启动后，发现自己没有提供服务，就会立刻停止
```

查看日志命令

```
docker logs --help

docker logs -f -t -n 容器id      没有日志

# 自己编写一段shell脚本
“while true; do echo LGQ ;sleep 2;done”

[root@localhost ~]#  docker run -d centos /bin/sh -c "while true; do echo
LGQ;sleep 1;done"
d303f1869bfb684c0d083ea61a56e00680b0787be5bcd2cb38af24c34f6c2ba2

docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
   PORTS     NAMES
d303f1869bfb   centos    "/bin/sh -c 'while t…"   10 seconds ago   Up 4 seconds
             goofy_shannon
2338d4c660bb   centos    "/bin/bash"              12 minutes ago   Up 12
minutes             gifted_merkle

# 显示日志
-tf
-n number   #要显示的日志条数
[root@localhost ~]# docker logs -tf -n 10 d303f1869bfb
```

查看容器中的进程信息

```
# 命令 docker top 容器id
[root@localhost ~]# docker top d303f1869bfb
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                6447                6426                0
01:54               ?                   00:00:00            /bin/sh -c while
true; do echo LGQ;sleep 1;done
root                6985                6447                0
02:01               ?                   00:00:00            /usr/bin/coreutils
--coreutils-prog-shebang=sleep /usr/bin/sleep 1
```

查看镜像元数据

```
# 命令
docker inspect 容器id

#测试
[root@localhost ~]# docker inspect d303f1869bfb
[
    {
        "Id":
"d303f1869bfb684c0d083ea61a56e00680b0787be5bcd2cb38af24c34f6c2ba2",
        "Created": "2021-04-05T05:54:41.902111543Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "while true; do echo LGQ;sleep 1;done"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 6447,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2021-04-05T05:54:46.823317086Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image":
"sha256:300e315adb2f96afe5f0b2780b87f28ae95231fe3bdd1e16b9ba606307728f55",
        "ResolvConfPath":
"/var/lib/docker/containers/d303f1869bfb684c0d083ea61a56e00680b0787be5bcd2cb38a
f24c34f6c2ba2/resolv.conf",
        "HostnamePath":
"/var/lib/docker/containers/d303f1869bfb684c0d083ea61a56e00680b0787be5bcd2cb38a
f24c34f6c2ba2/hostname",
        "HostsPath":
"/var/lib/docker/containers/d303f1869bfb684c0d083ea61a56e00680b0787be5bcd2cb38a
f24c34f6c2ba2/hosts",
        "LogPath":
"/var/lib/docker/containers/d303f1869bfb684c0d083ea61a56e00680b0787be5bcd2cb38a
f24c34f6c2ba2/d303f1869bfb684c0d083ea61a56e00680b0787be5bcd2cb38af24c34f6c2ba2-
json.log",
        "Name": "/goofy_shannon",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
```

```
        "MountLabel": "",
        "ProcessLabel": "",
        "AppArmorProfile": "",
        "ExecIDs": null,
        "HostConfig": {
            "Binds": null,
            "ContainerIDFile": "",
            "LogConfig": {
                "Type": "json-file",
                "Config": {}
            },
            "NetworkMode": "default",
            "PortBindings": {},
            "RestartPolicy": {
                "Name": "no",
                "MaximumRetryCount": 0
            },
            "AutoRemove": false,
            "VolumeDriver": "",
            "VolumesFrom": null,
            "CapAdd": null,
            "CapDrop": null,
            "CgroupnsMode": "host",
            "Dns": [],
            "DnsOptions": [],
            "DnsSearch": [],
            "ExtraHosts": null,
            "GroupAdd": null,
            "IpcMode": "private",
            "Cgroup": "",
            "Links": null,
            "OomScoreAdj": 0,
            "PidMode": "",
            "Privileged": false,
            "PublishAllPorts": false,
            "ReadonlyRootfs": false,
            "SecurityOpt": null,
            "UTSMode": "",
            "UsernsMode": "",
            "ShmSize": 67108864,
            "Runtime": "runc",
            "ConsoleSize": [
                0,
                0
            ],
            "Isolation": "",
            "CpuShares": 0,
            "Memory": 0,
            "NanoCpus": 0,
```

```
            "CgroupParent": "",
            "BlkioWeight": 0,
            "BlkioWeightDevice": [],
            "BlkioDeviceReadBps": null,
            "BlkioDeviceWriteBps": null,
            "BlkioDeviceReadIOps": null,
            "BlkioDeviceWriteIOps": null,
            "CpuPeriod": 0,
            "CpuQuota": 0,
            "CpuRealtimePeriod": 0,
            "CpuRealtimeRuntime": 0,
            "CpusetCpus": "",
            "CpusetMems": "",
            "Devices": [],
            "DeviceCgroupRules": null,
            "DeviceRequests": null,
            "KernelMemory": 0,
            "KernelMemoryTCP": 0,
            "MemoryReservation": 0,
            "MemorySwap": 0,
            "MemorySwappiness": null,
            "OomKillDisable": false,
            "PidsLimit": null,
            "Ulimits": null,
            "CpuCount": 0,
            "CpuPercent": 0,
            "IOMaximumIOps": 0,
            "IOMaximumBandwidth": 0,
            "MaskedPaths": [
                "/proc/asound",
                "/proc/acpi",
                "/proc/kcore",
                "/proc/keys",
                "/proc/latency_stats",
                "/proc/timer_list",
                "/proc/timer_stats",
                "/proc/sched_debug",
                "/proc/scsi",
                "/sys/firmware"
            ],
            "ReadonlyPaths": [
                "/proc/bus",
                "/proc/fs",
                "/proc/irq",
                "/proc/sys",
                "/proc/sysrq-trigger"
            ]
        },
        "GraphDriver": {
```

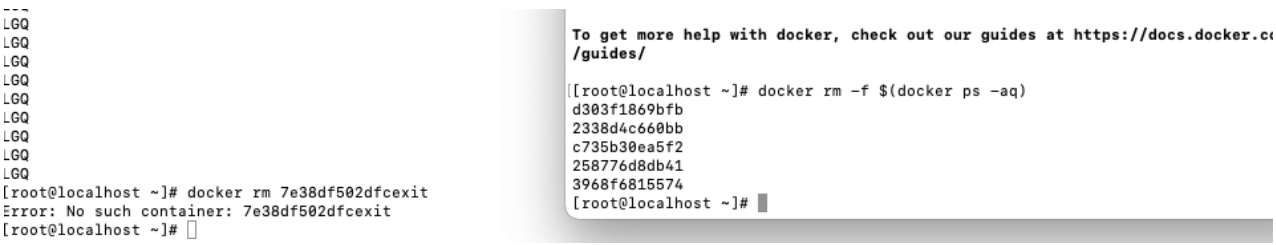
```
            "Data": {
                "LowerDir":
"/var/lib/docker/overlay2/7269492f665bf6b4e0009b7c1fad5aa941adf82c15efcee735da5
ec7d4d8391b-
init/diff:/var/lib/docker/overlay2/afe85b4c582c491304665a5b541b2ca1ace47a72febf
51a3c8abacf5b98c73d5/diff",
                "MergedDir":
"/var/lib/docker/overlay2/7269492f665bf6b4e0009b7c1fad5aa941adf82c15efcee735da5
ec7d4d8391b/merged",
                "UpperDir":
"/var/lib/docker/overlay2/7269492f665bf6b4e0009b7c1fad5aa941adf82c15efcee735da5
ec7d4d8391b/diff",
                "WorkDir":
"/var/lib/docker/overlay2/7269492f665bf6b4e0009b7c1fad5aa941adf82c15efcee735da5
ec7d4d8391b/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Hostname": "d303f1869bfb",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [

 "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "while true; do echo LGQ;sleep 1;done"
            ],
            "Image": "centos",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "org.label-schema.build-date": "20201204",
                "org.label-schema.license": "GPLv2",
                "org.label-schema.name": "CentOS Base Image",
                "org.label-schema.schema-version": "1.0",
                "org.label-schema.vendor": "CentOS"
```

```
            }
        },
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID":
"e37b7f8efaa36b77cef7130873ab76566e49ef10d66ed96cb65125c348f473cd",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {},
            "SandboxKey": "/var/run/docker/netns/e37b7f8efaa3",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID":
"ba4951dc04d4b9eec518a3b1ed4be18e142693178517fed9d342185c2073ddef",
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.3",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:03",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID":
"995c319e9566204b60640b4d8d3af4e208ed114072446aea47e77ebd78e1640a",
                    "EndpointID":
"ba4951dc04d4b9eec518a3b1ed4be18e142693178517fed9d342185c2073ddef",
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.3",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:03",
                    "DriverOpts": null
                }
            }
        }
    }
]
```


进入当前正在运行的容器

```
# 我们通常容器id都是使用后台方式运行的，需要进入其中的容器，修改一些配置
```

```
#命令
docker exec  -it 容器id /bin/bash

#测试
[root@localhost ~]#  docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
   PORTS     NAMES
d303f1869bfb   centos    "/bin/sh -c 'while t…"   27 minutes ago   Up 27
minutes             goofy_shannon
2338d4c660bb   centos    "/bin/bash"              40 minutes ago   Up 40
minutes             gifted_merkle
[root@localhost ~]# docker exec -it d303f1869bfb /bin/bash
[root@d303f1869bfb /]# ls
bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run
sbin  srv  sys  tmp  usr  var
[root@d303f1869bfb /]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 05:54 ?        00:00:01 /bin/sh -c while true; do
echo LGQ;sleep 1;done
root        1701       0  1 06:23 pts/0    00:00:00 /bin/bash
root        1733       1  0 06:23 ?        00:00:00 /usr/bin/coreutils --
coreutils-prog-shebang=sleep /usr/bin/sleep 1
root        1734    1701  0 06:23 pts/0    00:00:00 ps -ef


# 方式二
docker attach 容器id

#测试
[root@localhost ~]# docker attach d303f1869bfb
正在执行当前的代码==>


#docker exec        #进入容器后开启一个新的终端，可以在里面操作
#dockeer attach     # 进入容器正在执行的终端，不会执行新的进程!
```



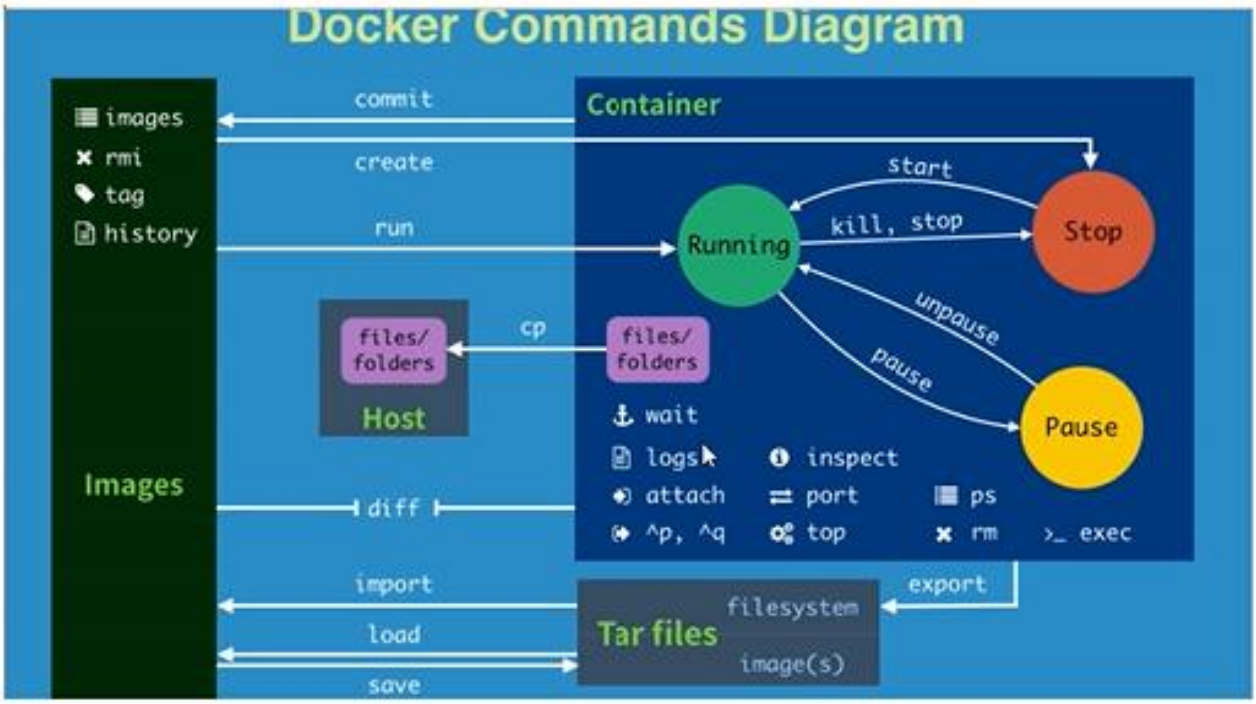
将容器内文件拷贝到主机上

```
docker cp 容器id:容器内路径   目的主机路径

#查看当前主机目录下
```

```
[root@localhost home]# ls
echo.java  echo_machine
[root@localhost home]# docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS          PORTS
NAMES
43726e8f2f2e   centos    "/bin/bash"   2 minutes ago    Up 2 minutes
laughing_ishizaka
# 进入docker容器内部
[root@localhost home]#  docker attach 43726e8f2f2e
[root@43726e8f2f2e /]# cd /home
[root@43726e8f2f2e home]# ls
# 在容器内新建一个文件
[root@43726e8f2f2e home]# touch test.java
[root@43726e8f2f2e home]# ls
test.java
[root@43726e8f2f2e home]# exit
exit
[root@localhost home]# docker ps
CONTAINER d   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
[root@localhost home]# docker ps -a
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS
   PORTS     NAMES
43726e8f2f2e   centos    "/bin/bash"   3 minutes ago    Exited (0) 10 seconds
ago             laughing_ishizaka
54c0838be883   centos    "/bin/bash"   4 minutes ago    Exited (0) 4 minutes ago
             vigilant_elgamal

#将文件拷贝出来到主机是上
[root@localhost home]# docker cp 43726e8f2f2e:/home/test.java /home
[root@localhost home]# ls
echo.java  echo_machine  test.java

# 拷贝是一个手动的过程 未来我们可以使用一个 -v 卷的技术，可以实现，自动同步
```

学习方式:将所有命令全部敲一遍，自己记录笔记!

小结:




```
attach    Attach to a running container                 # 当前 shell 下 attach 连接指定运行镜像
build     Build an image from a Dockerfile              # 通过 Dockerfile 定制镜像
commit    Create a new image from a container changes   # 提交当前容器为新的镜像
cp        Copy files/folders from the containers filesystem to the host path   #从容器中拷贝指定文件或者目录到宿主机中
create    Create a new container                        # 创建一个新的容器，同 run，但不启动容器
diff      Inspect changes on a container's filesystem   # 查看 docker 容器变化
events    Get real time events from the server          # 从 docker 服务获取容器实时事件
exec      Run a command in an existing container        # 在已存在的容器上运行命令
export    Stream the contents of a container as a tar archive   # 导出容器的内容流作为一个 tar 归档文件[对应 import ]
history   Show the history of an image                  # 展示一个镜像形成历史
images    List images                                   # 列出系统当前镜像
import    Create a new filesystem image from the contents of a tarball # 从tar包中的内容创建一个新的文件系统映像[对应export]
info      Display system-wide information               # 显示系统相关信息
inspect   Return low-level information on a container   # 查看容器详细信息
kill      Kill a running container                      # kill 指定 docker 容器
load      Load an image from a tar archive              # 从一个 tar 包中加载一个镜像[对应 save]
login     Register or Login to the docker registry server    # 注册或者登陆一个 docker 源服务器
logout    Log out from a Docker registry server          # 从当前 Docker registry 退出
logs      Fetch the logs of a container                 # 输出当前容器日志信息
port      Lookup the public-facing port which is NAT-ed to PRIVATE_PORT    # 查看映射端口对应的容器内部源端口
pause     Pause all processes within a container        # 暂停容器
ps        List containers                               # 列出容器列表
pull      Pull an image or a repository from the docker registry server   # 从docker镜像源服务器拉取指定镜像或者库镜像
push      Push an image or a repository to the docker registry server    # 推送指定镜像或者库镜像至docker源服务器
restart   Restart a running container                   # 重启运行的容器
rm        Remove one or more containers                 # 移除一个或者多个容器
rmi       Remove one or more images             # 移除一个或多个镜像[无容器使用该镜像才可删除，否则需删除相关容器才可继续或 -f 强制删除]
```

>

>

docker的命令是十分多的，上面学习的搜是最常用的容器和镜像命令，之后还有很多命令

> 接下来就是练习

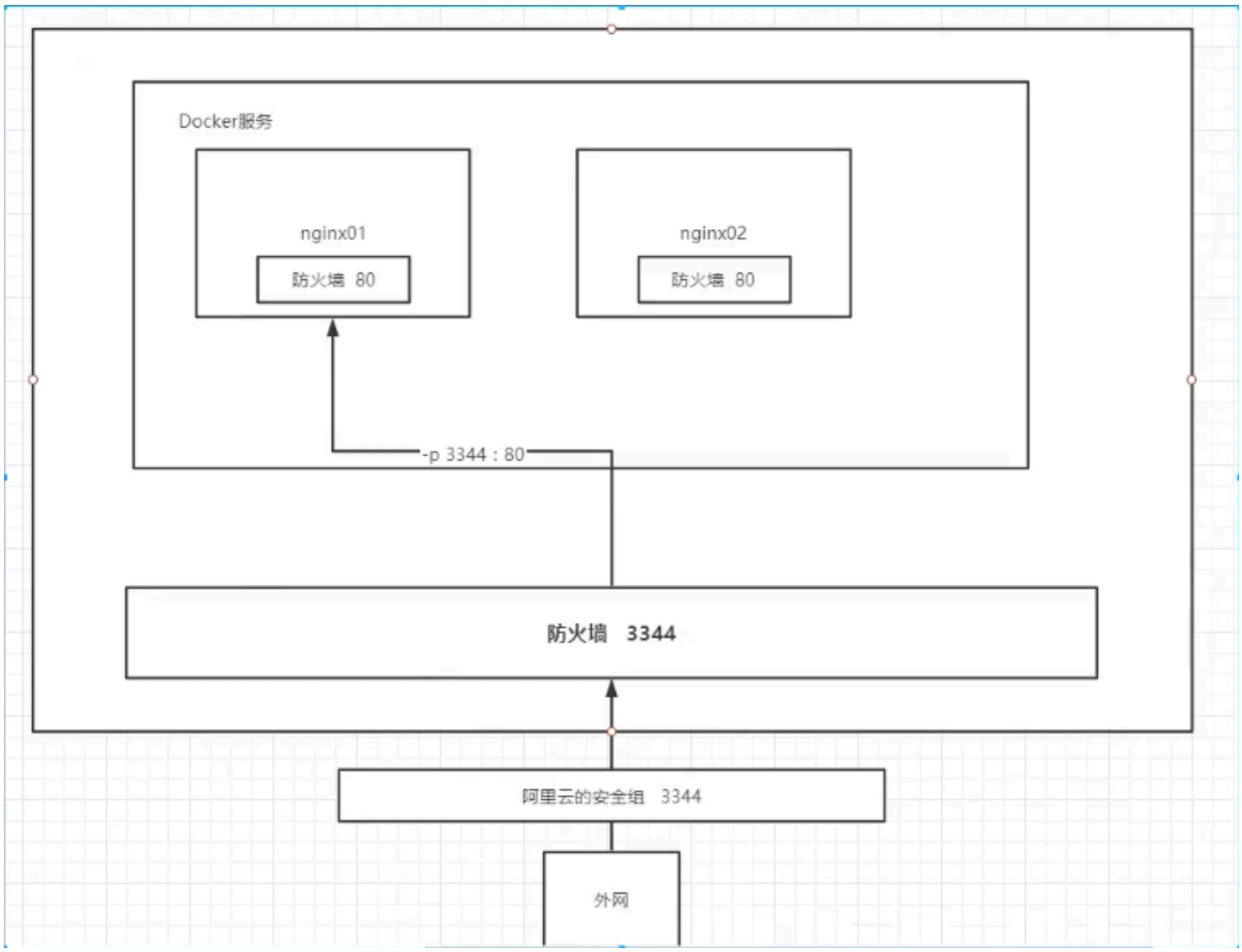
作业练习

> Docker安装Nginx

```
# 1.搜索镜像
  docker search nginx
# 2.下载镜像
  docker pull nginx
# 3.运行测试
[root@localhost home]# docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
nginx        latest    f6d0b4767a6c   2 months ago    133MB
centos       latest    300e315adb2f   3 months ago    209MB

# -d         后台运行
# --name     给容器命名
# -p         宿主机端口:容器内部端口
[root@localhost home]# docker run -d --name nginx01 -p 3344:80 nginx
ded5871cf4f883ea764a85d2b3b23a39994821fff98dfae2df115182ec14e723
o[root@localhost home]# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
  PORTS                  NAMES
ded5871cf4f8   nginx     "/docker-entrypoint.…"   17 seconds ago   Up 5 seconds
  0.0.0.0:3344->80/tcp   nginx01
[root@localhost home]# curl localhost:3344
d
#进入容器
[root@localhost home]# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
   PORTS                   NAMES
ded5871cf4f8   nginx     "/docker-entrypoint.…"   14 minutes ago   Up 14
minutes   0.0.0.0:3344->80/tcp   nginx01

[root@localhost home]# docker exec  -it nginx01 /bin/bash
root@ded5871cf4f8:/# whereis nginx
nginx: /usr/sbin/nginx /usr/lib/nginx /etc/nginx /usr/share/nginx

root@ded5871cf4f8:/# cd /etc/nginx
root@ded5871cf4f8:/etc/nginx# ls
conf.d  fastcgi_params  koi-utf  koi-win  mime.types  modules  nginx.conf
scgi_params  uwsgi_params  win-utf
```

端口暴露的概念:

```
# 容器的镜像相当于一个小的虚拟机，对于外部宿主机是没有关系的，所以必须端口映射才能通信


-p 3333:80

-P
```



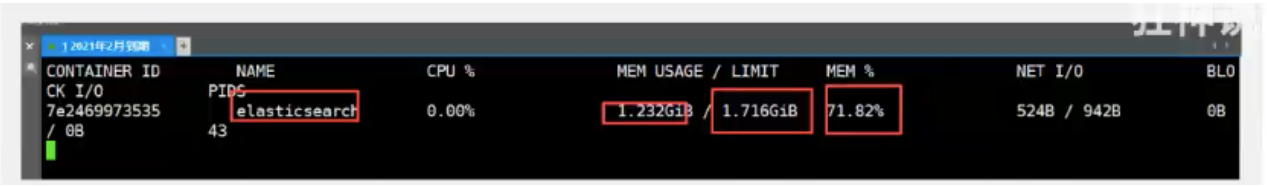
> docker来安装一个tomcat

```
# 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，容器还可以查到 docker run -it --rm tomcat:9.0
一般用来测试，用完即删

# 先下载再启动
docker pull tomcat:9.0

# 启动并运行
[root@localhost home]# docker run -d -p 3355:8080 --name tomcat01 tomcat
# 测试访问没有问题
[root@localhost home]# docker run -d -p 3355:8080 --name tomcat01 tomcat
b332d627414de29d6a8e83f70fd0f4b2dba4bf5513dbaad336c30ef42d03fa1a
# 进入容器
[root@localhost home]# docker exec -it tomcat01 /bin/bash

# 发现问题：1.linux命令少了 2.没有webapps 阿里云镜像的原因，默认是最小的镜像，所以不必要的
都剔除了
```

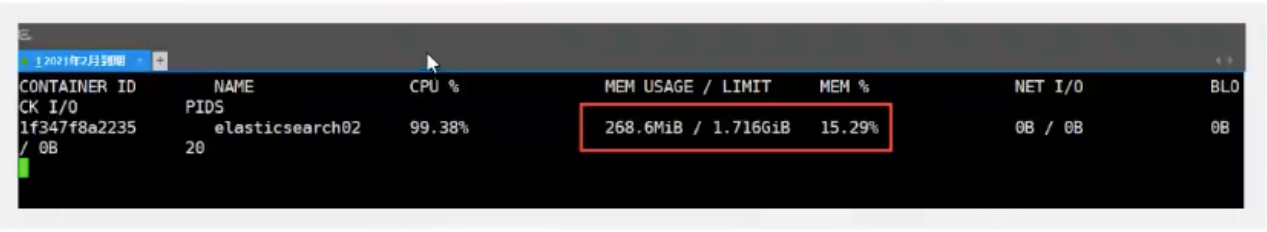
> 部署es+kibana

```
# es 暴露的端口很多
# es 十分消耗内存
# es 的数据一般需要放置到安全目录！挂载

#启动 elasticsearch
```

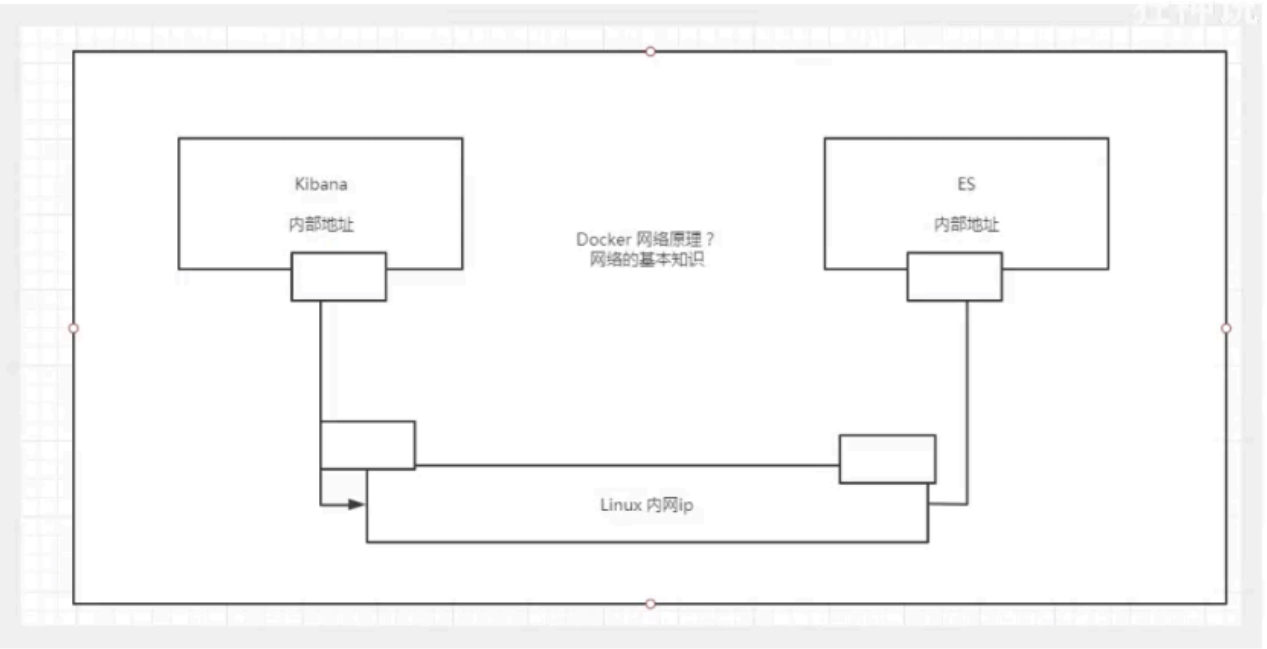
```
docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e
"discovery.type=single-node" elasticsearch:7.6.2

# 启动了，linux就卡住了     docker status查看cpu的状态

# 测试es是否成功

# es 是十分耗内存的

# 查看 docker stats

#赶紧关闭，增加内存限制
```




```
#赶紧关闭，增加内存限制,修改配置文件 -e 环境配置修改
docker run -d --name elasticsearch02 -p 9200:9200 -p "discorvery.type=single-
node" -e ES_JAVA_OPTS="-Xms 64m -Xmx512m" elasticsearch:7.6.2
```



思考：使用kibana连接es？思考如何才能连接过去



可视化

- portainer(先用这个)

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

- Rancher(CI/CD再用)

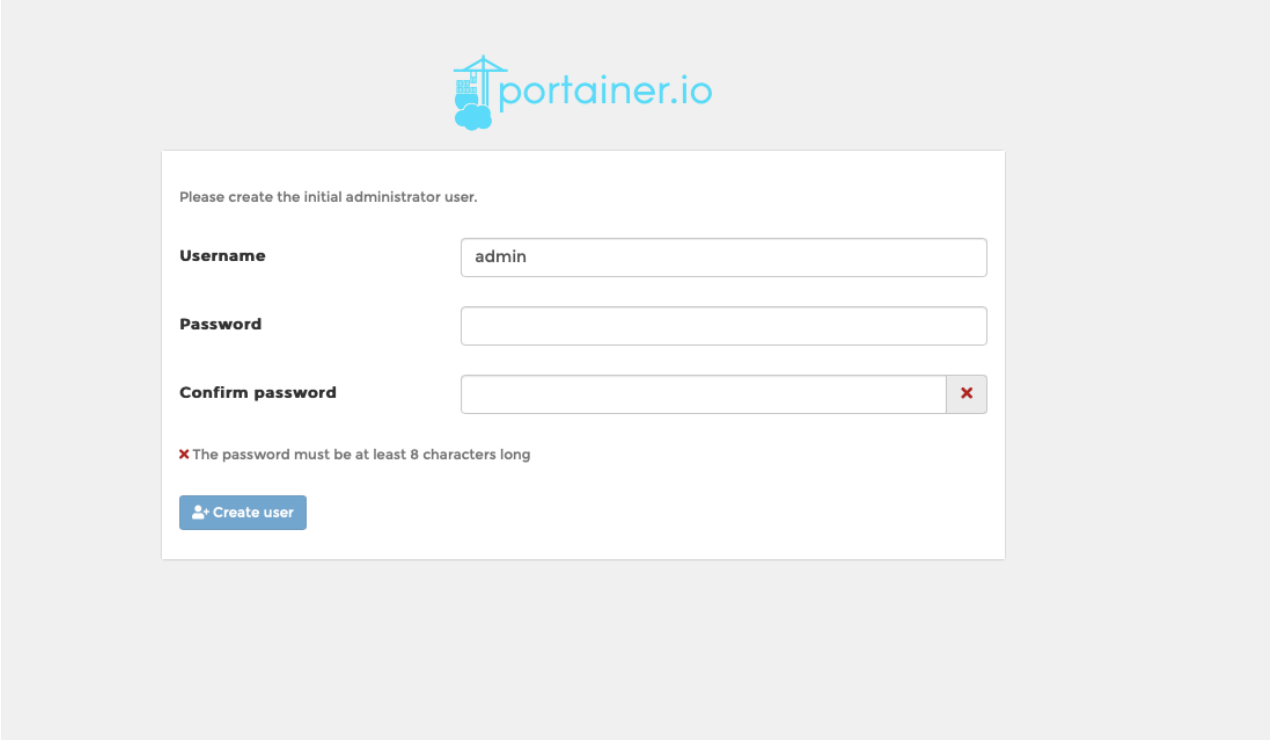
什么是portainer?

Docker　图形化界面管理工具！提供一个后台面板供我们操作

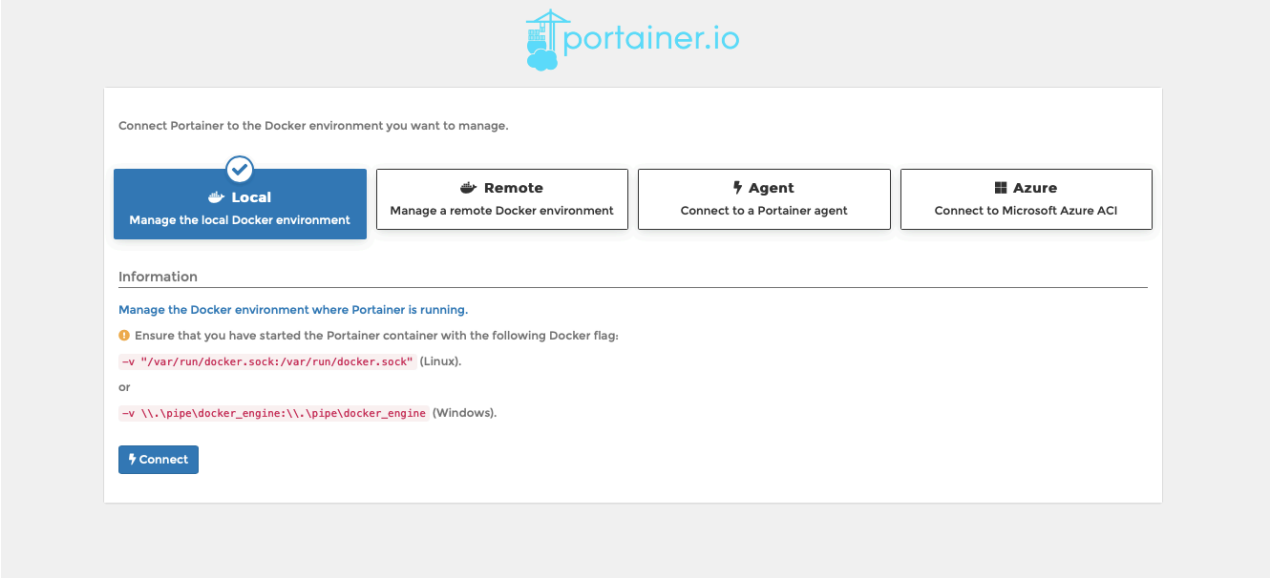
```
docker run -d -p 8088:9000 \
--restart=always -v /var/run/docker.sock:/var/run/docker.sock --privileged=true
portainer/portainer
```

访问测试：外网：8088 http://192.168.220.17:8088/#/init/admin

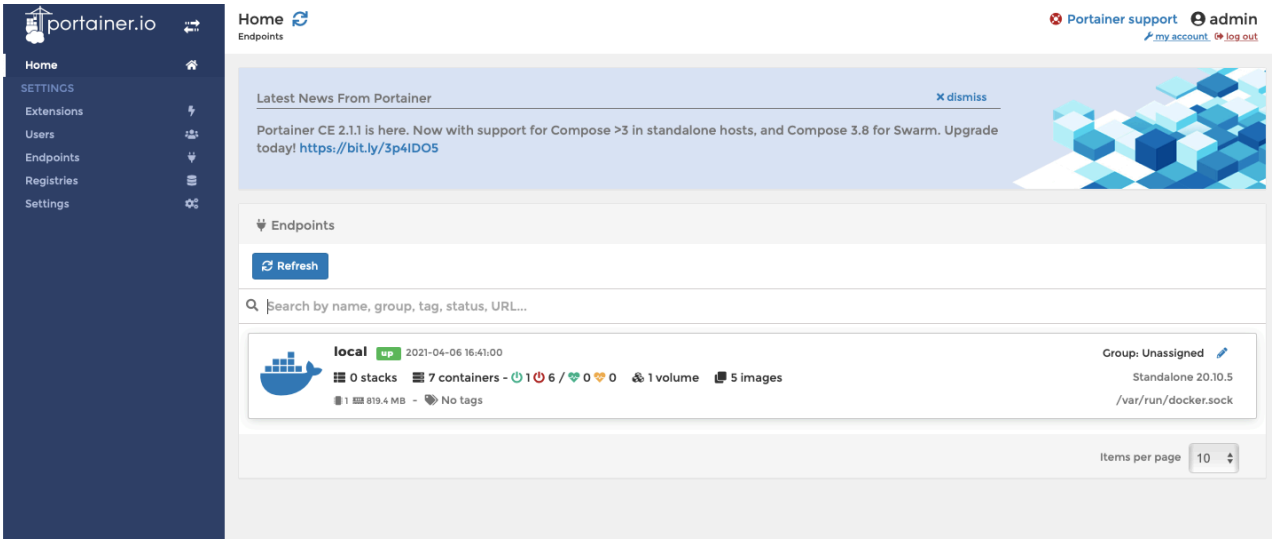
通过它来访问



选择本地的：



进入之后的面板：



可视化面板平时不会用，测试完即可

Docke镜像的讲解

镜像是什么

镜像是一种轻量级可执行的软件包，用来打包软件的运行环境和基于环境开发的软件，包含运行某个软件所需要的所有内容，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，就可以直接运行

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 DockerFile

Docker镜像加速

> UnionFS(联合文件系统)

我们下载的时候看到的一层一层就是这个

UnionFS(联合文件系统)：Union文件系统(UnionFS)是一种分层轻量级并且高性能的文件系统,他支持对文件系统的更改，作为一次提交来一层层的叠加，同时可以将不同的目录挂载到同一个虚拟文件系统下（unite several directories into a single virtual filesystem）。Union文件系统是Docker镜像的基础，镜像可以通过分层来继承，基于基础镜像（没有父镜像），可以制作各种具体的镜像。

特性：一次同时加载多个文件系统，但从外面来看，只能看到一个文件系统联合加载会把各层文件系统叠加起来，这样最终文件系统会包括所有底层的文件和目录

知乎解读：https://zhuanlan.zhihu.com/p/47025759?from_voters_page=true

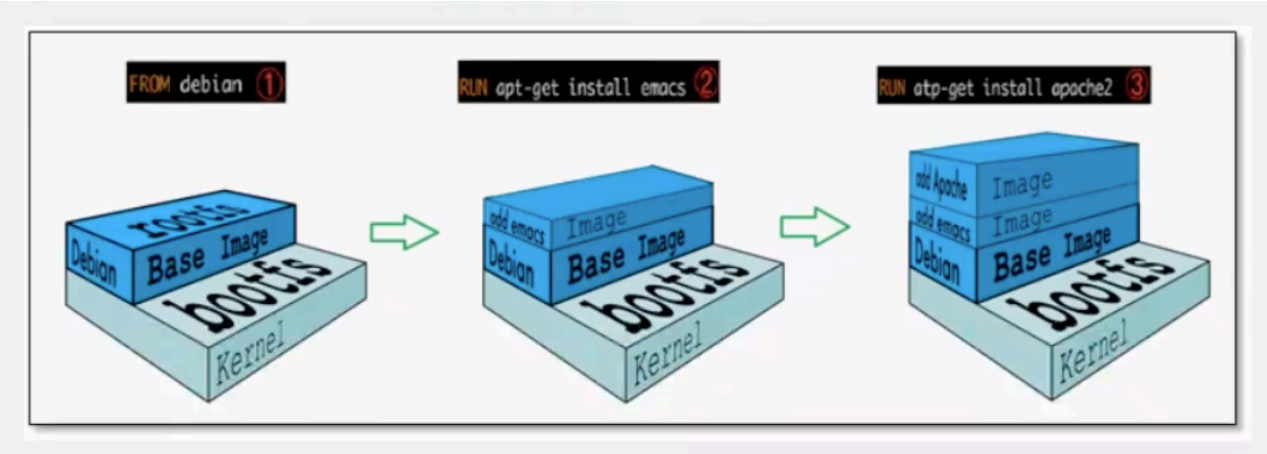
> Docker镜像加载的原理

Docker镜像实际由一层层的文件系统组成，这种层级的文件系统UnionFS。

bootfs(coot dile system)主要包含bootloader和kernel，bootloader主要是引导加载kernel，Linux刚启动时会加载bootfs的文件系统，在docker镜像的最底层时是botfs。这一层与我们典型的Linux/Unix是一样的，包含boot加载器和内核，当boot加载完之后整个内核就在内存中了，此时的内存使用权已由bootfs转交给内核，此时系统也会卸载bootfs。

黑屏---开机进入系统

rootfs(root file system),在bootfs上，包含的就是典型的Linux系统中的/dev，/proc，/bin，/etc等标准目录文件，rootfs就是各种不同的操作系统的发行版，比如ubuntu，Centos等等；



平时我们安装的虚拟机的CentOS都是好几个G，为什么Docker这里才200M?

```
[root@localhost ~]# docker images centos
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
centos       latest    300e315adb2f   3 months ago    209MB
[root@localhost ~]#
```

对于一个精简的OS，rootfs可以小，只需要包含最基本的命令，工具和程序库就可以了，因为底层直接用host的kernel，自己只需要提供一个rootfs酷可以了，由此可见对于不同的linux发行版bootfs‘基本上是一致的，rootfs‘会有差别，因此不同的发行版可以公用bootfs

虚拟机是分钟级别的，容器是秒级的

分层理解

> 分层的镜像

我们可以下载一个镜像，注意观察下载的日志输出，可以看到是一层一层在下载！

```
[root@localhost ~]# docker pull redis
Using default tag: latest
latest: Pulling from library/redis
a076a628af6f: Already exists
f40dd07fe7be: Pull complete
ce21c8a3dbee: Pull complete
ee99c35818f8: Pull complete
56b9a72e68ff: Pull complete
3f703e7f380f: Pull complete
Digest: sha256:0f97c1c9daf5b69b93390ccbe8d3e2971617ec4801fd0882c72bf7cad3a13494
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
```

思考：为什么docker镜像要采用这种分层结构呢?

最大的好处就是资源共享，比如有多个镜像都从想通的base镜像构建而来，那么宿主机只需在磁盘上保留一份base镜像，同时内存中也只需要加载一份镜像，这样就可以为所有的容器服务了，而且镜像的每一层都可以共享。

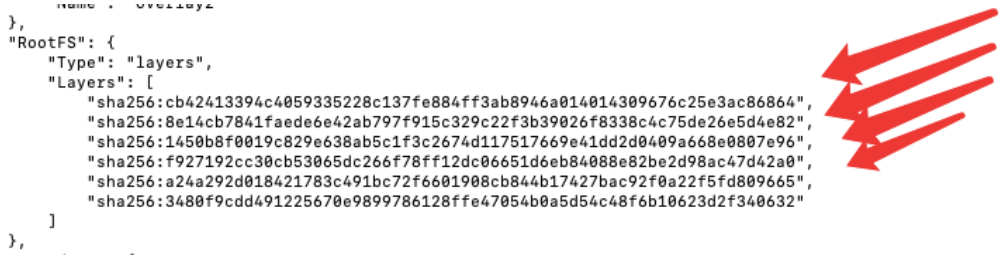
查看镜像分层的方式可以通过docker image inspect 命令!

```
[root@localhost ~]# docker image inspect redis:latest
[
    {
        "Id":
"sha256:621ceef7494adfcbe0e523593639f6625795cc0dc91a750629367a8c7b3ccebb",
        "RepoTags": [
            "redis:latest"
        ],
        "RepoDigests": [

 "redis@sha256:0f97c1c9daf5b69b93390ccbe8d3e2971617ec4801fd0882c72bf7cad3a13494
"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2021-01-13T09:45:41.527587343Z",
        "Container":
"16535cfaf84a4049b6c02840219e8473787d5610e29409049df3a41bbf77a333",
        "ContainerConfig": {
            "Hostname": "16535cfaf84a",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "6379/tcp": {}
```

```
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [

 "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "GOSU_VERSION=1.12",
                "REDIS_VERSION=6.0.10",
                "REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-
6.0.10.tar.gz",

 "REDIS_DOWNLOAD_SHA=79bbb894f9dceb33ca699ee3ca4a4e1228be7fb5547aeb2f99d921e86c
1285bd"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"redis-server\"]"
            ],
            "Image":
"sha256:222c0cecc006d8c73a04a58b5fa15ebae171a6e82a8ee8650ae616f6f1798ef4",
            "Volumes": {
                "/data": {}
            },
            "WorkingDir": "/data",
            "Entrypoint": [
                "docker-entrypoint.sh"
            ],
            "OnBuild": null,
            "Labels": {}
        },
        "DockerVersion": "19.03.12",
        "Author": "",
        "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "6379/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
```

```
 "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "GOSU_VERSION=1.12",
                "REDIS_VERSION=6.0.10",
                "REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-
6.0.10.tar.gz",

 "REDIS_DOWNLOAD_SHA=79bbb894f9dceb33ca699ee3ca4a4e1228be7fb5547aeb2f99d921e86c
1285bd"
            ],
            "Cmd": [
                "redis-server"
            ],
            "Image":
"sha256:222c0cecc006d8c73a04a58b5fa15ebae171a6e82a8ee8650ae616f6f1798ef4",
            "Volumes": {
                "/data": {}
            },
            "WorkingDir": "/data",
            "Entrypoint": [
                "docker-entrypoint.sh"
            ],
            "OnBuild": null,
            "Labels": null
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 104285909,
        "VirtualSize": 104285909,
        "GraphDriver": {
            "Data": {
                "LowerDir":
"/var/lib/docker/overlay2/6d8895d7e2a8f9eaabec1d196eb16764d53d07bebc46d6b40baa2
5866235602b/diff:/var/lib/docker/overlay2/7fb3d70a7ffd4d8f8dccc6730ea0fb3126552
04ee7f297e08392df212bb882a0/diff:/var/lib/docker/overlay2/26887ccac76ee95916c08
21165e738a7529fa71fb0fe248777b89c26119fa6b8/diff:/var/lib/docker/overlay2/a283c
4fd395fd124a80841ba4a5aa65aa3591de0521642e600b8ad07535c5a49/diff:/var/lib/docke
r/overlay2/398702ea418dc47812b1ce39a0fda1bc37f9ff37e728e8bf8cadde6825f99a02/dif
f",
                "MergedDir":
"/var/lib/docker/overlay2/4191be432db7cad94a155bc72650a0126499588c9291de3bc9b7f
6b81c3b0dde/merged",
                "UpperDir":
"/var/lib/docker/overlay2/4191be432db7cad94a155bc72650a0126499588c9291de3bc9b7f
6b81c3b0dde/diff",
                "WorkDir":
"/var/lib/docker/overlay2/4191be432db7cad94a155bc72650a0126499588c9291de3bc9b7f
6b81c3b0dde/work"
            },
```

```
            "Name": "overlay2"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [

"sha256:cb42413394c4059335228c137fe884ff3ab8946a014014309676c25e3ac86864",

"sha256:8e14cb7841faede6e42ab797f915c329c22f3b39026f8338c4c75de26e5d4e82",

"sha256:1450b8f0019c829e638ab5c1f3c2674d117517669e41dd2d0409a668e0807e96",

"sha256:f927192cc30cb53065dc266f78ff12dc06651d6eb84088e82be2d98ac47d42a0",

"sha256:a24a292d018421783c491bc72f6601908cb844b17427bac92f0a22f5fd809665",

"sha256:3480f9cdd491225670e9899786128ffe47054b0a5d54c48f6b10623d2f340632"
            ]
        },
        "Metadata": {
            "LastTagTime": "0001-01-01T00:00:00Z"
        }
    }
]
```

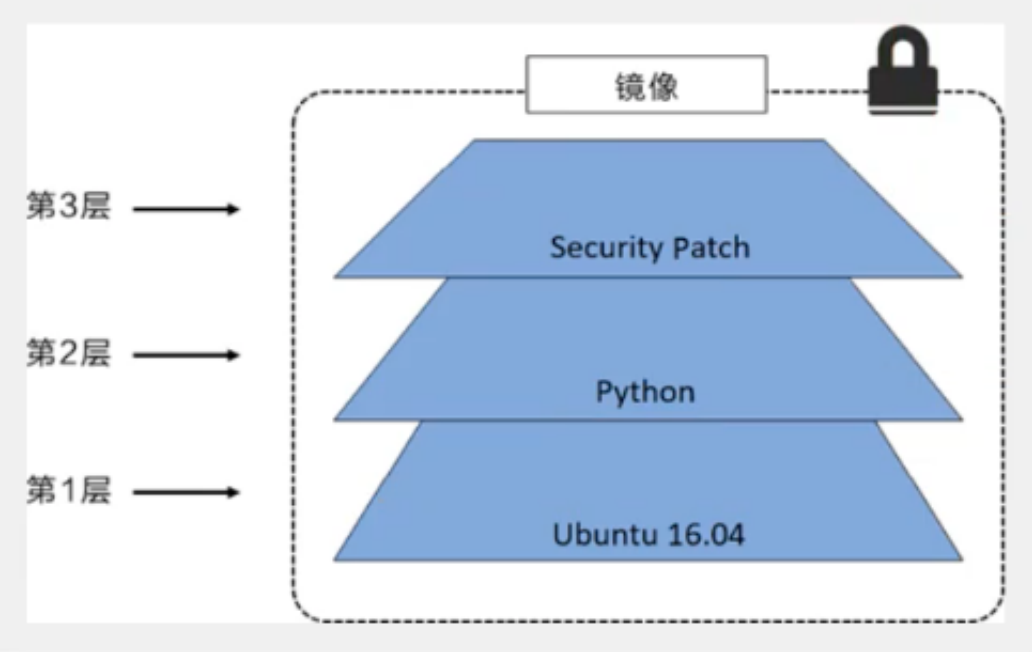
理解：

所有的docker镜像都起源于一个基础镜像层，当进行修改或增加新的内容时，就会在当前的镜像层上，创建新的镜像层。

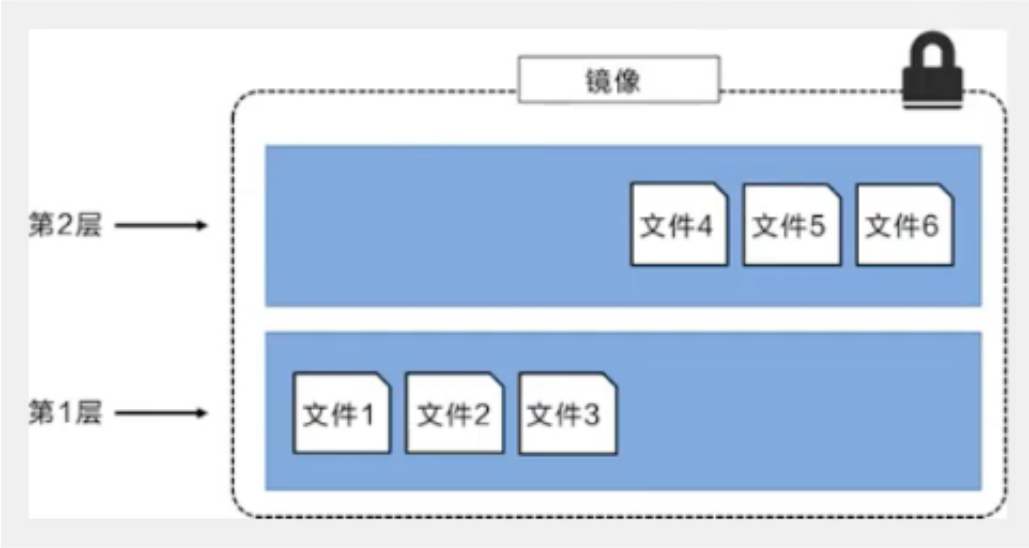
举一个简单的例子，假如基于Ubuntu linux 16.04创建一个新的镜像，这就是镜像的第一层；如果在搞镜像中添加python包，

就会在基础镜像上创建第二个镜像层；如果继续添加一个安全补丁，就会创建第三个镜像层。

改镜像当前已经包含3个镜像层如下图所示

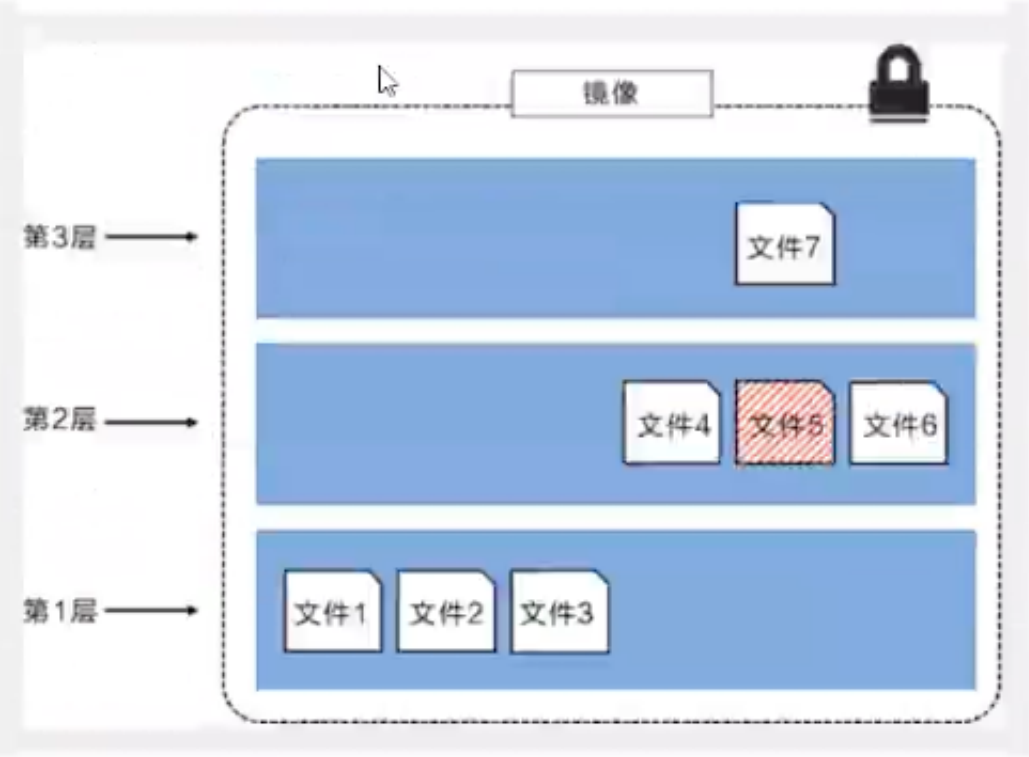


在添加额外镜像层的同时，镜像始终保持着当前所有镜像的组合，理解着一点非常重要，下图举了一个非常简单的例子，每个镜像层包含3个文件，而镜像包含了来自两个镜像层的6个文件



上面的镜像层和之前的略有区别，主要目的是便于展示文件。

下图站是了一个稍微复杂的三层镜像，在外部看来整个镜像只有6个文件，这是因为最上层的文件7时文件5的更新版



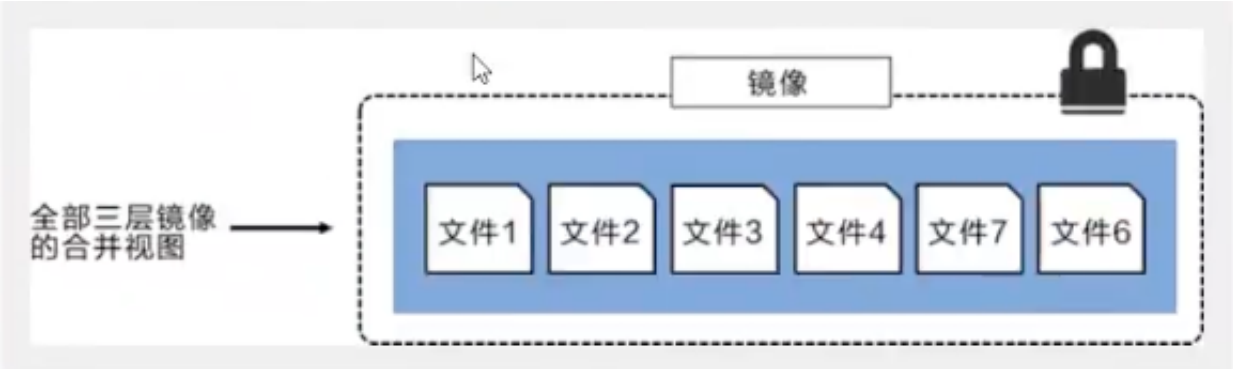
这种情况下，上层镜像的文件覆盖了底层镜像的文件，这样就使得文件的更新版本作为一个镜像层添加到镜像中，

Docker通过存储引擎（新版本采用快照机制）的方式来实现镜像层的堆栈，并保证多镜像层对外展示为统一的文件系统

Linux上可用的存储引擎有AUFS，Overlay2，Decvice Mapper，Btrfs以及ZFS。顾名思义，每种存储引擎都基于LInux中对应的文件系统或者快设备技术，并且每种存储引擎都有独有的性能特点

Docker在winows仅支持windowsfilter一种存储引擎，改引擎基于NTFS文件系统之上实现了分层和cow

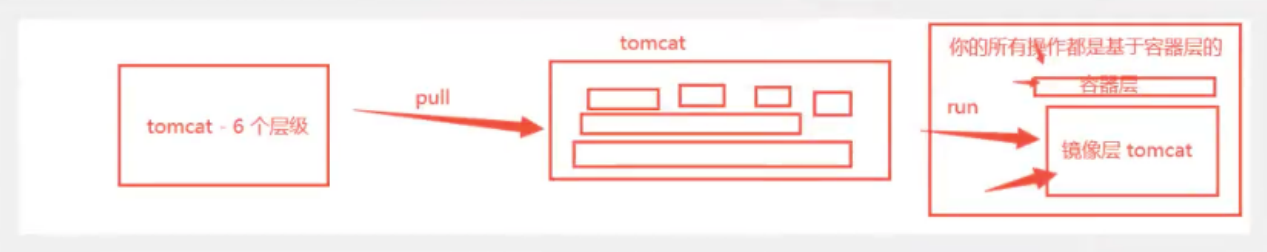
下面展示了与系统显示相同的三层镜像，所有镜像层堆叠并合并，对外提供统一的视图。



> 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，容器层之下就是镜像层！



如何提交一个自己的镜像?

commit镜像

```
docker commit 提交容器成为一个新的副本

#命令和git类似
docker commit -m="提交的描述信息" -a="作者" 容器id 目标镜像名,[TAG]
```

实战测试:

```
 # 1.启动一个默认的tomacat

 # 2.发现这个默认的tomcat是没有webapps的，镜像的原因，官方的镜像默认的webapps下面是没有文件的

 # 3,我自己拷贝进去了一个基本文件

 # 4.将我们操作过的容器通过commit提交为1个镜像！我们以后就使用我们修改过的镜像即可，这就是我们自己的一个镜像
```

```
[root@localhost ~]# docker commit -a="chenluo" -m="add webapps app" 154c72e82660 tomcat02:1.0
sha256:b3b91d9356f5db233195f4611f160f6d4c1b42c463372caa30bbdc69acfc6fcf
[root@localhost ~]# docker images
REPOSITORY            TAG       IMAGE ID       CREATED          SIZE
tomcat02              1.0       b3b91d9356f5   18 seconds ago   654MB
redis                 latest    621ceef7494a   2 months ago     104MB
tomcat                9.0       040bdb29ab37   2 months ago     649MB
tomcat                latest    040bdb29ab37   2 months ago     649MB
nginx                 latest    f6d0b4767a6c   2 months ago     133MB
centos                latest    300e315adb2f   3 months ago     209MB
portainer/portainer   latest    62771b0b9b09   8 months ago     79.1MB
elasticsearch         7.6.2     f29a1ee41030   12 months ago    791MB
[root@localhost ~]#
```

学习方式说明：理解概念但是一定要实践最后实践和理论相结合，一次搞定

```
如果你想要保存当前的容器状态，就可以通过commit提交 ，或的一个镜像，就好比vm的快照
```

到这里才算是入门!

容器数据卷

什么是容器数据卷

docker的理念回顾

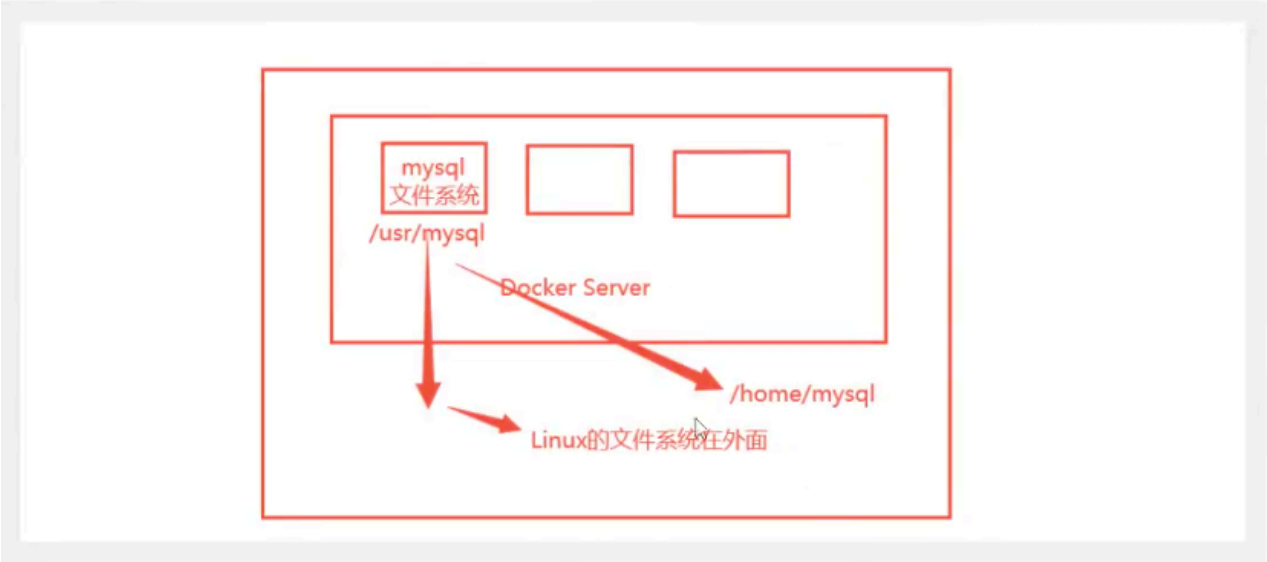
将应用和环境打包成一个镜像！

数据？如果数据在容器中，容器一旦删除，数据就会丢失！需求：数据可以持久化

MySQL，容器删了，删库跑路 ==需求：Mysql的数据可以存储在本地或者其他地方==

容器之间可以有数据共享的技术！ Docker容器中产生的数据，同步到本地！

这就是卷技术！目录的挂载，将我们容器内的目录，挂载到linux



总结：容器的持久化和同步操作！容器间也是可以数据共享的

使用数据卷

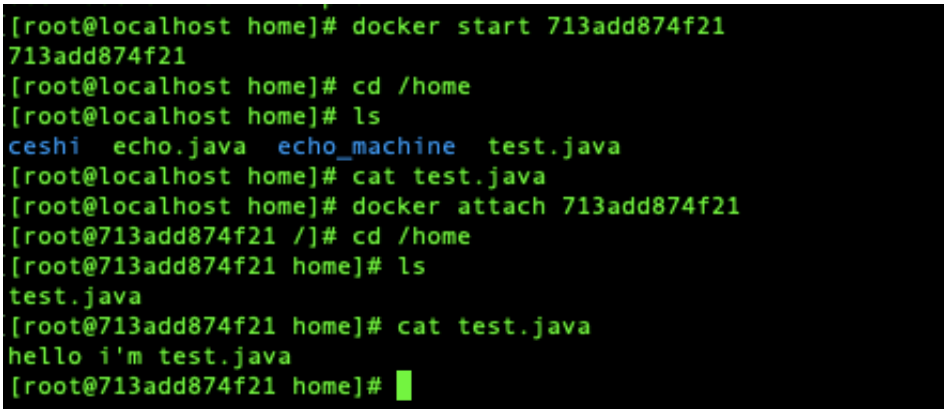
> 方式一：直接使用命令来挂载 -v

```
docker run -it -v 主机目录:容器目录 -P 主机端口:容器端口

# 测试：
[root@localhost home]# docker run -it -v /home/ceshi:/home centos /bin/bash

# 启动后可以通过docker inspect 容器id
```

测试文件的同步

再来测试！

1. 停止容器
2. 在宿主机上修改文件
3. 再次启动容器
4. 发现容器内的文件修改同步



好处：我们以后修改只需在本地修改即可，容器内会自动同步

实战：安装Mysql

思考:mysql数据持久化问题 data

```
# 获取镜像
[root@localhost ~]# docker pull mysql:5.7
# 运行容器，需要做数据挂载！启动mysql，需要配置密码的，要注意
# 官方测试：  docker run --name some-mysql-e MYSQL_ROOT_PASSWORD=my-secret-pw -d
mysql:tag

# 启动我们的
-d 后台运行
-p 端口映射
-v 数据卷挂载
-e 环境配置
--name 容器名字
[root@localhost ~]# 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 mysql101 mysql:5.7

# 启动成功之后在本地使用sqlyog链接测试
# 本地的sqlyog-h -u -p 连接到服务器的3310，3310和容器内的3306 映射
mysql -h 192.168.220.17 -P 3310 -u root -p123456

# 在本地测是创建一个数据库,查看我们映射的路径是否OK
```

```
liuguangquan@liuguangquandeMacBook-Air ~ % mysql -h 192.168.220.17 -P 3310 -u root -p123456
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.33 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

```
[root@localhost data]# ls
auto.cnf    client-cert.pem  ibdata1      ibtmp1             private_key.pem  server-key.p
ca-key.pem  client-key.pem   ib_logfile0  mysql              public_key.pem   sys
ca.pem      ib_buffer_pool   ib_logfile1  performance_schema  server-cert.pem
[root@localhost data]# ls
auto.cnf    client-cert.pem  ibdata1      ibtmp1  performance_schema  server-cert.pem
ca-key.pem  client-key.pem   ib_logfile0  inform  private_key.pem     server-key.pem
ca.pem      ib_buffer_pool   ib_logfile1  mysql   public_key.pem      sys
[root@localhost data]#
```

假设我们将容器删除

```
[root@localhost data]# docker rm -f mysql101
mysql101
```

发现，我们挂载到本地的数据卷 依旧没有丢失，这就实现了容器数据持久化功能

具名挂载和匿名挂载

```
# 匿名挂载
-v 容器内路径
docker run -d -p --name ngnix01 -v /etc/ngnix

#查看所有的volumn的情况
[root@localhost data]# docker volume ls
DRIVER    VOLUME NAME
local     69ba25f4bcc878ef4968aae079f5f45f8b72ff043b49e3d7dcf9e0fa0cbdf59a
local     c16215573a37f23685d7460e6168165879983022ca76df042d6e01cffdd762ed
# 这里发现，这种就是匿名挂载，我们在-v 的时只写了容器内的路径，没有写容器外的路径

# 具名挂载
[root@localhost data]# docker run -d -P --name nginx03 -v juming-
nginx:/etc/nginx nginx
cec0d66667b776760d8d36063892c5b27d0bf0863549eed191f7cce88fa6a22f
[root@localhost data]# docker volume ls
DRIVER    VOLUME NAME
local     69ba25f4bcc878ef4968aae079f5f45f8b72ff043b49e3d7dcf9e0fa0cbdf59a
local     c16215573a37f23685d7460e6168165879983022ca76df042d6e01cffdd762ed
local     juming-nginx
# 通过 -v 卷名:容器内路径
# 查看一下这个卷
```

```
[root@localhost data]# docker volume inspect juming-nginx
[
    {
        "CreatedAt": "2021-04-07T04:59:45-04:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/juming-nginx/_data",
        "Name": "juming-nginx",
        "Options": null,
        "Scope": "local"
    }
]
```

所有docker容器内的卷，没有指定目录的情况下都是在 **/var/lib/docker/volumes/juming-nginx/_data**

我们通过具名挂载可以更方便的找到我们的卷，大多数情况使用**具名挂载**

```
# 如何确定是具名挂载还是匿名挂载
-v 容器内路径          #匿名挂载
-v 卷名:容器内路径    #具名挂载
-v /宿主机的路径:容器内路径 #指定路径挂载
```

扩展：

```
# 通过-v 容器内路径，加上ro/rw改变读写权限
ro readonly
re 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 nginx01 -v juming-nginx:/etc/nginx;ro nginx
docker run -d -P --name nginx01 -v juming-nginx:/etc/nginx;rw nginx

#ro 只要看到ro就说明这个路径只能通过宿主机来操作，容器内部是无法操作的
```

初始DockerFile

DockerFile就是用来构建docker镜像的构建文件！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

```
# 创建一个dockerfile的文件，名字可以随机 建议Dockerfile
# 文件中的内容   指令    参数
From centos
。。。

# 每个命令都是镜像的一层
```

```
# 启动自己写的容器
```

```
[root@kuangshen docker-test-volume]# docker run -it 5d04f189a434 /bin/bash
[root@c655f70789c3 /]# ls -l
total 56
lrwxrwxrwx   1 root root    7 May 11  2019 bin -> usr/bin
drwxr-xr-x   5 root root  360 May 15 11:56 dev
drwxr-xr-x   1 root root 4096 May 15 11:56 etc
drwxr-xr-x   2 root root 4096 May 11  2019 home
lrwxrwxrwx   1 root root    7 May 11  2019 lib -> usr/lib
lrwxrwxrwx   1 root root    9 May 11  2019 lib64 -> usr/lib64
drwx------   2 root root 4096 Jan 13 21:48 lost+found
drwxr-xr-x   2 root root 4096 May 11  2019 media
drwxr-xr-x   2 root root 4096 May 11  2019 mnt
drwxr-xr-x   2 root root 4096 May 11  2019 opt
dr-xr-xr-x 121 root root    0 May 15 11:56 proc
dr-xr-x---   2 root root 4096 Jan 13 21:49 root
drwxr-xr-x  11 root root 4096 Jan 13 21:49 run
lrwxrwxrwx   1 root root    8 May 11  2019 sbin -> usr/sbin
drwxr-xr-x   2 root root 4096 May 11  2019 srv
dr-xr-xr-x  13 root root    0 Mar 23 14:00 sys
drwxrwxrwt   7 root root 4096 Jan 13 21:49 tmp
drwxr-xr-x  12 root root 4096 Jan 13 21:49 usr
drwxr-xr-x  20 root root 4096 Jan 13 21:49 var
drwxr-xr-x   2 root root 4096 May 15 11:56 volume01
drwxr-xr-x   2 root root 4096 May 15 11:56 volume02
[root@c655f70789c3 /]#
```

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

这个卷和外部有一个同步的目录

```
FROM centos

VOLUME ["volume01","volume02"]

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

匿名挂载

查看卷挂载的路径

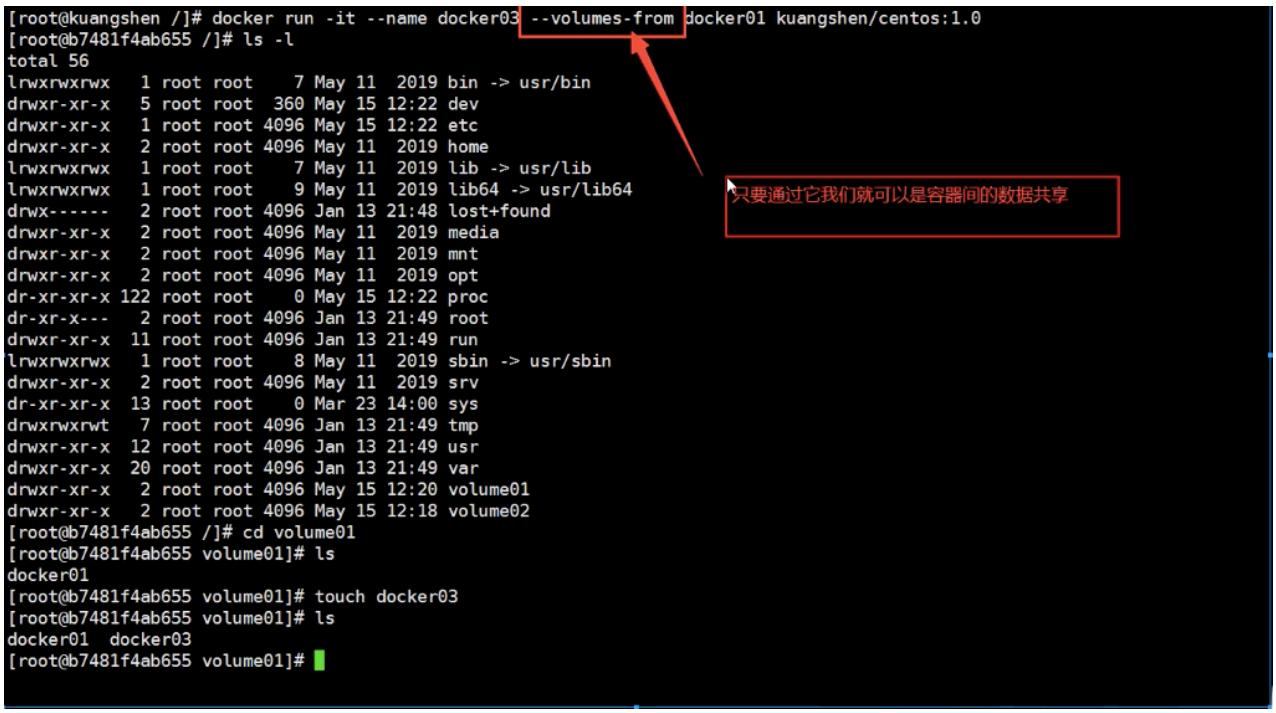
```
},
"Mounts": [
    {
        "Type": "volume",
        "Name": "fa24189079ae28c2993bf312ec8791fef127d9696982044cfca0df420082a98c",
        "Source": "/var/lib/docker/volumes/fa24189079ae28c2993bf312ec8791fef127d9696982044cfca0df420082a98c/_data",
        "Destination": "volume01",
        "Driver": "local",
        "Mode": "",
        "RW": true,
        "Propagation": ""
    },
    {
        "Type": "volume",
        "Name": "5404b86e92683e18d7446762d22117bbaf78adad7ff2a5b1ce3795aa08739aec",
        "Source": "/var/lib/docker/volumes/5404b86e92683e18d7446762d22117bbaf78adad7ff2a5b1ce3795aa08739aec/_data",
        "Destination": "volume02",
        "Driver": "local",
        "Mode": "",
        "RW": true,
        "Propagation": ""
    }
],
```

假设构建镜像时没有挂载卷要手动挂载镜像　　-v 卷名:容器内路径!

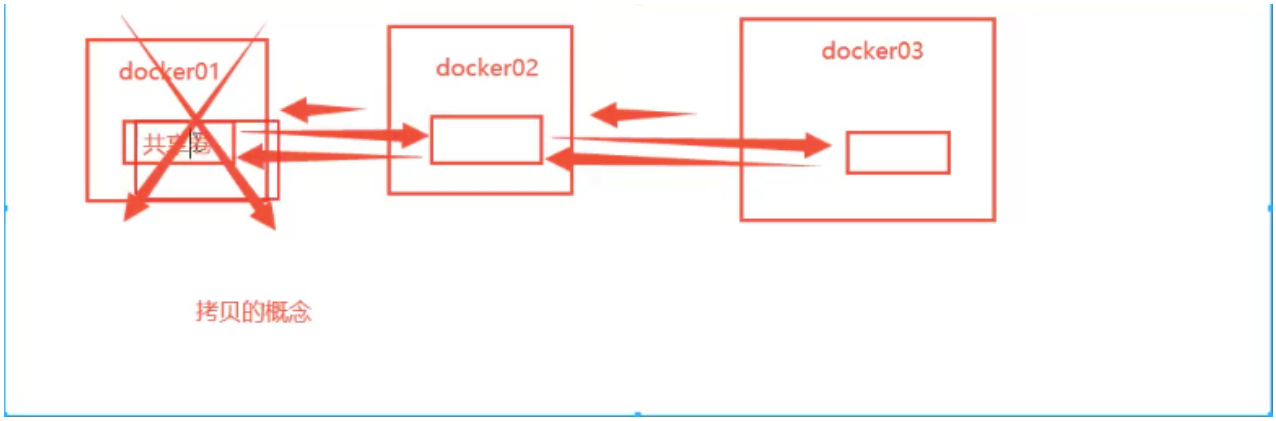
数据卷容器

多个mysql同步数据


```
docker run -it --name docker02 --volumes-from docker01 kuangshen/centos:1.0
```




```
# 测试，可以删除docker01，查看一下docker02和docker03是否可以访问这个文件
# 测试依旧可以访问
```



多个mysql实现数据共享

```
[root@localhost ~]# docker run -d -p 3310:3306 -v /etc/mysql/conf.d -v
/home/mysql/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 --name mysql101
mysql:5.7

[root@localhost ~]# docker run -d -p 3310:3306 -e MYSQL_ROOT_PASSWORD=123456 --
name mysql102 --volumes-form mysql101 mysql:5.7

# 这个时候可以实现两个容器同步
```

结论:

容器之间信息传递，数据卷容器的生命周期可以持续到没有容器使用为止

但是一旦持久化到了本地，知这时候本地的数据是不会删除的

DockerFile

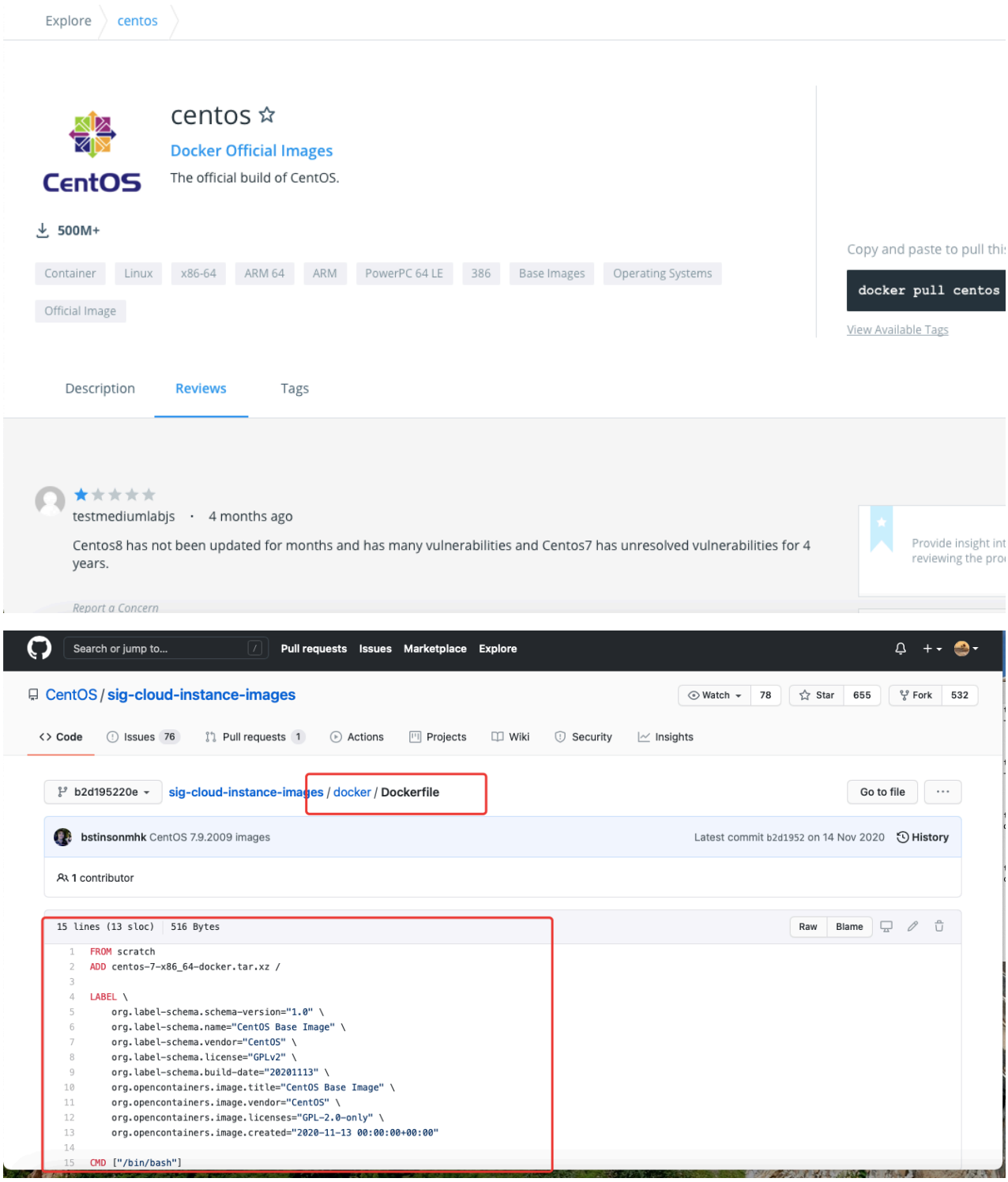
DockerFile的介绍:

dockerfile是用来构建docker镜像的文件

构建步骤:

1. 编写一个dockerfile文件
2. dockerbuild构建一个镜像
3. docker run运行镜像
4. docker push发布镜像（dockerhub， 阿里云镜像仓库！）

查看一下官方



很多官方的镜像都是基础包，很多功能都没有，我么通常自己搭建自己的镜像

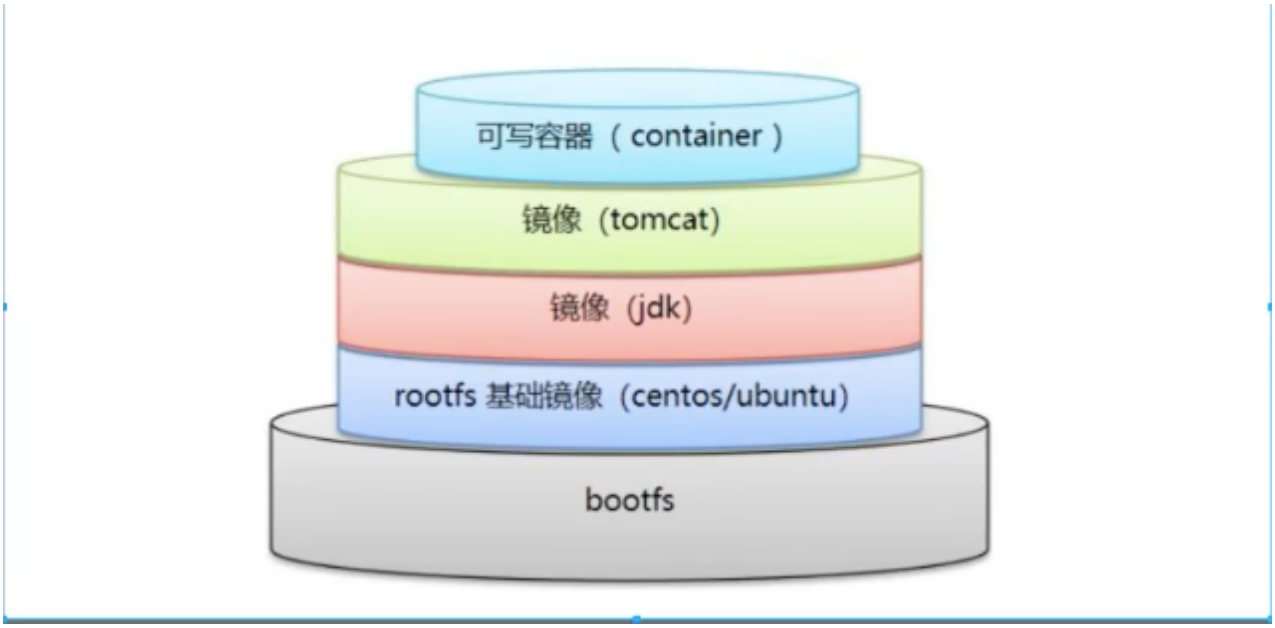
广泛既然可以制作镜像我们也可以!

DockerFile的构建过程

基础知识:

1. 每个保留关键字（指令）都是大写字母
2. 执行从上到下的顺序
3. #表示注释

4. 每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，就要编写dockefile，这个文件十分简单

Docker镜像 逐渐成为了企业交付的标准，必须要掌握！

步骤：开发，部署，运维。。。缺一不可

Docker File：构建文件，定义了一切的步骤，源代码

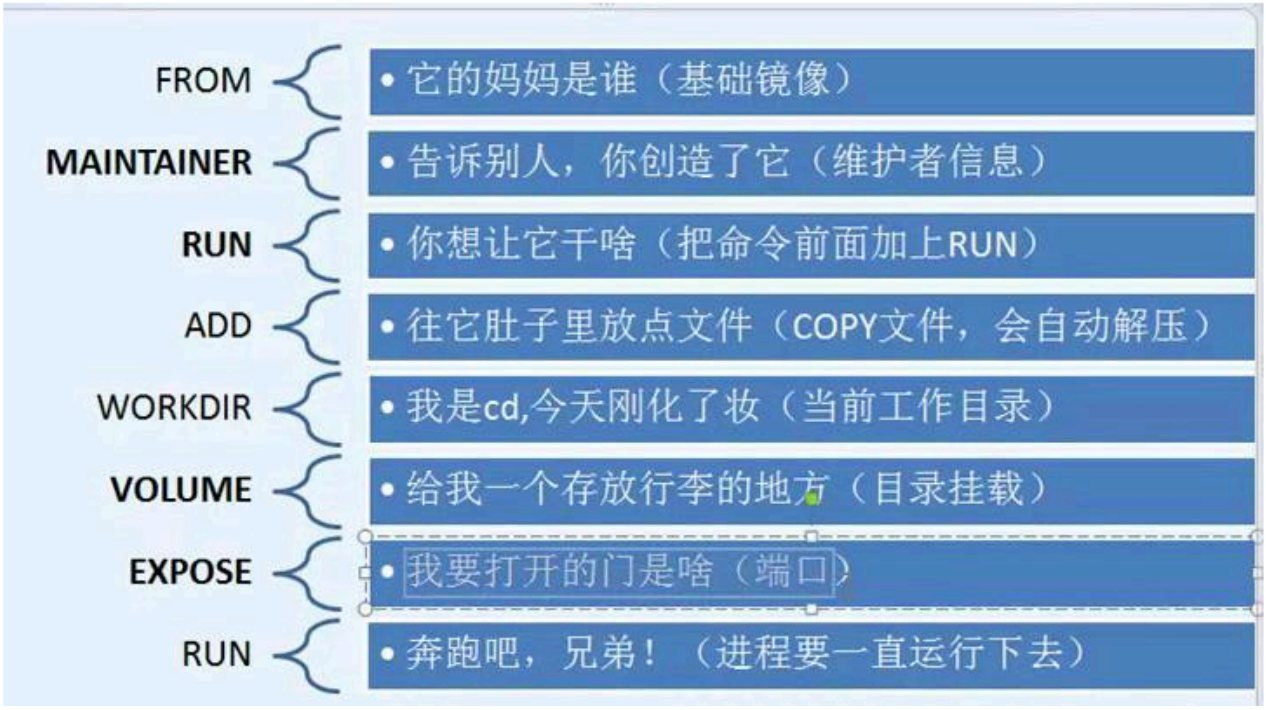
Dockerimage：通过Docjerfile构建生成的镜像，最终运行的产品，原来是一个jar包，war包

Docker容器：镜像运行起来提供服务器

Doclkerfile的指令

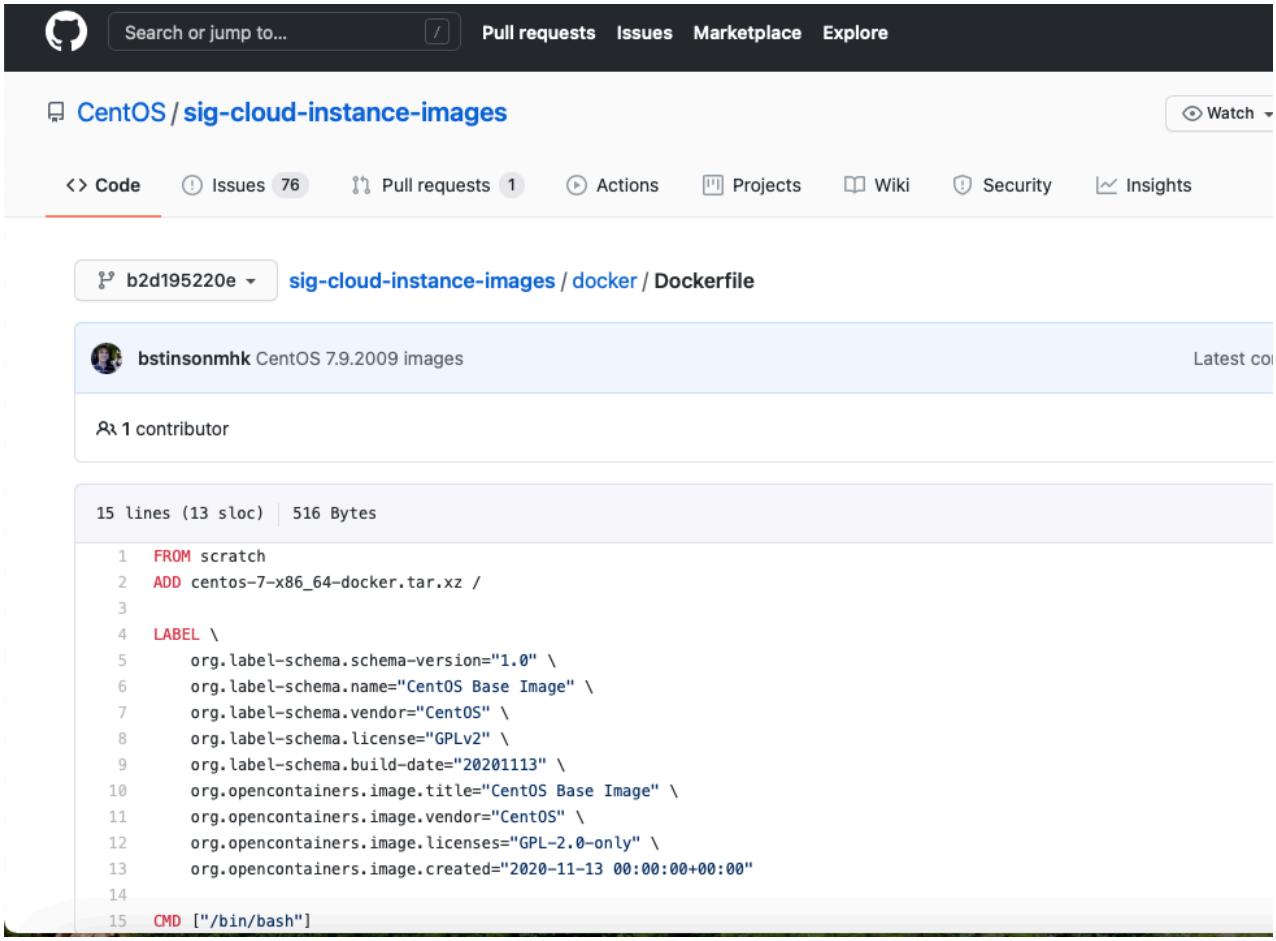
以前的话用别人的，现在我们知道这些指令后就创建自己的镜像

```
FROM                    # 基础镜像 centos ubuntu，一切从这里构建
MAINTAINER              # 镜像是谁写的，姓名+邮箱
RUN                     # Docker镜像构建时需要运行的命令
ADD                     # 步骤， tomacat镜像，这个tomcat压缩包（添加内容）
WORKDIR                 # 镜像的工作     /bin/bash
VUOLUME                 # 挂载的目录
EXPOST                  # 暴露端口，保留端口配置
CMD                     # 指定这个容器的时候要运行的命令 cmd echo，只有最后一个会生效，可
被替代
ENTRYPOINT              # 指定这个容器的时候要运行的命令 ，可以追加命令
ONBUILD                 # 当构建一个被继承的DockerFile 这个时候就会运行ONBUILD的指令触发
指令
COPY                    # 类似ADD命令 ，将我们的文件拷贝到镜像中
ENV                     # 构建的时候设置环境变量
```

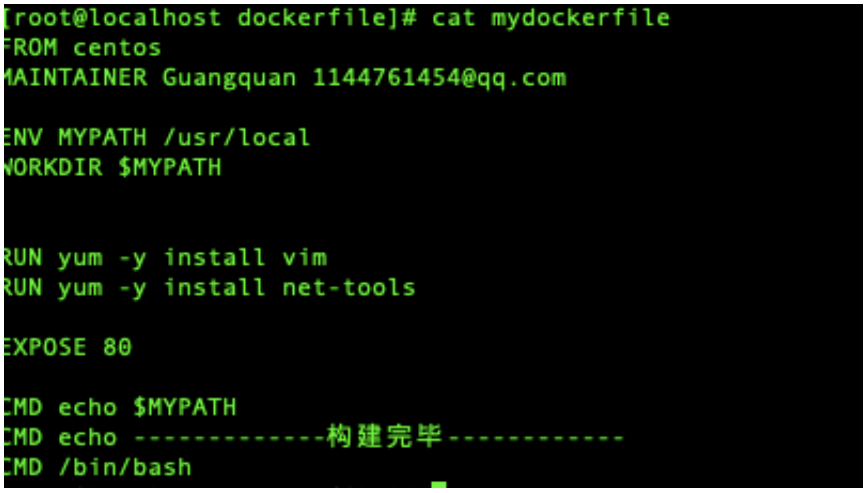


实战测试

DockerHub中99%的镜像都是从这个基础镜像过来的FROM scratch，然后配置需要的软件和配置来进行的构建的

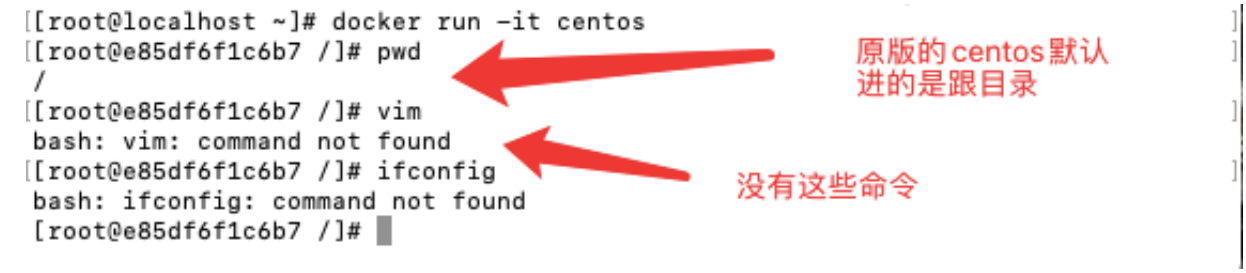


> 创建一个自己的centos




```
# 1.编写DockerFile文件
[root@localhost dockerfile]# cat mydockerfile
FROM centos
MAINTAINER Guangquan 1144761454@qq.com

ENV MYPATH /usr/local
WORKDIR $MYPATH


RUN yum -y install vim
RUN yum -y install net-tools

EXPOSE 80

CMD echo $MYPATH
CMD echo -------------构建完毕------------
CMD /bin/bash
# 2.通过文件构建镜像
# 命令 docker build -f dockerfile文件路径 -t 镜像名:[tag] .
docker build -f mydockerfile -t mycentos:1.0
Successfully built 534dcfd25443
Successfully tagdged mycentos:1.0
# 3.测试运行
```

之前原生的centos



我们增加之后的镜像

```
[[root@localhost dockerfile]# docker run -it mycentos:1.0
[[root@41a7f8419159 local]# pwd
/usr/local
[[root@41a7f8419159 local]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.6  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:ac:11:00:06  txqueuelen 0  (Ethernet)
        RX packets 23  bytes 2762 (2.6 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[[root@41a7f8419159 local]# vim test
[root@41a7f8419159 local]#
```

我们可以列出本地镜像的变更历史,

```
[root@localhost dockerfile]# docker history 534dcfd25443
IMAGE          CREATED          CREATED BY                                      SIZE      COMMENT
534dcfd25443   10 minutes ago   /bin/sh -c #(nop)  CMD ["/bin/sh" "-c" "/bin…   0B
b5a10b4ab2a8   10 minutes ago   /bin/sh -c #(nop)  CMD ["/bin/sh" "-c" "echo…   0B
c200932d8c37   10 minutes ago   /bin/sh -c #(nop)  CMD ["/bin/sh" "-c" "echo…   0B
b789ec0d4808   10 minutes ago   /bin/sh -c #(nop)  EXPOSE 80                    0B
c3eee4896489   10 minutes ago   /bin/sh -c yum -y install net-tools             23.3MB
ac4eccafbc08   10 minutes ago   /bin/sh -c yum -y install vim                   58MB
98fab7fbe1f1   11 minutes ago   /bin/sh -c #(nop) WORKDIR /usr/local            0B
f7826c6b840d   11 minutes ago   /bin/sh -c #(nop)  ENV MYPATH=/usr/local        0B
ced0cb743f31   11 minutes ago   /bin/sh -c #(nop)  MAINTAINER Guangquan 1144…   0B
300e315adb2f   4 months ago     /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      4 months ago     /bin/sh -c #(nop)  LABEL org.label-schema.sc…   0B
<missing>      4 months ago     /bin/sh -c #(nop) ADD file:bd7a2aed6ede423b7…   209MB
```

从这我拿到一个docker就知道他是怎么做的了

> CMD 和 ENTRYPOINT的区别

```
CMD                     # 指定这个容器的时候要运行的命令 cmd echo，只有最后一个会生效，可被替代
ENTRYPOINT              # 指定这个容器的时候要运行的命令 ，可以追加命令
```

测试cmd

```
# 编写dockerfile文件
[root@localhost dockerfile]# vim dockerfile-cmd-test
FROM centos
CMD ["ls","-a"]

# 构建镜像
[root@localhost dockerfile]# docker build -f dockerfile-cmd-test -t cmdtest .

# run运行，发现我们的ls -a命令生效了
[root@localhost dockerfile]# docker run cmdtest
```

```
.
..
.dockerenv
bin
dev
etc
home
lib
lib64
lost+found
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
[root@localhost dockerfile]#
[root@localhost dockerfile]#

# 想追加命令-l
[root@localhost dockerfile]# docker run cmdtest -l
docker: Error response from daemon: OCI runtime create failed:
container_linux.go:367: starting container process caused: exec: "-l":
executable file not found in $PATH: unknown.

# cmd情况下 -l替换了CMD["ls","-a"]命令，-l不是命令所以报错
```

测试ENTRYPOINT

```
# 编写dockerfile文件
[root@localhost dockerfile]# vim dockerfile-cmd-test
FROM centos
ENTRYPOINT ["ls","-a"]


[root@localhost dockerfile]# docker run entrypoint-test
.
..
.dockerenv
bin
dev
etc
```

```
home
lib
lib64
lost+found
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
[root@localhost dockerfile]# docker build -f dockerfile-entrypoint-test -t
entrypoint-test .
Sending build context to Docker daemon   16.9kB
Step 1/2 : FROM centos
 ---> 300e315adb2f
Step 2/2 : ENTRYPOINT ["ls","-a"]
 ---> Using cache
 ---> 2dbbec8e487f
Successfully built 2dbbec8e487f
Successfully tagged entrypoint-test:latest

# 我们的命令是直接拼接到后面的
[root@localhost dockerfile]# docker run entrypoint-test -l
total 0
drwxr-xr-x.   1 root root   6 Apr  7 13:19 .
drwxr-xr-x.   1 root root   6 Apr  7 13:19 ..
-rwxr-xr-x.   1 root root   0 Apr  7 13:19 .dockerenv
lrwxrwxrwx.   1 root root   7 Nov  3 15:22 bin -> usr/bin
drwxr-xr-x.   5 root root 340 Apr  7 13:19 dev
drwxr-xr-x.   1 root root  66 Apr  7 13:19 etc
drwxr-xr-x.   2 root root   6 Nov  3 15:22 home
lrwxrwxrwx.   1 root root   7 Nov  3 15:22 lib -> usr/lib
lrwxrwxrwx.   1 root root   9 Nov  3 15:22 lib64 -> usr/lib64
drwx------.   2 root root   6 Dec  4 17:37 lost+found
drwxr-xr-x.   2 root root   6 Nov  3 15:22 media
drwxr-xr-x.   2 root root   6 Nov  3 15:22 mnt
drwxr-xr-x.   2 root root   6 Nov  3 15:22 opt
dr-xr-xr-x. 284 root root   0 Apr  7 13:19 proc
dr-xr-x---.   2 root root 162 Dec  4 17:37 root
drwxr-xr-x.  11 root root 163 Dec  4 17:37 run
lrwxrwxrwx.   1 root root   8 Nov  3 15:22 sbin -> usr/sbin
drwxr-xr-x.   2 root root   6 Nov  3 15:22 srv
dr-xr-xr-x.  13 root root   0 Apr  7 08:01 sys
```

```
drwxrwxrwt.   7 root root 145 Dec  4 17:37 tmp
drwxr-xr-x.  12 root root 144 Dec  4 17:37 usr
drwxr-xr-x.  20 root root 262 Dec  4 17:37 var
[root@localhost dockerfile]#
```

DockerFile中很懂命令都十分相似，我们都需要了解他，最好是进行测试

实战：Tomcat镜像

1. 准备tomacat镜像压缩包，jdk压缩包！
2. 编写dockerfile文件，官方命名“Dockerfile”，bulid时会自动寻找这个文件，不需要-f指定了
3. 构建镜像

```
# docker build
```

4. 启动镜像
5. 访问测试
6. 发布项目（由于做了卷挂载，我们直接在本地编写项目，就可以发布了）

发布自己的镜像

> Dckerhub

1. 地址https://registry.hub.docker.com 注册自己的账号
2. 确定账号可以登陆
3. 在我们的服务器上提交自己的镜像

```
[root@localhost dockerfile]# docker login --help

Usage:  docker login [OPTIONS] [SERVER]

Log in to a Docker registry.
If no server is specified, the default is defined by the daemon.

Options:
  -p, --password string   Password
      --password-stdin    Take the password from stdin
  -u, --username string   Username
[root@localhost dockerfile]#
`
[root@localhost dockerfile]# docker login -u echogyty
Password:
WARNING! Your password will be stored unencrypted in
/root/.docker/config.json.
Configure a credential helper to remove this warning. See
```

```
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

4. 登陆完毕后就可以提交自己的镜像了,就是一步push

```
# push自己的镜像到服务器上!
[root@kuangshen tomcat]# docker push diytomcat
The push refers to repository [docker.io/library/diytomcat]
fcc7fccb8e04: Preparing
b5577f344233: Preparing
bdcb94365850: Preparing
1c5bd81521f5: Preparing
0683de282177: Preparing
denied: requested access to the resource is denied # 拒绝

# push镜像的问题?
[root@kuangshen tomcat]# docker push kuangshen/diytomcat:1.0
The push refers to repository [docker.io/kuangshen/diytomcat2]
An image does not exist locally with the tag: kuangshen/diytomcat2

# 解决,增加一个 tag
[root@kuangshen tomcat]# docker tag f8559daf1fc2 kuangshen/tomcat:1.0

# docker psuh上去即可!
[root@kuangshen tomcat]# docker push kuangshen/tomcat:1.0
The push refers to repository [docker.io/kuangshen/tomcat]
fcc7fccb8e04: Preparing
b5577f344233: Preparing
bdcb94365850: Preparing
1c5bd81521f5: Preparing
0683de282177: Preparing
```

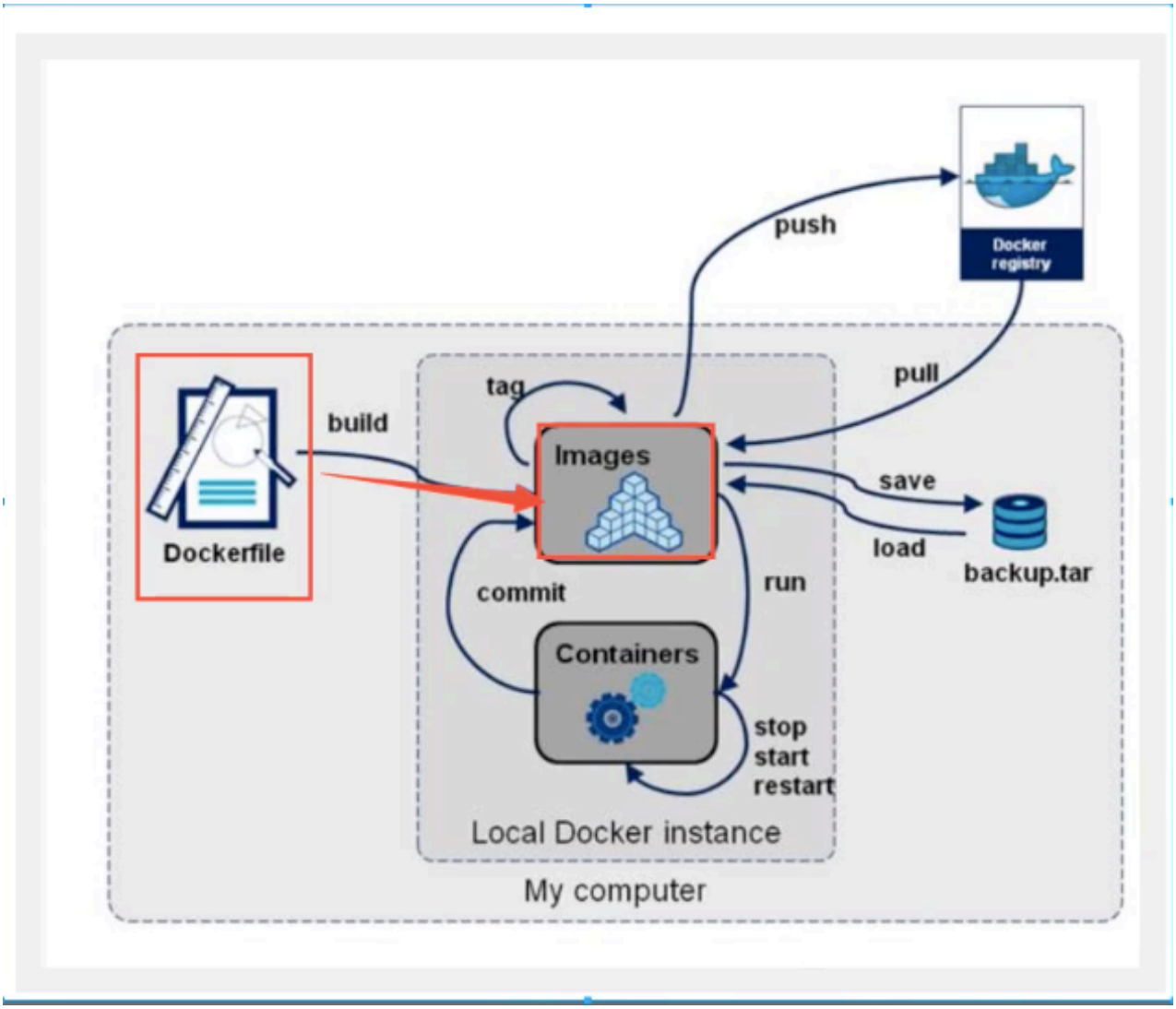
提交的时候也是按照层级来进行提交

> 发布阿里云镜像服务

1. 登陆阿里云
2. 找到容器镜像服务
3. 创建命名空间
4. 创建容器镜像
5. 浏览阿里云信息

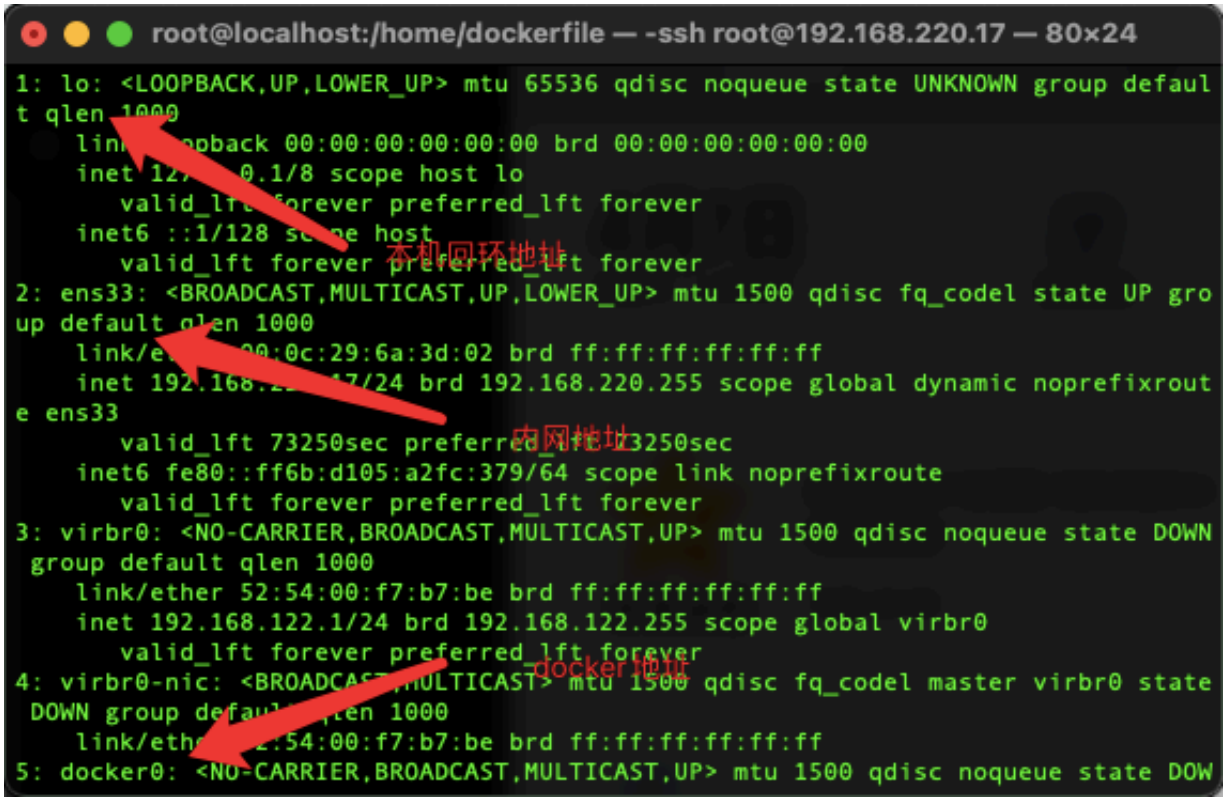
阿里云镜像参考官方地址

小结



Docker网络

理解Docker网络



三个网络：

```
# docker是如何处理容器网络访问的
```

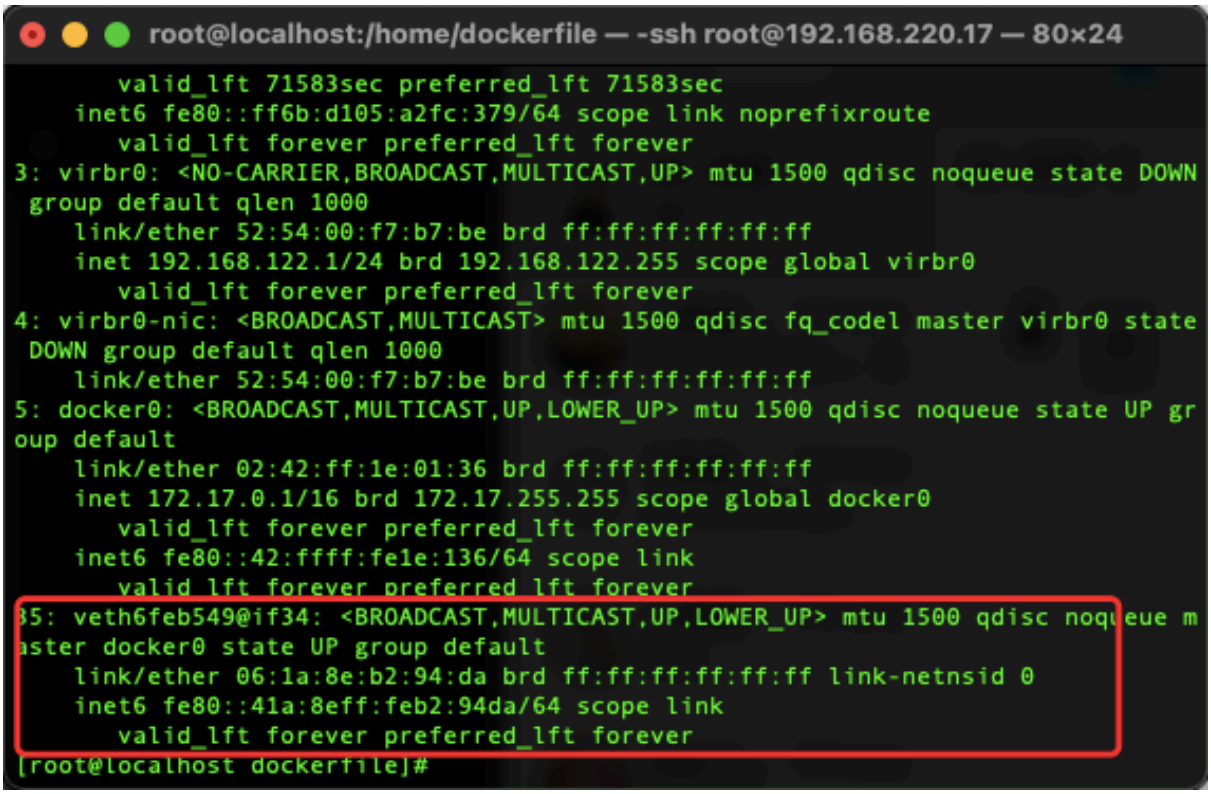
```
[root@localhost dockerfile]# docker run -d -P --name tomcat01 tomcat

#查看容器的内部网络地址，发现容器启动的时候会得到eth0@if35这样的IP地址，docker分配的
[root@localhost dockerfile]# docker exec -it tomcat01 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
34: eth0@if35: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever

#思考 linux服务器能不能ping通
[root@localhost dockerfile]# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.106 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.062 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.081 ms
64 bytes from 172.17.0.2: icmp_seq=4 ttl=64 time=0.067 ms
```

```
64 bytes from 172.17.0.2: icmp_seq=5 ttl=64 time=0.086 ms
64 bytes from 172.17.0.2: icmp_seq=6 ttl=64 time=0.109 ms
64 bytes from 172.17.0.2: icmp_seq=7 ttl=64 time=0.087 ms
64 bytes from 172.17.0.2: icmp_seq=8 ttl=64 time=0.190 ms
64 bytes from 172.17.0.2: icmp_seq=9 ttl=64 time=0.087 ms
64 bytes from 172.17.0.2: icmp_seq=10 ttl=64 time=0.085 ms
64 bytes from 172.17.0.2: icmp_seq=11 ttl=64 time=0.088 ms
64 bytes from 172.17.0.2: icmp_seq=12 ttl=64 time=0.086 ms
64 bytes from 172.17.0.2: icmp_seq=13 ttl=64 time=0.100 ms
^C
--- 172.17.0.2 ping statistics ---
13 packets transmitted, 13 received, 0% packet loss, time 296ms
rtt min/avg/max/mdev = 0.062/0.094/0.190/0.032 ms
```

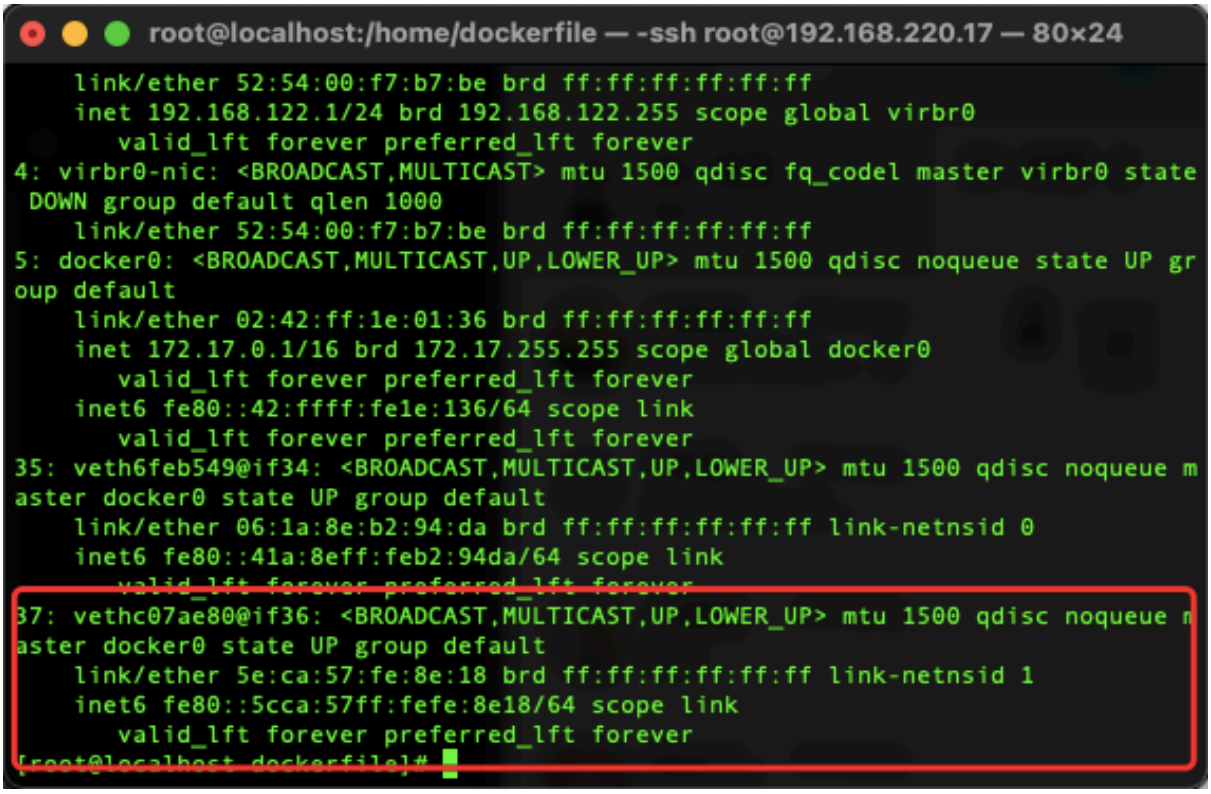
> 原理：

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

   再次测试

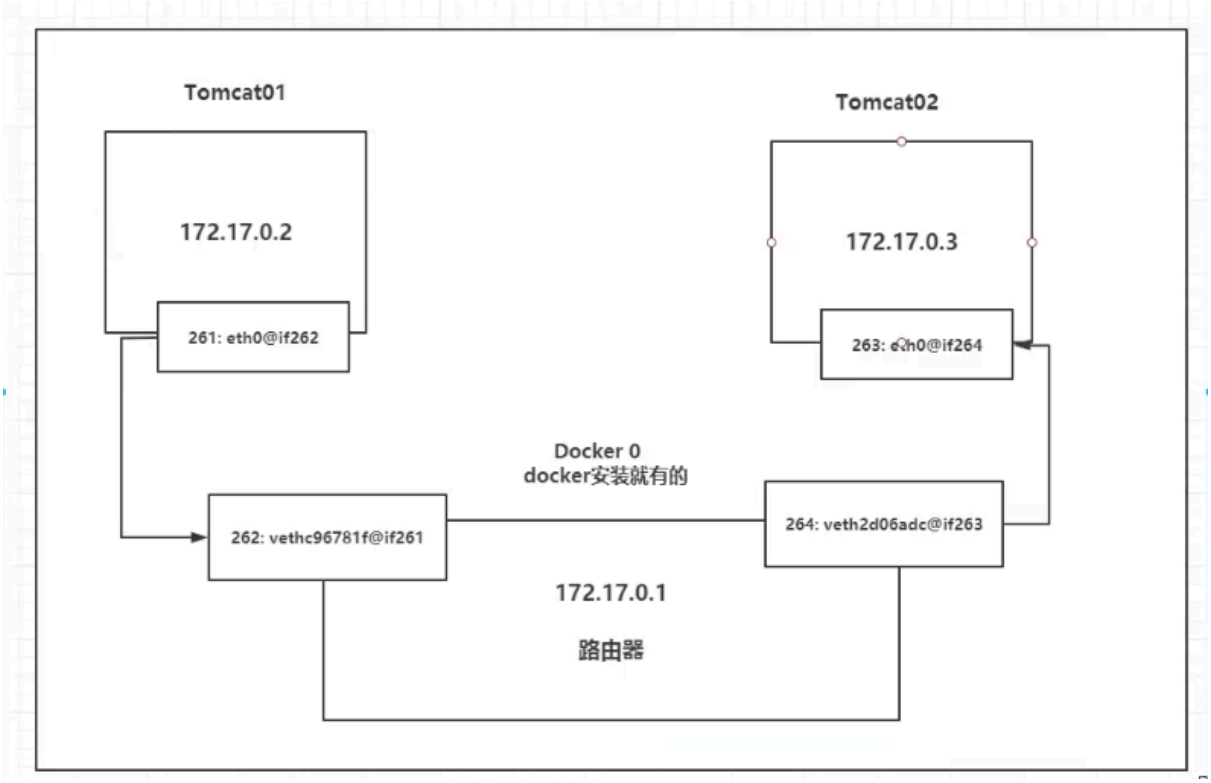
2. 再启动一个容器,发现又多了一块网卡




```
# 我们发现这些容器带来的技术都是一对一对的
# evth-pair 就是一对的虚拟设备接口，他们都是成对的，一段连着协议，一段彼此相连
# 正因为有这个特性，通常用evth-pair技术充当一个桥梁，连接各种虚拟设备
# openstack，DOcker容器之间的连接，OVS的连接，都是使用evth-pair的技术
```

3. 我们来测试tomcat01和tomcat02之间是否能ping通

```
[root@localhost dockerfile]# docker exec -it tomcat02 ping 172.17.0.2

# 结论容器之间是可以互相ping通的
```

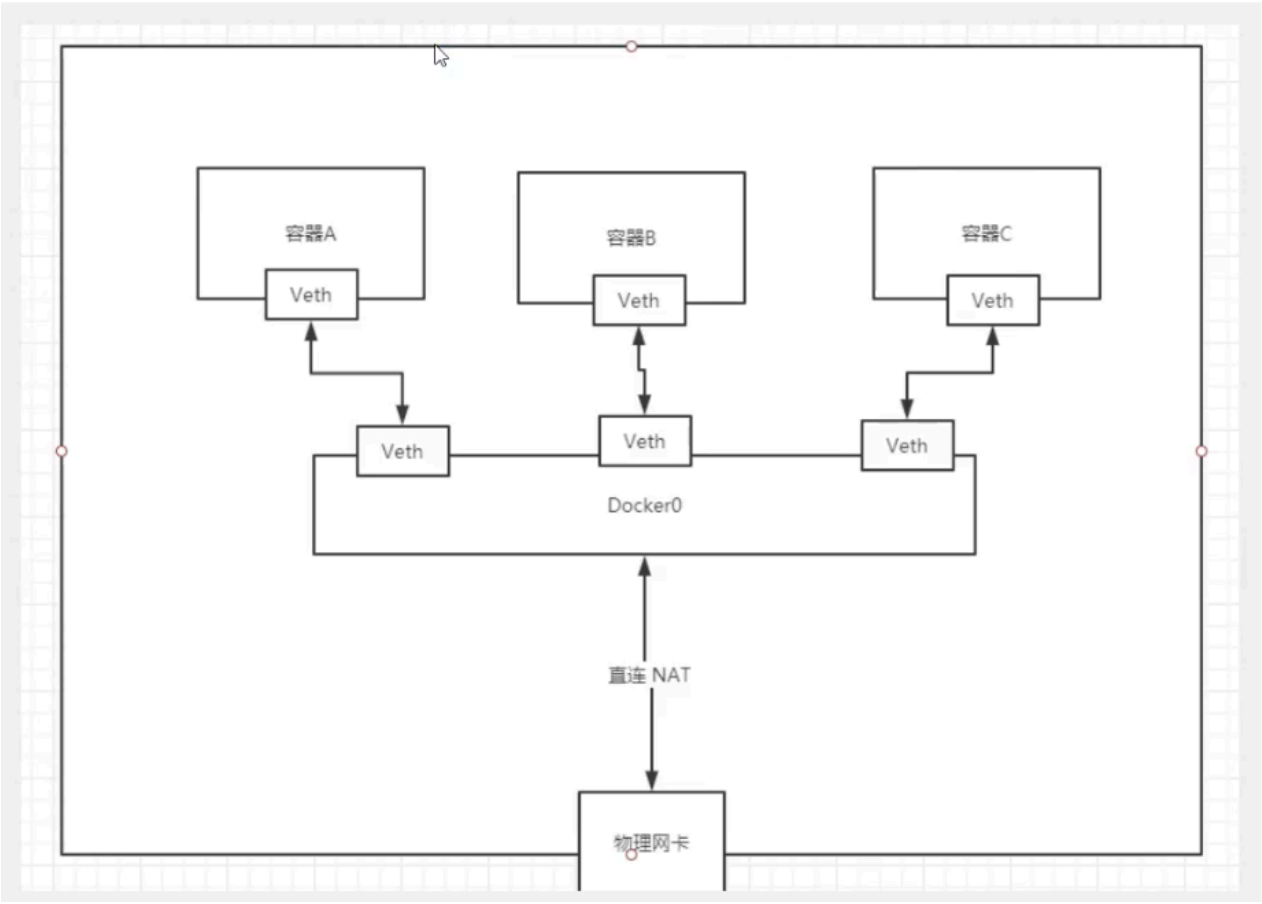


结论：tomcat01和tomcat02是公用的1个交换机，docker0

所有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

> 小结

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



Docker中的所有的网络接口都是虚拟的，虚拟的转发效率高！（内网传递文件！）

只要容器删除对应的网桥就没了

> 思考：我们编写了一个微服务 database url=ip，项目不重启，数据库ip换了，我们希望可以通过名字访问服务

```
[root@localhost dockerfile]# docker exec -it tomcat02 ping tomcat01
ping: tomcat01: Name or service not known

#如何解决呢
[root@localhost dockerfile]# docker exec -it tomcat02 ping tomcat01
ping: tomcat01: Name or service not known
# 通过--link就可以解决了
docker run -it --name=tomcat03 --link tomcat02 tomcat
[root@localhost dockerfile]# docker exec -it tomcat03 ping tomcat02
PING tomcat02 (172.17.0.3) 56(84) bytes of data.
64 bytes from tomcat02 (172.17.0.3): icmp_seq=1 ttl=64 time=0.131 ms
64 bytes from tomcat02 (172.17.0.3): icmp_seq=2 ttl=64 time=0.135 ms
64 bytes from tomcat02 (172.17.0.3): icmp_seq=3 ttl=64 time=0.120 ms
ç64 bytes from tomcat02 (172.17.0.3): icmp_seq=4 ttl=64 time=0.091 ms
^C
--- tomcat02 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 68ms
rtt min/avg/max/mdev = 0.091/0.119/0.135/0.018 m
```

探究inspect

其实tomacat03就是在本地配置了tomcat02

```
# 查看hosts配置
[root@localhost dockerfile]# docker exec -it tomcat03 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
172.17.0.3  tomcat02 2110f95268bc
172.17.0.4  896c41fa1fd8
```

--link再hosts中增加了

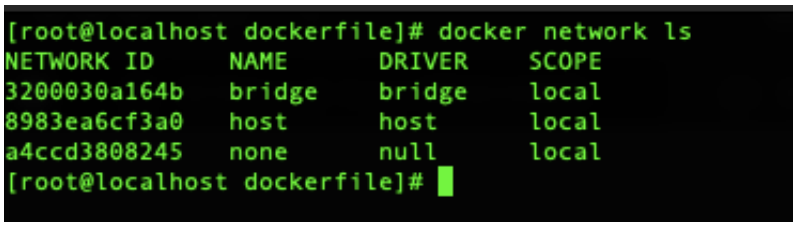
自定义网络

docker0的问题：不支持容器名访问

自定义网络

容器互连

> 查看所有的docker网络



网络模式

bridge：桥接模式 docker（默认）自己创建

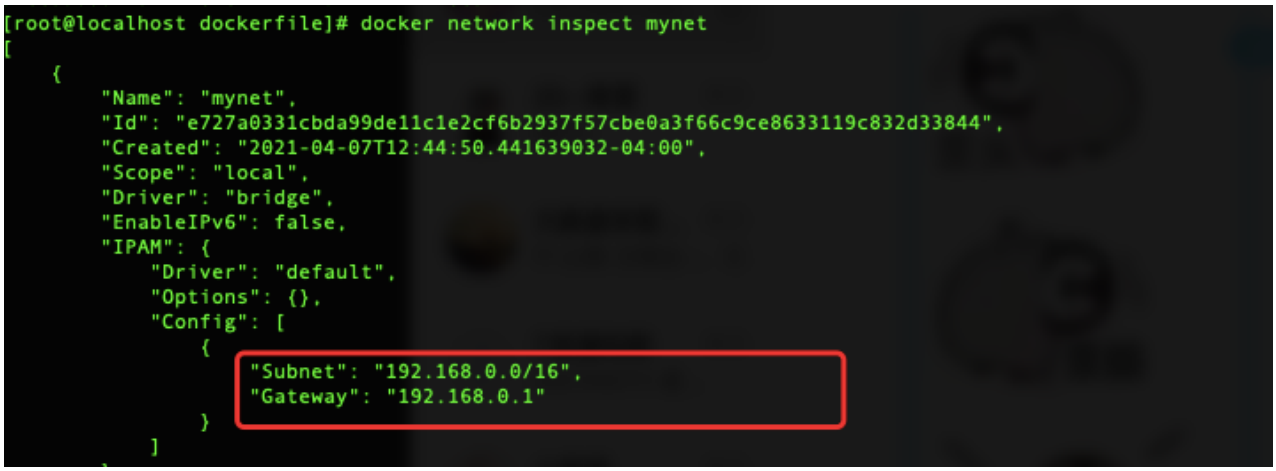
none：不配置网络

host：主机模式，和宿主机共享网络

测试

```
# 我们直接启动的命令 --net bridge，而这个就是我们的docker0
docker run -d -p --name tomcat01 tomcat
docker run -d -p --name tomcat01 --net bridge tomcat

#docker0特点：默认的，域名不能访问，  --link可以打通连接

#自定义一个网络
# --driver bridge
```

```
# --subnet 192.168.0.0/16
# --gatway 192.168.0.1/16
[root@localhost dockerfile]# docker network create --driver bridge --subnet
192.168.0.0/16 --gateway 192.168.0.1 mynet
e727a0331cbda99de11c1e2cf6b2937f57cbe0a3f66c9ce8633119c832d33844
[root@localhost dockerfile]# docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
3200030a164b   bridge    bridge    local
8983ea6cf3a0   host      host      local
e727a0331cbd   mynet     bridge    local
```



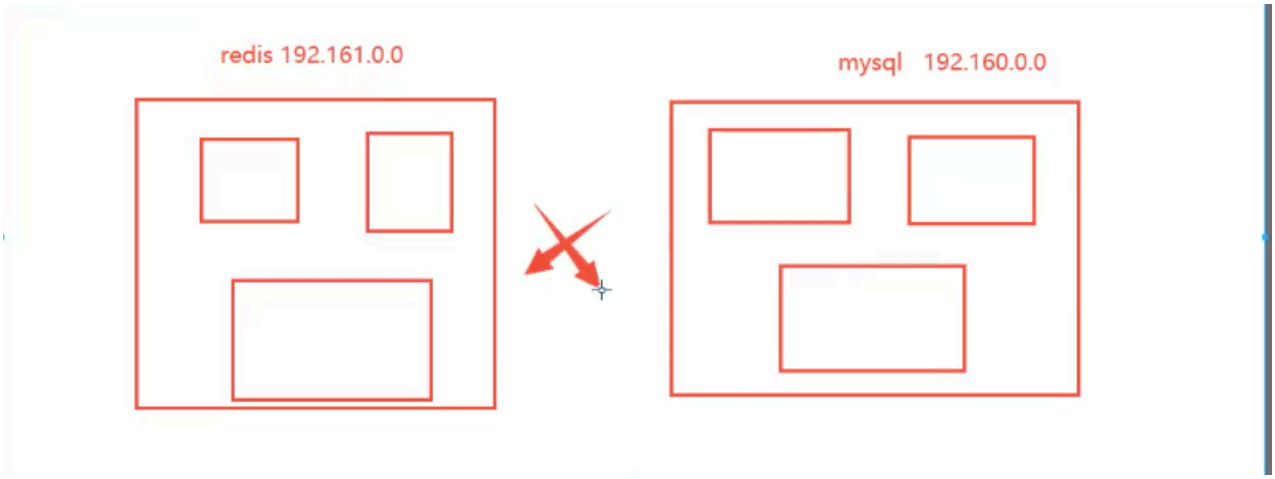

```
 "ConfigOnly": false,
        "Containers": {
            "84823d554f25e0c981c235cdba256de5a83b636671fb65561516258e1f1e3280":
{
                "Name": "tomcat-net-01",
                "EndpointID":
"c8c2d2839c94aee4ffeef6ce3e79e04ced247f08f8591227e1a9578285887cea",
                "MacAddress": "02:42:c0:a8:00:02",
                "IPv4Address": "192.168.0.2/16",
                "IPv6Address": ""
            },
            "95ae06c0919d7b3114cde99d5e24282920e40a995574dd578071d9684d889cd2":
{
                "Name": "tomcat-net-02",
                "EndpointID":
"e327e7d8f65e433e03756c54a136c1b5f3f6a97a1d862d9abd8dcb006aa18ded",
                "MacAddress": "02:42:c0:a8:00:03",
                "IPv4Address": "192.168.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
```

```
]

#再次测试ping连接，现在不使用---link也可以ping名字
[root@localhost dockerfile]# docker exec -it tomcat-net-01 ping tomcat-net-02
PING tomcat-net-02 (192.168.0.3) 56(84) bytes of data.
64 bytes from tomcat-net-02.mynet (192.168.0.3): icmp_seq=1 ttl=64 time=0.304
ms
64 bytes from tomcat-net-02.mynet (192.168.0.3): icmp_seq=2 ttl=64 time=0.130
ms
64 bytes from tomcat-net-02.mynet (192.168.0.3): icmp_seq=3 ttl=64 time=0.130
ms
^C
--- tomcat-net-02 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 8ms
rtt min/avg/max/mdev = 0.130/0.188/0.304/0.082 ms
[root@localhost dockerfile]#
```

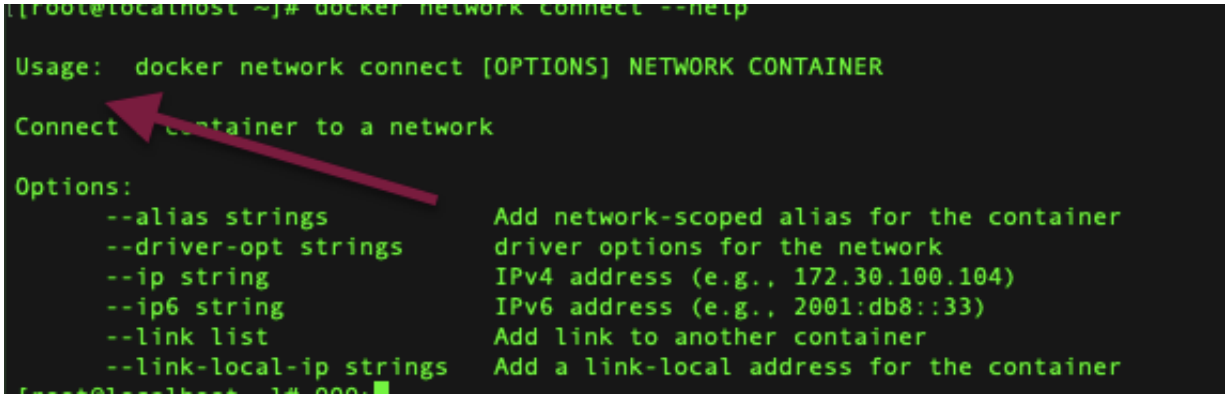
我们自定义的网络docker都已经维护好了对应关系

好处：

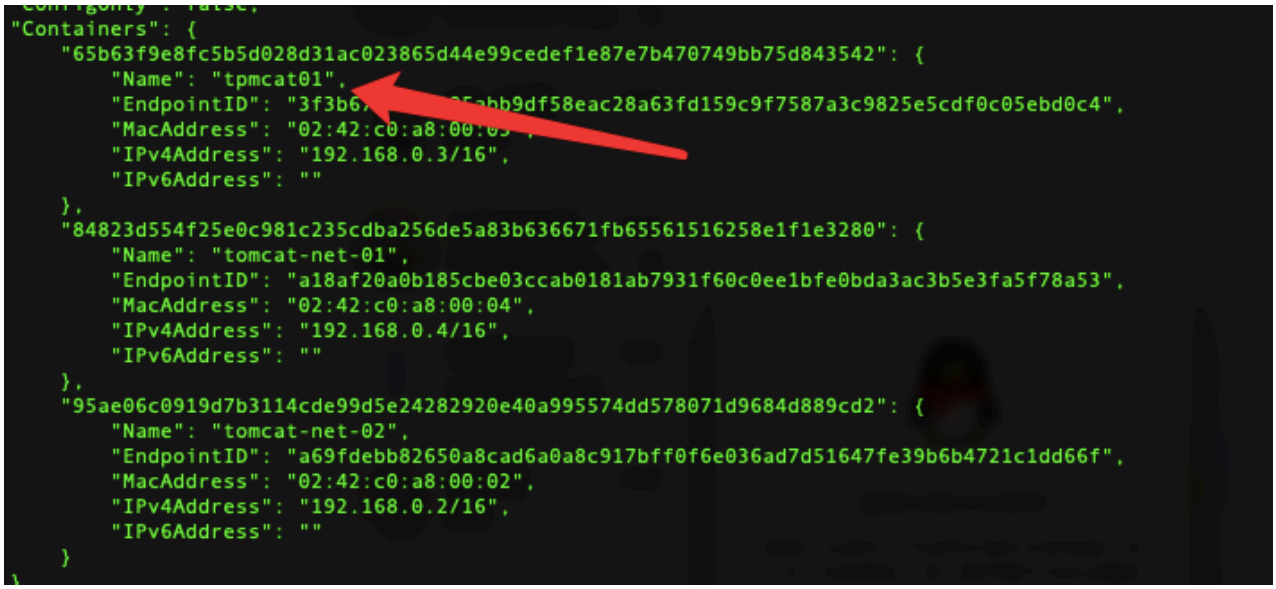
redis：不同的集群使用不同的网络，保证集群最安全



网络连通



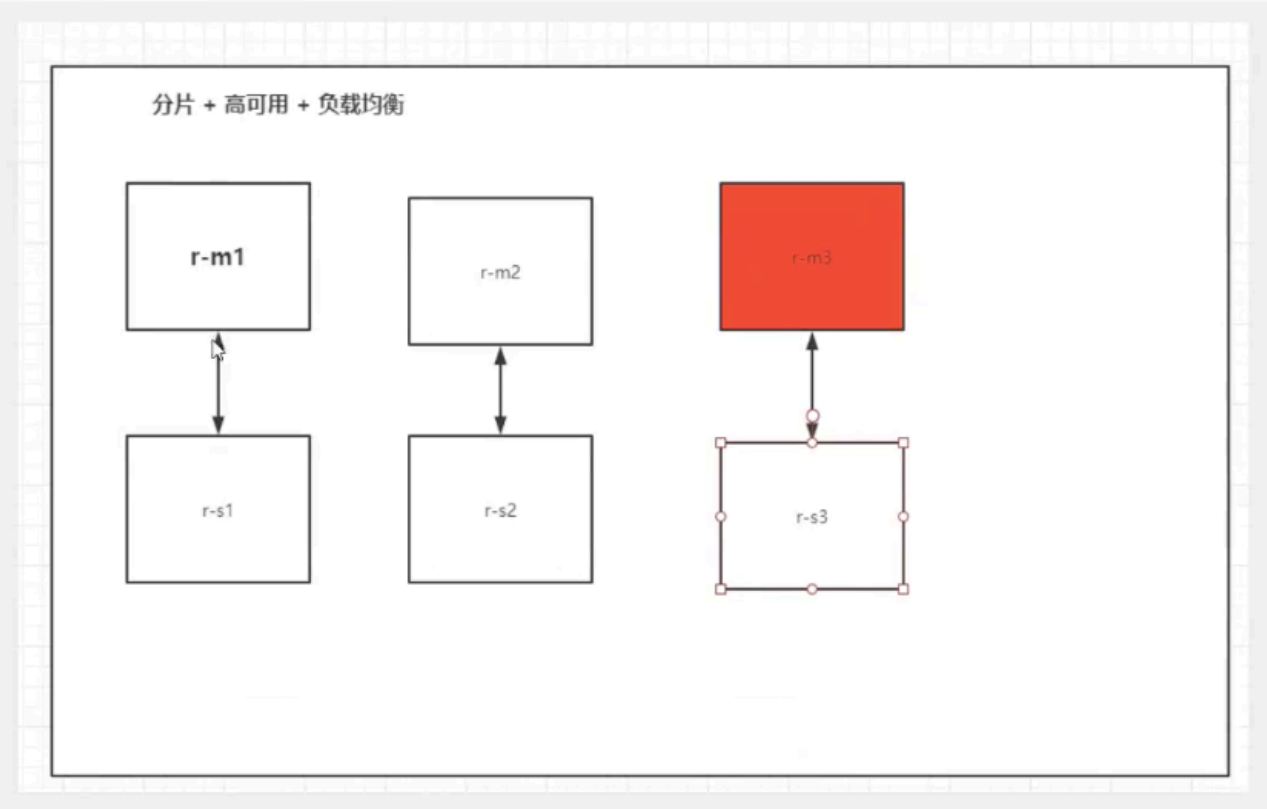
```
# 测试打通tomcat01到mynet
[root@localhost ~]# docker network connect mynet 65b63f9e8fc5

#  连通之后，就是讲tomcat01放到mynet网络下（一个容器两个ip） 阿里云服务器 一个公网ip 一个私网ip
```




```
# tomcqat01能够连通
[root@localhost ~]# docker exec tpmcat01  ping tomcat-net-01
PING tomcat-net-01 (192.168.0.4) 56(84) bytes of data.
64 bytes from tomcat-net-01.mynet (192.168.0.4): icmp_seq=1 ttl=64 time=0.145 ms
64 bytes from tomcat-net-01.mynet (192.168.0.4): icmp_seq=2 ttl=64 time=0.079 ms
64 bytes from tomcat-net-01.mynet (192.168.0.4): icmp_seq=3 ttl=64 time=0.132 ms
```

假设要跨网络操作别人，就需要docker network connect 连通！！

实战:部署redis集群




```
# 创建网卡
docker network create redis --subnet 172.38.0.0/16

# 部署6个redis集群的脚本
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

docker run -p 637${port}:6379 -p 1637${port}:16379 --name redis-${port} -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-serve  /etc/redis/redis.conf
```

```
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 redis-serve /etc/redis/redis.conf
```