

Docker容器学习笔记一（狂神说Java）

狂神说B站视频：<https://www.bilibili.com/video/BV1og4y1q7M4?p=1>

Docker容器学习笔记二（狂神说Java）：https://blog.csdn.net/qq_41822345/article/details/107123141

学习docker：<https://www.runoob.com/docker/docker-tutorial.html>

使用docker：<https://labs.play-with-docker.com/>

一、Docker概述

1.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运行！版本更新，导致服务不可用！对于运维来说考验十分大？

环境配置是十分的麻烦，每一个机器都要部署环境（集群Redis、ES、Hadoop...）！费事费力。

发布一个项目（jar + （Redis MySQL JDK ES）），项目能不能带上环境安装打包！

之前在服务器配置一个应用的环境 Redis MySQL JDK ES Hadoop 配置超麻烦了，不能够跨平台。

开发环境Windows，最后发布到Linux！

传统：开发jar，运维来做！

现在：开发打包部署上线，一套流程做完！

安卓流程：java — apk — 发布（应用商店） — 张三使用apk — 安装即可用！

docker流程：java-jar（环境） — 打包项目带上环境（镜像） — （Docker仓库：商店） -----

Docker给以上的问题，提出了解决方案！

Docker的思想就来自于集装箱！

JRE - 多个应用（端口冲突） - 原来都是交叉的！

隔离：Docker核心思想！打包装箱！每个箱子是互相隔离的。

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了一些问题，我们需要去解决，才去学习！

2.Docker的历史

2010年，几个的年轻人，就在美国成立了一家公司 dotcloud

做一些pass的云计算服务！LXC（Linux Container容器）有关的容器技术！

- Linux Container容器是一种内核虚拟化技术，可以提供轻量级的虚拟化，以便隔离进程和资源。

他们将自己的技术（容器化技术）命名就是 Docker。

Docker刚刚诞生的时候，没有引起行业的注意！dotCloud，就活不下去！

- 开源

2013年，Docker开源！

越来越多的人发现docker的优点！火了。Docker每个月都会更新一个版本！

2014年4月9日，Docker1.0发布！

docker为什么这么火？ 十分的轻巧！

在容器技术出来之前，我们都是使用虚拟机技术！

虚拟机：在window中装一个VMware，通过这个软件我们可以虚拟出来一台或者多台电脑！笨重！

虚拟机也属于虚拟化技术，Docker容器技术，也是一种虚拟化技术！

- vm : linux centos 原生镜像（一个电脑！）隔离、需要开启多个虚拟机！ 几个G 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！ 几个M 秒级启动！

Docker基于Go语言开发的！开源项目！

docker官网：<https://www.docker.com/>

文档：<https://docs.docker.com/> Docker的文档是超级详细的！

仓库：<https://hub.docker.com/>

3.Docker能做什么？

比较Docker和虚拟机技术的不同：

- 传统虚拟机，虚拟出一条硬件，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
- 容器内的应用直接运行在宿主机的内容，容器是没有自己的内核的，也没有虚拟我们的硬件，所以就轻便了
- 每个容器间是互相隔离，每个容器内都有一个属于自己的文件系统，互不影响

4.DevOps (开发、运维)

- 应用更快速的交付和部署

传统：一对帮助文档，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

使用了 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！服务器B

- 更简单的系统运维

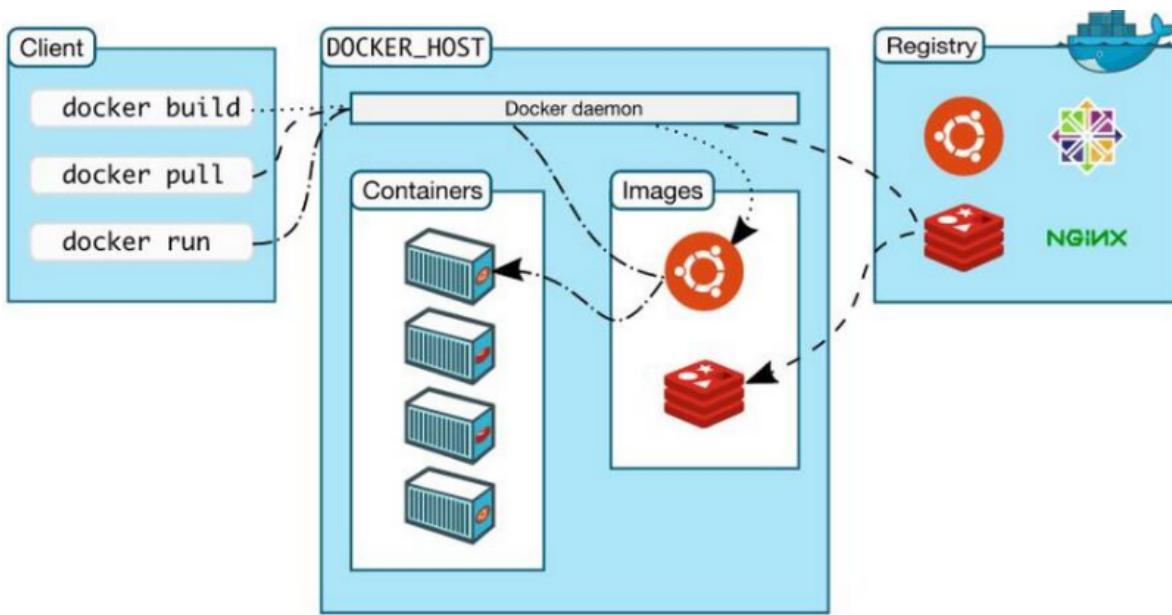
在容器化之后，我们的开发，测试环境都是高度一致的

- 更高效的计算资源利用

Docker是内核级别的虚拟化，可以在一个物理机上可以运行很多的容器实例！服务器的性能可以被压榨到极致。

二、Docker安装

1.Docker的基本组成



- 镜像 (image):

docker镜像就好比是一个目标，可以通过这个目标来创建容器服务，tomcat镜像==>run==>容器（提供服务器），通过这个镜像可以创建多个容器（最终服务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，通过镜像来创建的。

启动，停止，删除，基本命令

目前就可以把这个容器理解为就是一个简易的 Linux系统。

- 仓库(repository):

仓库就是存放镜像的地方！

仓库分为公有仓库和私有仓库。(很类似git)

Docker Hub是国外的。

阿里云...都有容器服务器(配置镜像加速!)

2.安装Docker

- 环境准备

Linux要求内核3.0以上

```

→ ~ uname -r
4.15.0-96-generic # 要求3.0以上
→ ~ cat /etc/os-release
NAME="Ubuntu"
VERSION="18.04.4 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.4 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-
policy"VERSION_CODENAME=bionic
UBUNTU_CODENAME=bionic

```

- 安装

帮助文档: <https://docs.docker.com/engine/install/>

#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
```

#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，不推荐

#推荐使用国内的

```
yum-config-manager \
    --add-repo \
    https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
```

#更新yum软件包索引

```
yum makecache fast
```

#4. 安装docker相关的 docker-ce 社区版 而ee是企业版

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

#6. 使用docker version查看是否按照成功

```
docker version
```

#7. 测试

```
docker run hello-world
```

#7. 测试

```
→ ~ docker run hello-world
```

```
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/
```

#8. 查看一下下载的镜像

→ ~ docker images

REPOSITORY	TAG	IMAGE ID	CREATED
SIZE			
hello-world	latest	bf756fb1ae65	4 months 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

/var/lib/docker 是docker的默认工作路径！

- 阿里云镜像加速

1、登录阿里云找到容器服务

2、找到镜像加速器

The screenshot shows the Alibaba Cloud Container Registry interface. On the left, there's a sidebar with options like 'Default Instance', 'Image Library', 'Naming Space', 'Authorization Management', 'Code Source', 'Access Token', 'Enterprise Edition Instances', 'Image Center', 'Image Search', 'Favorites', and 'Docker Accelerator'. The 'Docker Accelerator' option is highlighted with a red box. The main content area has tabs for 'Image Accelerator' and 'Operation Document'. Under 'Image Accelerator', it says 'Using an accelerator can improve the speed of fetching official Docker image'. It shows a 'Accelerator Address' input field containing 'https://qiyb9988.mirror.aliyuncs.com' with a 'Copy' button. Below that is a 'Operation Document' tab for 'CentOS' (which is selected, indicated by a red box around its tab), with sub-sections for 'Install / Upgrade Docker Client' and 'Configure Image Accelerator'. It also includes a note for users with Docker client versions above 1.10.0. A code block at the bottom shows how to modify the daemon configuration file:

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json << EOF
{
  "registry-mirrors": ["https://qiyb9988.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

3、配置使用

```

sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://qiyb9988.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker

```

- HelloWorld

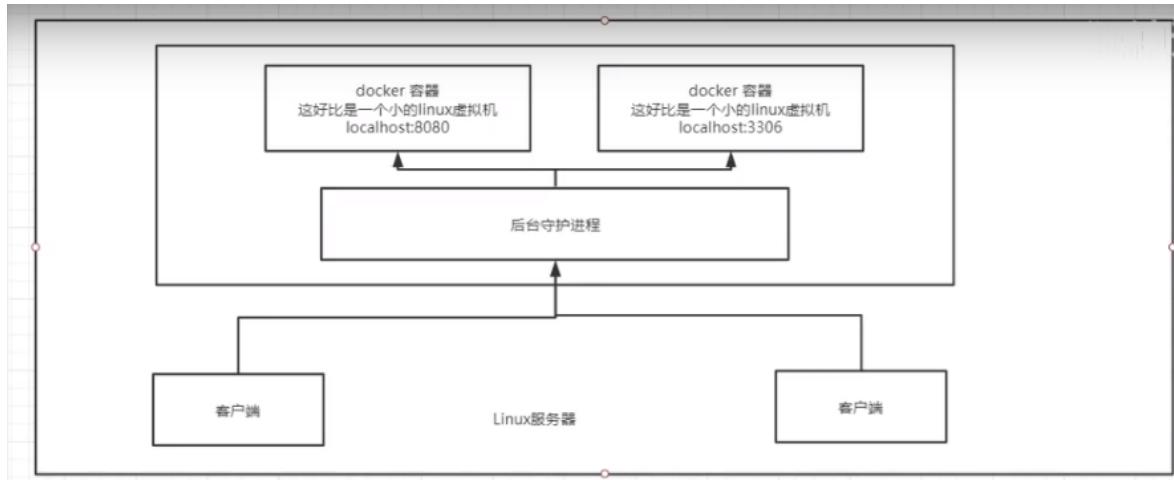
docker run 流程图

3.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，就会执行这个命令！



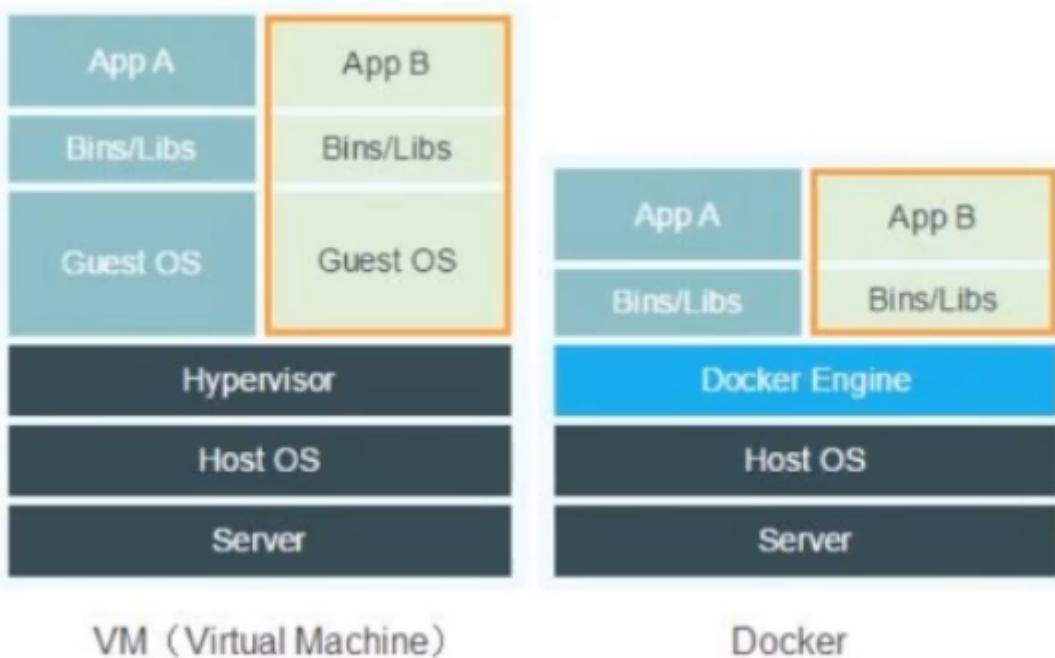
- 为什么Docker比Vm快

1、docker有着比虚拟机更少的抽象层。由于docker不需要Hypervisor实现硬件资源虚拟化,运行在docker容器上的程序直接使用的都是实际物理机的硬件资源。因此在CPU、内存利用率上docker将会在效率上有明显优势。

2、docker利用的是宿主机的内核,而不需要Guest OS。

GuestOS: VM (虚拟机) 里的的系统 (OS) ;

HostOS: 物理机里的系统 (OS) ;



因此，当新建一个容器时，docker不需要和虚拟机一样重新加载一个操作系统内核。然而避免引导、加载操作系统内核是个比较费时费资源的过程，当新建一个虚拟机时，虚拟机软件需要加载GuestOS，这个新建过程是分钟级别的。而docker由于直接利用宿主机的操作系统，则省略了这个复杂的过程，因此新建一个docker容器只需要几秒钟。

三、Docker的常用命令

1.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 #帮助命令
#帮助文档的地址: https://docs.docker.com/engine/reference/commandline/build/
```

2.镜像命令

```
docker images #查看所有本地主机上的镜像 可以使用docker image ls代替
docker search 搜索镜像
docker pull 下载镜像 docker image pull
docker rmi 删除镜像 docker image rm
```

docker images 查看所有本地的主机上的镜像

```
→ ~ docker images
REPOSITORY          TAG      IMAGE ID      CREATED
SIZE
mysql              5.7      e73346bdf465   24 hours ago
448MB
# 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           Show all images (default hides intermediate images) #列出所有镜像
-q, --quiet        Only show numeric IDs # 只显示镜像的id

→ ~ docker images -aq # 显示所有镜像的id
e73346bdf465
d03312117bb0
d03312117bb0
602e111c06b6
2869fc110bf7
470671670cac
bf756fb1ae65
5acf0e8da90b
```

docker pull 下载镜像

```
# 下载镜像 docker pull 镜像名[:tag]
→ ~ docker pull tomcat:8
8: Pulling from library/tomcat #如果不写tag, 默认就是latest
90fe46dd8199: Already exists #分层下载: docker image 的核心 联合文件系统
35a4f1977689: Already exists
bbc37f14aded: Already exists
74e27dc593d4: Already exists
93a01fbfad7f: Already exists
1478df405869: Pull complete
64f0dd11682b: Pull complete
68ff4e050d11: Pull complete
f576086003cf: Pull complete
3b72593ce10e: Pull complete
Digest: sha256:0c6234e7ec9d10ab32c06423ab829b32e3183ba5bf2620ee66de866df640a027
# 签名 防伪
Status: Downloaded newer image for tomcat:8
docker.io/library/tomcat:8 #真实地址

#等价于
docker pull tomcat:8
docker pull docker.io/library/tomcat:8
```

docker rmi 删除镜像

```
→ ~ 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 镜像id#删除指定的镜像
→ ~ docker rmi -f $(docker images -aq) #删除全部的镜像
```

3.容器命令

docker run	镜像id 新建容器并启动
docker ps	列出所有运行的容器 docker container list
docker rm	容器id 删除指定容器
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

说明：我们有了镜像才可以创建容器，Linux，下载centos镜像来学习

```

→ ~ docker container
Usage: docker container COMMAND
Manage containers
Commands:
  attach      Attach local standard input, output, and error streams to a
running container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's
filesystem
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  inspect    Display detailed information on one or more containers
  kill        Kill one or more running containers
  logs       Fetch the logs of a container
  ls          List containers
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  prune      Remove all stopped containers
  rename     Rename a container
  restart    Restart one or more containers
  rm          Remove one or more containers
  run         Run a command in a new container
  start      Start one or more stopped containers
  stats      Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  top         Display the running processes of a container
  unpause    Unpause all processes within one or more containers
  update     Update configuration of one or more containers
  wait       Block until one or more containers stop, then print their exit
codes
Run 'docker container COMMAND --help' for more information on a command.

```

新建容器并启动

```

docker run [可选参数] image | docker container run [可选参数] image
#参考说明
--name="Name"      容器名字 tomcat01 tomcat02 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(宿主机):8080(容器)
                  -p ip:主机端口:容器端口
                  -p 主机端口:容器端口(常用)
                  -p 容器端口
                  容器端口
-P(大写)           随机指定端口
# 测试、启动并进入容器
→ ~ docker run -it centos /bin/bash
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
8a29a15cefae: Already exists
Digest: sha256:fe8d824220415eed5477b63addf40fb06c3b049404242b31982106ac204f6700
Status: Downloaded newer image for centos:latest
[root@95039813da8d /]# ls

```

```

bin dev etc home lib lib64 lost+found media mnt opt proc root run
sbin srv sys tmp usr var
[root@95039813da8d /]# exit #从容器退回主机
exit
→ ~ ls
shell user.txt

```

列出所有运行的容器

```

#docker ps命令 #列出当前正在运行的容器
-a, --all           Show all containers (default shows just running)
-n, --last int      Show n last created containers (includes all states)
(default -1)
-q, --quiet          Only display numeric IDs

→ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
          STATUS              PORTS            NAMES
68729e9654d4        portainer/portainer   "/portainer"
          Up About a minute  0.0.0.0:8088->9000/tcp    funny_curie
d506a017e951         nginx              "nginx -g 'daemon of..."
          Up 15 hours       0.0.0.0:3344->80/tcp    nginx01

→ ~ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
          STATUS              PORTS            NAMES
95039813da8d        centos              "/bin/bash"
          Exited (0) 2 minutes ago
1e46a426a5ba        tomcat              "catalina.sh run"
ago             Exited (130) 9 minutes ago
14bc9334d1b2        bf756fb1ae65     "/hello"
          Exited (0) 3 hours ago
f10d60f473f5        bf756fb1ae65     "/hello"
          Exited (0) 3 hours ago
68729e9654d4        portainer/portainer   "/portainer"
          Up About a minute  0.0.0.0:8088->9000/tcp    funny_curie
677cde5e4f1d        elasticsearch       "/docker-entrypoint..."
          Exited (143) 8 minutes ago
33eb3f70b4db        tomcat              "catalina.sh run"
          Exited (143) 8 minutes ago
d506a017e951         nginx              "nginx -g 'daemon of..."
          Up 15 hours       0.0.0.0:3344->80/tcp    nginx01
24ce2db02e45        centos              "/bin/bash"
          Exited (0) 15 hours ago
42267d1ad80b        bf756fb1ae65     "/hello"
          Exited (0) 16 hours ago

→ ~ docker ps -aq
95039813da8d
1e46a426a5ba
14bc9334d1b2
f10d60f473f5
68729e9654d4
677cde5e4f1d
33eb3f70b4db
d506a017e951
24ce2db02e45
42267d1ad80b

```

退出容器

```
exit #容器直接退出  
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，如果要强制删除 rm -rf  
docker rm -f $(docker ps -aq) #删除指定的容器  
docker ps -a -q|xargs docker rm #删除所有的容器
```

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
```

4.常用其他命令

后台启动命令

```
# 命令 docker run -d 镜像名  
→ ~ docker run -d centos  
a8f922c255859622ac45ce3a535b7a0e8253329be4756ed6e32265d2dd2fac6c  
→ ~ docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES  
STATUS              PORTS              NAMES  
# 问题docker ps. 发现centos 停止了  
# 常见的坑，docker容器使用后台运行，就必须要有要一个前台进程，docker发现没有应用，就会自动停止  
# nginx，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
```

查看日志

```
docker logs --help  
Options:  
      --details          Show extra details provided to logs  
      *  -f, --follow       Follow log output  
      --since string     Show logs since timestamp (e.g. 2013-01-02T13:23:37) or  
relative (e.g. 42m for 42 minutes)  
      *  --tail string    Number of lines to show from the end of the logs  
(default "all")  
      *  -t, --timestamps   Show timestamps  
      --until string     Show logs before a timestamp (e.g. 2013-01-02T13:23:37)  
or relative (e.g. 42m for 42 minutes)  
→ ~ docker run -d centos /bin/sh -c "while true;do echo 6666;sleep 1;done" #模拟日志  
#显示日志  
-tf      #显示日志信息（一直更新）  
--tail number #需要显示日志条数  
docker logs -t --tail n 容器id #查看n行日志  
docker logs -ft 容器id #跟着日志
```

查看容器中进程信息 ps

```
docker top 容器id
```

查看镜像的元数据

```
# 命令
docker inspect 容器id
#测试
→ ~ docker inspect 55321bcae33d
[
  {
    "Id": "55321bcae33d15da8280bcac1d2bc1141d213bcc8f8e792edfd832ff61ae5066",
    "Created": "2020-05-15T05:22:05.515909071Z",
    "Path": "/bin/sh",
    ...
  }
]
→ ~
```

进入当前正在运行的容器

```
# 我们通常容器都是使用后台方式运行的，需要进入容器，修改一些配置
# 命令
docker exec -it 容器id bashshell
#测试
→ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
55321bcae33d        centos              "/bin/sh -c 'while t..."   10 minutes ago   Up 10 minutes      bold_bell
a7215824a4db        centos              "/bin/sh -c 'while t..."   13 minutes ago   Up 13 minutes      zen_kepler
55a31b3f8613        centos              "/bin/bash"          15 minutes ago   Up 15 minutes      lucid_clarke
→ ~ docker exec -it 55321bcae33d /bin/bash
[root@55321bcae33d /]#
```

```
# 方式二
docker attach 容器id
#测试
docker attach 55321bcae33d
正在执行当前的代码...
区别
#docker exec #进入当前容器后开启一个新的终端，可以在里面操作。（常用）
#docker attach # 进入容器正在执行的终端
```

从容器内拷贝到主机上

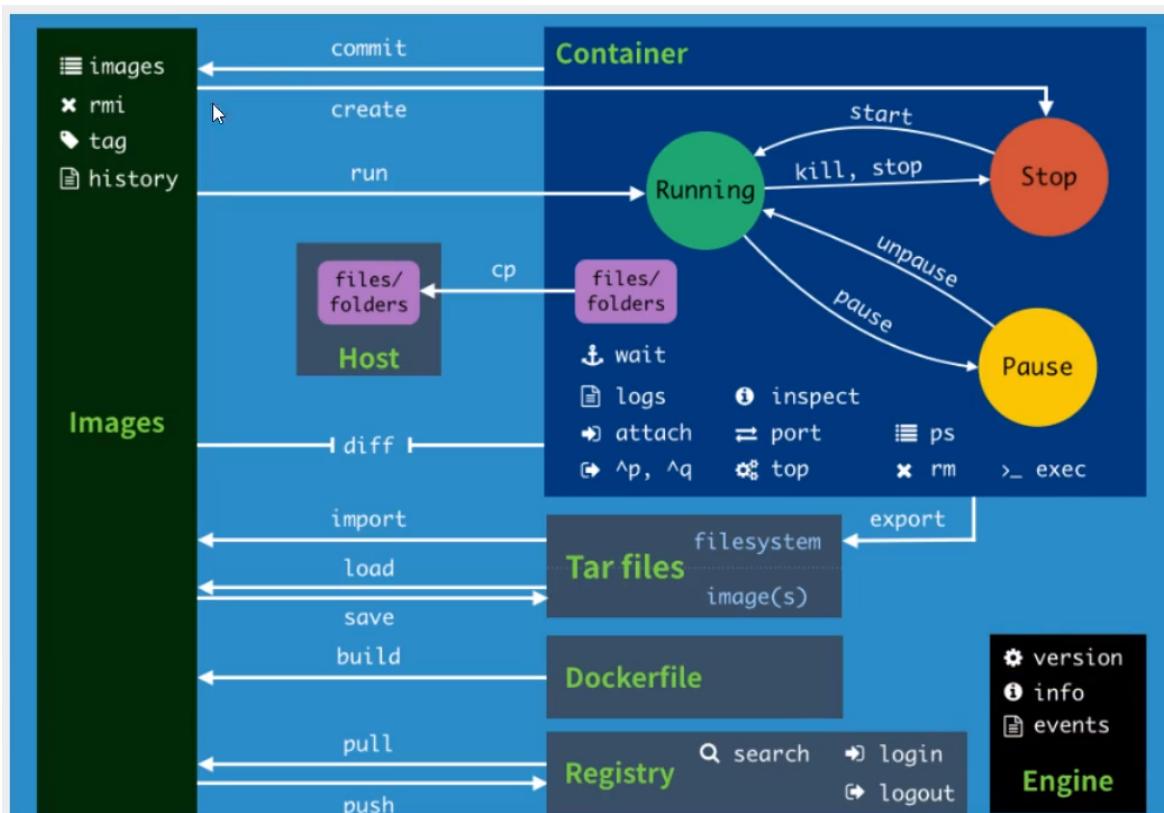
```
docker cp 容器id:容器内路径 主机目的路径
#进入docker容器内部
→ ~ docker exec -it 55321bcae33d /bin/bash
[root@55321bcae33d /]# ls
bin  etc  lib  lost+found  mnt  proc  run  srv  tmp  var
dev  home  lib64  media  opt  root  sbin  sys  usr
```

```
#新建一个文件
[root@55321bcae33d /]# echo "hello" > java.java
[root@55321bcae33d /]# cat java.java
hello
[root@55321bcae33d /]# exit
exit
→ ~ docker cp 55321bcae33d:/java.java /      #拷贝
→ ~ cd /
→ / ls #可以看见java.java存在
bin  home   lib    mnt   run    sys  vmlinuz
boot initrd.img lib64  opt   sbin   tmp  vmlinuz.old
dev  initrd.img.old lost+found proc   srv    usr  wget-log
etc  java.java   media  root   swapfile var
```

学习方式：将所有笔记敲一遍，自己记录笔记！

四、小结

Docker的所有命令



Docker命令帮助文档 (重要)

```
attach      Attach local standard input, output, and error streams to a
running container
#当前shell下 attach连接指定运行的镜像
build       Build an image from a Dockerfile # 通过Dockerfile定制镜像
commit      Create a new image from a container's changes #提交当前容器为新的镜像
cp          Copy files/folders between a container and the local filesystem #
拷贝文件
create      Create a new container #创建一个新的容器
diff        Inspect changes to files or directories on a container's
filesystem #查看docker容器的变化
events      Get real time events from the server # 从服务获取容器实时时间
exec        Run a command in a running container # 在运行中的容器上运行命令
```

```

export      Export a container's filesystem as a tar archive #导出容器文件系统作为一个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 #列出系统当前的镜像
import     Import the contents from a tarball to create a filesystem image #
从tar包中导入内容创建一个文件系统镜像
info       Display system-wide information # 显示全系统信息
inspect   Return low-level information on Docker objects #查看容器详细信息
kill      Kill one or more running containers # kill指定docker容器
load      Load an image from a tar archive or STDIN #从一个tar包或标准输入中加载一个镜像[对应save]
login     Log in to a Docker registry #
logout   Log out from a Docker registry
logs      Fetch the logs of a container
pause    Pause all processes within one or more containers
port     List port mappings or a specific mapping for the container
ps        List containers
pull     Pull an image or a repository from a registry
push     Push an image or a repository to a registry
rename   Rename a container
restart  Restart one or more containers
rm       Remove one or more containers
rmi     Remove one or more images
run      Run a command in a new container
save     Save one or more images to a tar archive (streamed to STDOUT by default)
search   Search the Docker Hub for images
start    Start one or more stopped containers
stats    Display a live stream of container(s) resource usage statistics
stop     Stop one or more running containers
tag      Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
top      Display the running processes of a container
unpause  Unpause all processes within one or more containers
update   Update configuration of one or more containers
version  Show the Docker version information
wait    Block until one or more containers stop, then print their exit codes

```

作业练习

三个作业：作业1告诉我们暴露端口的重要性；作业2告诉我们进入容器的重要性；作业3告诉我们查看当前容器状态的重要性，如何修改容器运行的环境。

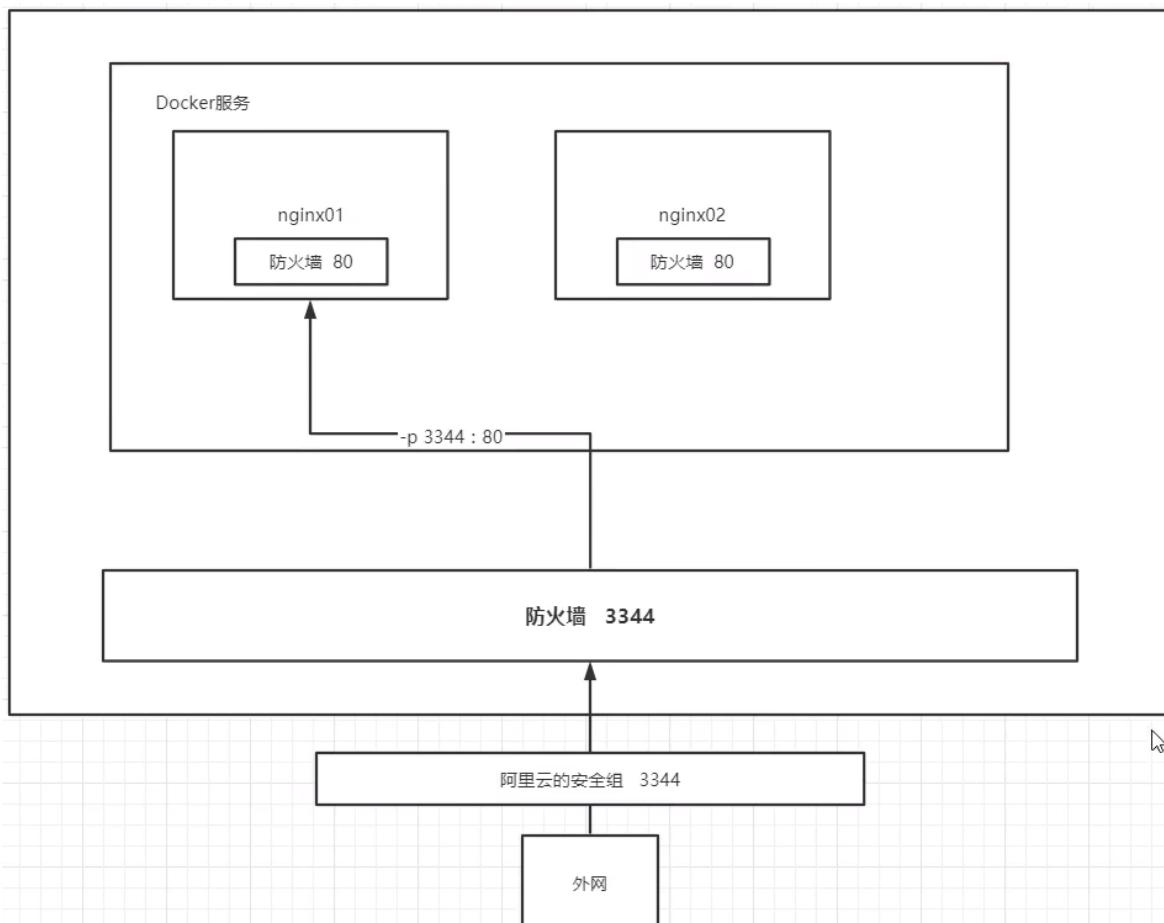
作业1：Docker 安装Nginx

```

#1. 搜索镜像 search 建议大家去docker搜索，可以看到帮助文档
#2. 拉取镜像 pull
#3. 运行测试
# -d 后台运行
# --name 给容器命名
# -p 宿主机端口: 容器内部端口
→ ~ docker run -d --name nginx00 -p 3344:80 nginx
75943663c116f5ed006a0042c42f78e9a1a6a52eba66311666eee12e1c8a4502
→ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
75943663c116        nginx              "nginx -g 'daemon of..."   41 seconds ago   Up 40 seconds   nginx00
→ ~ curl localhost:3344    #测试
<!DOCTYPE html>,,,

```

端口暴露示意图：



思考问题：我们每次改动nginx配置文件，都需要进入容器内部？十分的麻烦，要是可以在容器外部提供一个映射路径，达到在容器修改文件名，容器内部就可以自动修改？→ 数据卷！

作业2：docker 来装一个tomcat

```

# 官方的使用
docker run -it --rm tomcat:9.0
# 之前的启动都是后台，停止了容器，容器还是可以查到
# docker run -it --rm image 一般是用来测试，用完就删除（暂时不建议）
--rm      Automatically remove the container when it exits
# 下载
docker pull tomcat
# 启动运行

```

```

docker run -d -p 8080:8080 --name tomcat01 tomcat
#测试访问有没有问题
curl localhost:8080

#进入容器
→ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
STATUS              PORTS              NAMES
db09851cf82e       tomcat              "catalina.sh run"   28 seconds ago
Up 27 seconds       0.0.0.0:8080->8080/tcp   tomcat01
→ ~ docker exec -it db09851cf82e /bin/bash
root@db09851cf82e:/usr/local/tomcat#
# 发现问题：1、linux命令少了。 2.没有webapps
# 阿里云镜像（阉割版），它为保证最小镜像，将不必要的都剔除了→保证最小可运行环境！

```

思考问题：我们以后要部署项目，如果每次都要进入容器是不是十分麻烦？要是可以在容器外部提供一个映射路径， webapps，我们在外部放置项目，就自动同步内部就好了！

作业3：部署es+kibana

```

# es 暴露的端口很多！
# es 十分耗内存
# es 的数据一般需要放置到安全目录！挂载
# --net somenetwork ? 网络配置

# 下载启动elasticsearch (Docker一步搞定)
docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e
"discovery.type=single-node" elasticsearch:7.6.2

# 测试一下es是否成功启动
→ ~ curl localhost:9200
{
  "name" : "d73ad2f22dd3",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "atFKgANxS8CzgIyCB8PGxA",
  "version" : {
    "number" : "7.6.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "ef48eb35cf30adf4db14086e8ab07ef6fb113f",
    "build_date" : "2020-03-26T06:34:37.794943Z",
    "build_snapshot" : false,
    "lucene_version" : "8.4.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
# 查看docker容器使用内存情况（每秒刷新，也挺耗内存的一个命令）
→ ~ docker stats

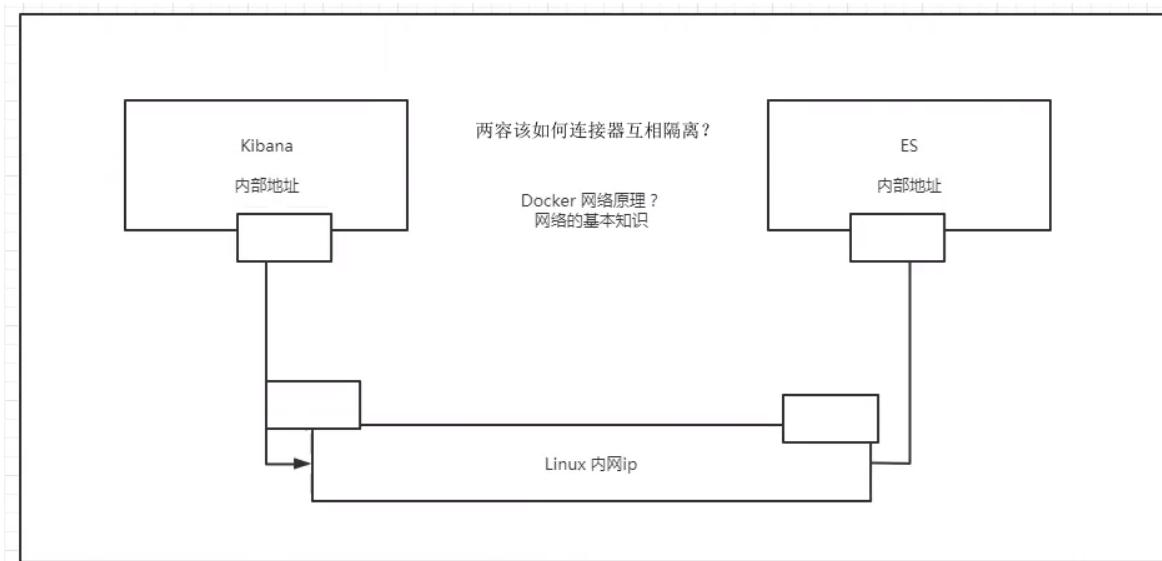
```

```
#关闭，添加内存的限制，修改配置文件 -e 环境配置修改
→ ~ docker rm -f d73ad2f22dd3

→ ~ 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
```

```
→ ~ curl localhost:9200
{
  "name" : "b72c9847ec48",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "yNAK0EORSvq3Wtaqe2QqAg",
  "version" : {
    "number" : "7.6.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "ef48eb35cf30adf4db14086e8abd07ef6fb113f",
    "build_date" : "2020-03-26T06:34:37.794943Z",
    "build_snapshot" : false,
    "lucene_version" : "8.4.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

作业：使用kibana连接es？思考网络如何才能连接。



Docker可视化

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！

```
# 运行如下命令即可 打开可视化服务
docker run -d -p 8080:9000 \
--restart=always -v /var/run/docker.sock:/var/run/docker.sock --privileged=true
portainer/portainer
```

访问: <http://ip:8080/>

五、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，他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**。

将所有的应用和环境，直接打包为docker镜像，就可以直接运行。

1.Docker镜像加载原理

- UnionFs (联合文件系统)

我们下载的时候看到一层层的下载就是这个。

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修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下 (`unite several directories into a single virtual filesystem`)。Union文件系统是 Docker镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。

****特性**:** 一次同时加载多个文件系统，但从外面看起来，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，这样最终的文件系统会包含所有底层的文件和目录。

- Docker镜像加载原理

docker的镜像实际上由一层一层的文件系统组成，这种层级的文件系统UnionFS。

`boots`(boot file system) 主要包含 `bootloader`和 `Kernel`，`bootloader`主要是引导加 `kernel`，Linux刚启动时会加`bootfs`文件系统，在 Docker镜像的最底层是 `boots`。这一层与我们典型的 Linux/Unix系统是一样的，包含`boot`加载器和内核。当`boot`加载完成之后整个内核就都在内存中了，此时内存的使用权已由 `bootfs`转交给内核，此时系统也会卸载`bootfs`。

`rootfs` (root file system)，在 `bootfs`之上。包含的就是典型 Linux系统中的`/dev`,`/proc`,`/bin`,`/etc`等标准目录和文件。`rootfs`就是各种不同的操作系统发行版，比如 `Ubuntu`, `Centos`等等。

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

对于个精简的OS，`rootfs`可以很小，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`rootfs`就可以了。由此可见对于不同的Linux发行版，`boots`基本是一致的，`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，容器是秒级！

2.分层理解

我们可以去下载一个镜像，注意观察下载的日志输出，可以看到是一层层的在下载。

```
→ / docker pull redis
Using default tag: latest
latest: Pulling from library/redis
54fec2fa59d0: Already exists
9c94e11103d9: Pull complete
04ab1bfc453f: Pull complete
a22fde870392: Pull complete
def16cac9f02: Pull complete
1604f5999542: Pull complete
Digest: sha256:399a9b17b8522e24fbe2fd3b42474d4bb668d3994153c4b5d38c3dafd5903e32
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的结构呢？

最大的好处，我觉得莫过于资源共享了！比如有多个镜像都从相同的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base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被共享。

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

```
→ / docker image inspect redis
[
  {
    "Id": "sha256:f9b9909726890b00d2098081642edf32e5211b7ab53563929a47f250bcd1d7c",
    "RepoTags": [
      "redis:latest"
    ],
    "RepoDigests": [
      "redis@sha256:399a9b17b8522e24fbe2fd3b42474d4bb668d3994153c4b5d38c3dafd5903e32"
    ],
    "Parent": "",
    "Comment": "",
    "Created": "2020-05-02T01:40:19.112130797Z",
    "Container": "d30c0bcea88561bc5139821227d2199bb027eeba9083f90c701891b4affce3bc",
    "ContainerConfig": {
      "Hostname": "d30c0bcea885",
      "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=6.0.1",
        "REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-6.0.1.tar.gz"
      ]
    }
  }
]
```

```
"REDIS_DOWNLOAD_SHA=b8756e430479edc162ba9c44dc89ac394316cd482f2dc6b91bcd5fe1259
3f273"
],
"Cmd": [
    "/bin/sh",
    "-c",
    "#(nop) ",
    "CMD [\"redis-server\"]"
],
"ArgsEscaped": true,
"Image":
"sha256:704c602fa36f41a6d2d08e49bd2319ccd6915418f545c838416318b3c29811e0",
"Volumes": {
    "/data": {}
},
"WorkingDir": "/data",
"Entrypoint": [
    "docker-entrypoint.sh"
],
"OnBuild": null,
"Labels": {}
},
"DockerVersion": "18.09.7",
"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=6.0.1",
        "REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-
6.0.1.tar.gz",
        "REDIS_DOWNLOAD_SHA=b8756e430479edc162ba9c44dc89ac394316cd482f2dc6b91bcd5fe1259
3f273"
    ],
    "Cmd": [
        "redis-server"
    ],
    "ArgsEscaped": true,
    "Image":
"sha256:704c602fa36f41a6d2d08e49bd2319ccd6915418f545c838416318b3c29811e0",
    "Volumes": {
        "/data": {}
    }
},
```

```

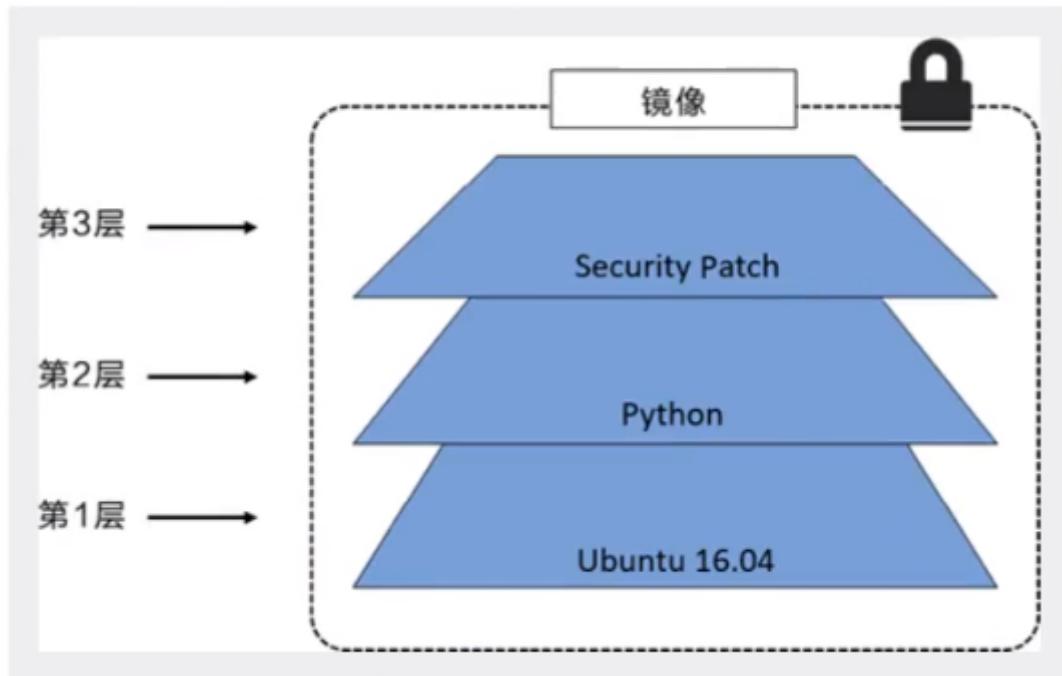
        "WorkingDir": "/data",
        "Entrypoint": [
            "docker-entrypoint.sh"
        ],
        "OnBuild": null,
        "Labels": null
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Size": 104101893,
    "VirtualSize": 104101893,
    "GraphDriver": {
        "Data": {
            "LowerDir":
"/var/lib/docker/overlay2/adea96bbe6518657dc2d4c6331a807eea70567144abda686588ef6
c3bb0d778a/diff:/var/lib/docker/overlay2/66abd822d34dc644e6bebe73721dfd1dc497c2
c8063c43ffb8cf8140e2caeb6/diff:/var/lib/docker/overlay2/d19d24fb6a24801c5fa639c1
d979d19f3f17196b3c6dde96d3b69cd2ad07ba8a/diff:/var/lib/docker/overlay2/a1e95aae5
e09ca6df4f71b542c86c677b884f5280c1d3e3a1111b13644b221f9/diff:/var/lib/docker/ove
rlay2/cd90f7a9cd0227c1db29ea992e889e4e6af057d9ab2835dd18a67a019c18bab4/diff",
            "MergedDir":
"/var/lib/docker/overlay2/afa1de233453b60686a3847854624ef191d7bc317fb01e015b4f06
671139fb11/merged",
            "UpperDir":
"/var/lib/docker/overlay2/afa1de233453b60686a3847854624ef191d7bc317fb01e015b4f06
671139fb11/diff",
            "WorkDir":
"/var/lib/docker/overlay2/afa1de233453b60686a3847854624ef191d7bc317fb01e015b4f06
671139fb11/work"
        },
        "Name": "overlay2"
    },
    "RootFS": {
        "Type": "layers",
        "Layers": [
"sha256:c2adabaecedbda0af72b153c6499a0555f3a769d52370469d8f6bd6328af9b13",
"sha256:744315296a49be711c312dfa1b3a80516116f78c437367ff0bc678da1123e990",
"sha256:379ef5d5cb402a5538413d7285b21aa58a560882d15f1f553f7868dc4b66afa8",
"sha256:d00fd460effb7b066760f97447c071492d471c5176d05b8af1751806a1f905f8",
"sha256:4d0c196331523cfed7bf5bafd616ecb3855256838d850b6f3d5fba911f6c4123",
"sha256:98b4a6242af2536383425ba2d6de033a510e049d9ca07ff501b95052da76e894"
        ]
    },
    "Metadata": {
        "LastTagTime": "0001-01-01T00:00:00Z"
    }
}
]

```

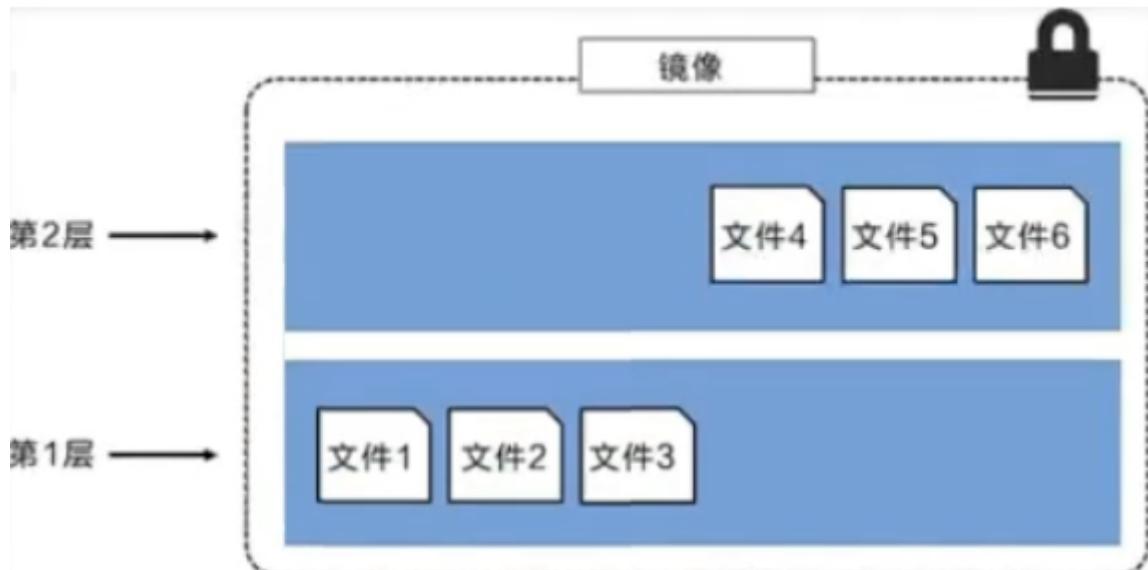
理解：

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

举一个简单的例子，假如基于 Ubuntu Linux16.04 创建一个新的镜像，这就是新镜像的第一层；如果在该镜像中添加 Python 包，就会在基础镜像层之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（这只是一个用于演示的很简单的例子）。

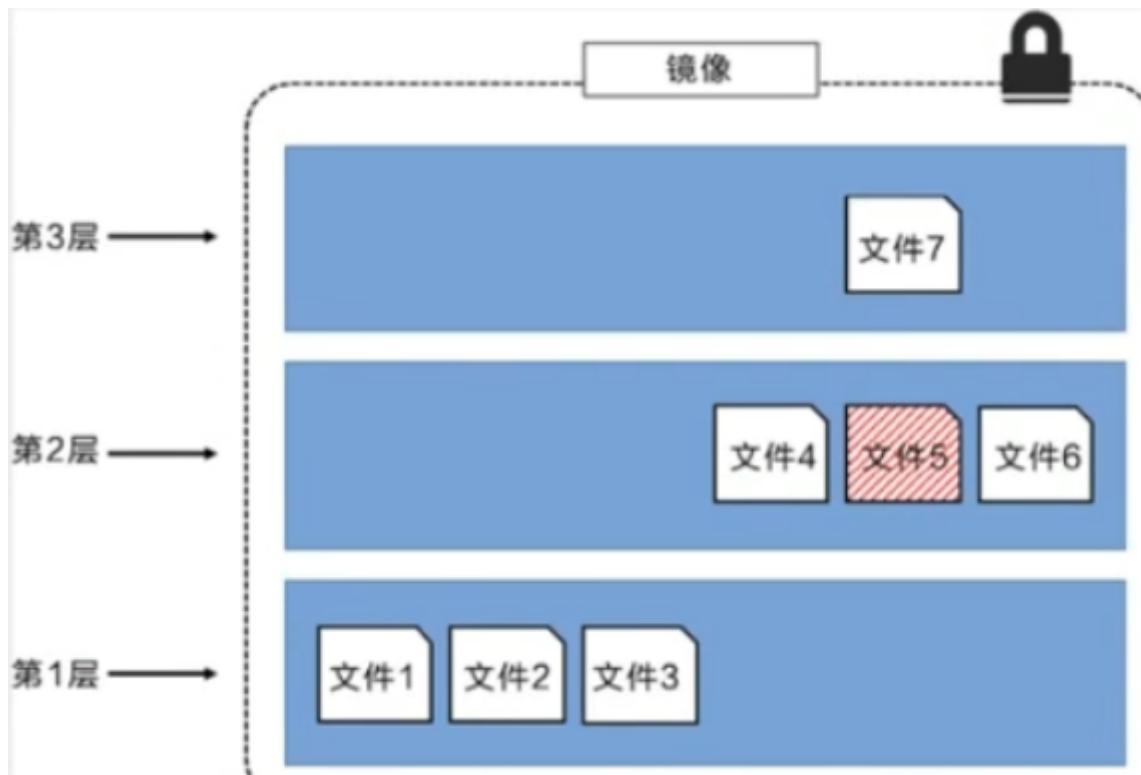


在添加额外的镜像层的同时，镜像始终保持是当前所有镜像的组合，理解这一点非常重要。下图中举了一个简单的例子，每个镜像层包含3个文件，而镜像包含了来自两个镜像层的6个文件。



上图中的镜像层跟之前图中的略有区别，主要目的是便于展示文件

下图中展示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，这是因为最上层中的文件7是文件5的一个更新版



在这种情况下，上层镜像层中的文件覆盖了底层镜像层中的文件。这样就使得文件的更新版本作为一个新镜像层添加到镜像当中。

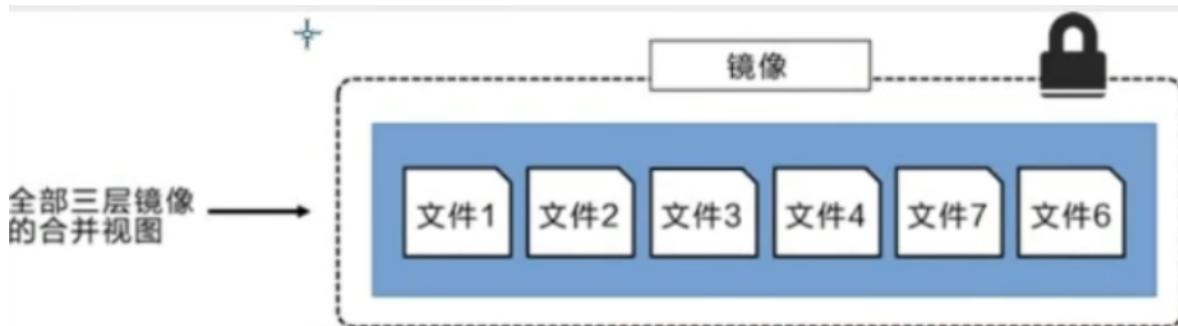
Docker通过存储引擎（新版本采用快照机制）的方式来实现镜像层堆栈，并保证多镜像层对外展示为统一的文件系统。

Linux上可用的存储引擎有AUFS、Overlay2、Device Mapper、Btrfs以及ZFS。顾名思义，每种存储引擎都基于Linux中对应的。

文件系统或者块设备技术，并且每种存储引擎都有其独有的性能特点。

Docker在Windows上仅支持windowsfilter一种存储引擎，该引擎基于NTFS文件系统之上实现了分层和CoW [1]。

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



Docker镜像都是只读的，当容器启动时，一个新的可写层加载到镜像的顶部！

这一层就是我们通常说的容器层，容器之下的都叫镜像层！

3.commit镜像

```
docker commit 提交容器成为一个新的副本
```

```
# 命令和git原理类似
docker commit -m="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tomcat
docker run -d -p 8080:8080 tomcat
# 2、发现这个默认的tomcat 是没有webapps应用，官方的镜像默认webapps下面是没有文件的！
docker exec -it 容器id
# 3、拷贝文件进去

# 4、将操作过的容器通过commit调教为一个镜像！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修改的镜像。
docker commit -m="描述信息" -a="作者" 容器id 目标镜像名:[TAG]
docker commit -a="kuangshen" -m="add webapps app" 容器id tomcat02:1.0
```

如果你想要保存当前容器的状态，就可以通过commit来提交，获得一个镜像，就好比我们使用虚拟机的快照。

入门成功！！！！

下篇：Docker容器学习笔记二（狂神说Java）

容器数据卷

DockerFile

IDEA整合Docker

Docker网络讲解

Docker容器学习笔记二（狂神说Java）

狂神说B站视频：<https://www.bilibili.com/video/BV1og4y1q7M4?p=21>

Docker容器学习笔记一（狂神说Java）：https://blog.csdn.net/qq_41822345/article/details/107123094

六、容器数据卷

1.什么是容器卷？

docker的理念回顾

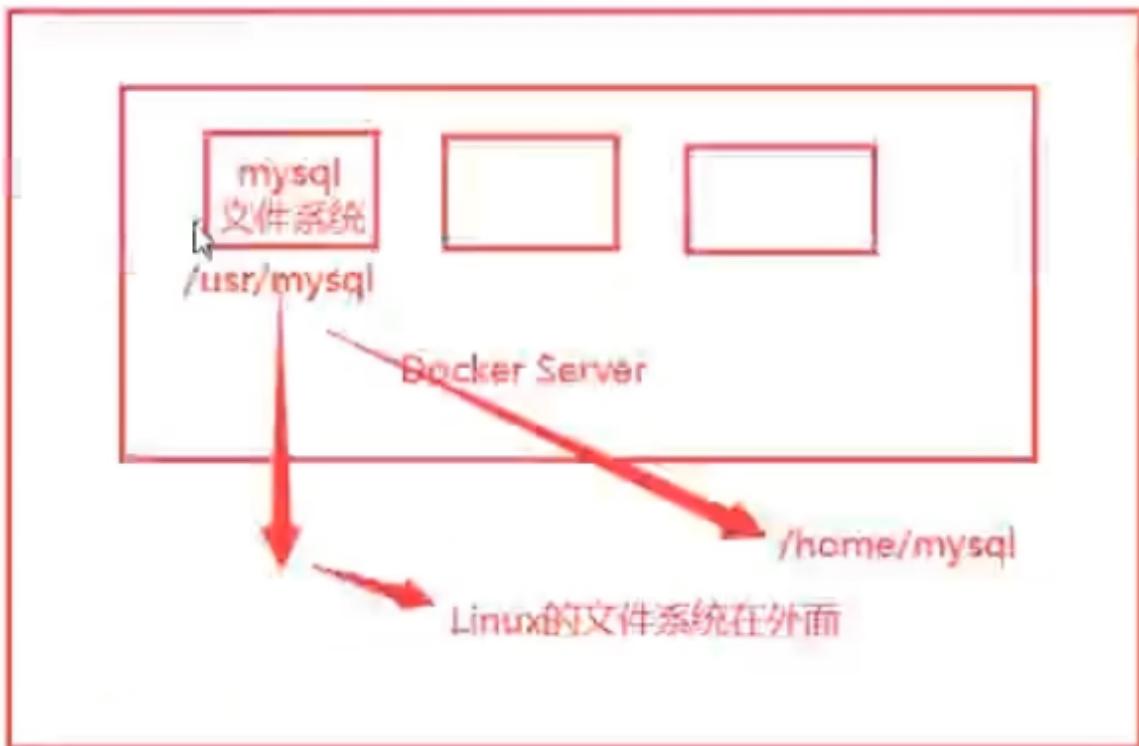
将应用和环境打包成一个镜像！

数据？如果数据都在容器中，那么我们容器删除，数据就会丢失！需求：数据可以持久化

MySQL，容器删除了，删库跑路！需求：MySQL数据可以存储在本地！

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！目录的挂载，将我们容器内的目录，挂载到Linux上面！



总结一句话：容器的持久化和同步操作！容器间也是可以数据共享的！

2. 使用数据卷

- 方式一：直接使用命令挂载 -v

```
-v, --volume list           Bind mount a volume

docker run -it -v 主机目录:容器内目录 -p 主机端口:容器内端口
→ ~ docker run -it -v /home/ceshi:/home centos /bin/bash
#通过 docker inspect 容器id 查看
```

```

    "Name": "overlay2"
},
"Mounts": [      挂载 -v 卷
{
    "Type": "bind",
    "Source": "/home/ceshi", 主机内地址
    "Destination": "/home", docker容器内的地址
    "Mode": "",
    "RW": true,
    "Propagation": "rprivate"
}
],
"Config": {
    "Hostname": "UbuntuDocker"
}
```

测试文件的同步

```

[1] 2021年2月到期 + [2] 2021年2月到期
[root@kuangshen ~]# cd /home
[root@kuangshen home]# ls
!   compose test  kuangshen
[root@kuangshen home]# docker run -it -v /home/ceshi:/home centos
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
8a29e15cef6e: Pull complete
Digest: sha256:fe8d82422041eed5477b63addf40fb06c3b049404242b3198
Status: Downloaded newer image for centos:latest
[root@5bc0fd0f208b ~]# cd /home
[root@5bc0fd0f208b home]# ls
[root@5bc0fd0f208b home]# docker inspect 5bc0fd0f208b
bash: docker: command not found
bash: clear: command not found
[root@5bc0fd0f208b home]# ls
[root@5bc0fd0f208b home]# touch test.java
[root@5bc0fd0f208b home]# ls
test.java
[root@5bc0fd0f208b home]# 

```

```

[1] 2021年2月到期 + [2] 2021年2月到期
[root@kuangshen home]# ls
!   ceshi  compose test  kuangshen
[root@kuangshen home]# cd ceshi
[root@kuangshen ceshi]# ls
[root@kuangshen ceshi]# ls
test.java
[root@kuangshen ceshi]# 

```

同步的过程
双向绑定

再来测试！

- 1、停止容器
- 2、宿主机修改文件
- 3、启动容器
- 4、容器内的数据依旧是同步的

```

[1] 2021年2月到期 + [2] 2021年2月到期
[root@kuangshen home]# docker ps
CONTAINER ID        IMAGE           COMMAND      CREATED          STATUS          PORTS
STATUS              PORTS           NAMES
[root@kuangshen home]# docker ps -a
CONTAINER ID        IMAGE           COMMAND      CREATED          STATUS          PORTS
CREATED
NAMES
5bc0fd0f208b        centos          "/bin/bash"  6 minutes ago   Exited (0) 58 seconds ago
"romantic_jackson"
[root@kuangshen home]# docker start 5bc0fd0f208b
5bc0fd0f208b
[root@kuangshen home]# docker attach 5bc0fd0f208b
[root@5bc0fd0f208b ~]# cd /home
[root@5bc0fd0f208b home]# ls
test.java
[root@5bc0fd0f208b home]# cat test.java
hello,linux update
[root@5bc0fd0f208b home]# 

```

```

[1] 2021年2月到期 + [2] 2021年2月到期
[root@kuangshen home]# ls
!   ceshi  compose test  kuangshen
[root@kuangshen home]# cd ceshi
[root@kuangshen ceshi]# ls
[root@kuangshen ceshi]# ls
test.java
[root@kuangshen ceshi]# vim test
[1]+ Stopped                  vim test
[root@kuangshen ceshi]# vim test.java
[root@kuangshen ceshi]# 

```

好处：我们以后修改只需要在本地修改即可，容器内会自动同步！

3.实战：安装MySQL

- 思考：MySQL的数据持久化的问题

```

# 获取mysql镜像
→ ~ docker pull mysql:5.7
# 运行容器,需要做数据挂载 #安装启动mysql, 需要配置密码的, 这是要注意点!
# 参考官网hub
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d
mysql:tag

#启动我们得
-d 后台运行
-p 端口映射
-v 卷挂载
-e 环境配置
-- name 容器名字
→ ~ docker run -d -p 3306:3306 -v /home/mysql/conf:/etc/mysql/conf.d -v
/home/mysql/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 --name mysql01
mysql:5.7

# 启动成功之后, 我们在本地使用sqlyog来测试一下
# sqlyog-连接到服务器的3306--和容器内的3306映射

```

```
# 在本地测试创建一个数据库，查看一下我们映射的路径是否ok!
```

假设我们将容器删除：

```
[root@kuangshen home]# docker rm -f mysql01
mysql01
[root@kuangshen home]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
[root@kuangshen home]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
NAMES
5bc0fd0f208b        centos             "/bin/bash"         19 minutes ago   Exited (127) 9 minutes ago
romantic_jackson
```

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。

4.具名和匿名挂载

```
# 匿名挂载
→ 容器内路径！
docker run -d -P --name nginx01 -v /etc/nginx nginx

# 查看所有的volume的情况
→ ~ docker volume ls
DRIVER      VOLUME NAME
local
33ae588fae6d34f511a769948f0d3d123c9d45c442ac7728cb85599c2657e50d
local
# 这里发现，这种就是匿名挂载，我们在 -v只写了容器内的路径，没有写容器外的路劲！

# 具名挂载
→ ~ docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx nginx
→ ~ docker volume ls
DRIVER      VOLUME NAME
local      juming-nginx

# 通过 -v 卷名：容器内路径
# 查看一下这个卷
```

```
→ ~ docker volume inspect juming-nginx
[
  {
    "CreatedAt": "2020-05-16T11:32:01+08:00",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/juming-nginx/_data",
    "Name": "juming-nginx",
    "Options": null,
    "Scope": "local"
  }
]
```

所有的docker容器内的卷，没有指定目录的情况下都是在 `/var/lib/docker/volumes/xxxx/_data` 下

如果指定了目录，`docker volume ls` 是查看不到的。

```

→ ~ docker run -d -P --name nginx05 -v /home/nginx05:/etc/nginx/conf.d nginx
863196e8441ce2fd443c8dd8a04eaba053bef007cf31845671cd1ab87cbe793 启动挂载到具体目录
→ ~ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
863196e8441ce2fd443c8dd8a04eaba053bef007cf31845671cd1ab87cbe793
3->80/tcp          nginx              "nginx -g 'daemon of..." 4 seconds ago   Up 3 seconds      0.0.0.0:3277
bc2cc955f71d      nginx              "nginx -g 'daemon of..." 3 minutes ago   Up 3 minutes      0.0.0.0:3277
2->80/tcp          nginx04           "nginx -g 'daemon of..." 9 minutes ago   Up 9 minutes      0.0.0.0:3277
dbb8319f1788      nginx              "nginx -g 'daemon of..." 9 minutes ago   Up 9 minutes      0.0.0.0:3277
0->80/tcp          nginx02           "nginx -g 'daemon of..." 9 minutes ago   Up 9 minutes      0.0.0.0:3277
→ ~ docker volume ls 带着不到
DRIVER              VOLUME NAME
local               juming-nginx
→ ~

```

三种挂载： 匿名挂载、具名挂载、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 docker volume ls 是查看不到的

拓展：

```

# 通过 -v 容器内路径: ro rw 改变读写权限
ro #readonly 只读
rw #readwrite 可读可写
docker run -d -P --name nginx05 -v juming:/etc/nginx:ro nginx
docker run -d -P --name nginx05 -v juming:/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！

```

5.初识Dockerfile

Dockerfile 就是用来构建docker镜像的构建文件！命令脚本！先体验一下！

通过这个脚本可以生成镜像，镜像。

```

# 创建一个dockerfile文件，名字可以随便 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
#这里的每个命令，就是镜像的一层！

```

```

[root@kuangshen docker-test-volume]# docker build -f /home/docker-test-volume/dockerfile1 -t kuangshen/centos:1.0 .
Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM centos
--> 470671670cac
Step 2/4 : VOLUME ["volume01","volume02"]
--> Running in 6cdf866fee64
Removing intermediate container 6cdf866fee64
--> e9747f457806
Step 3/4 : CMD echo "----end----"
--> Running in 1614b3f6649a
Removing intermediate container 1614b3f6649a
--> ebfd137d5f0c
Step 4/4 : CMD /bin/bash
--> Running in ba53710754b2
Removing intermediate container ba53710754b2
--> 5d04f189a434
Successfully built 5d04f189a434
Successfully tagged kuangshen/centos:1.0
[root@kuangshen docker-test-volume]# docker images
REPOSITORY          TAG           IMAGE ID            CREATED             SIZE
kuangshen/centos   1.0           5d04f189a434      22 seconds ago    237MB
mysql               5.7           e73346bdff465     32 hours ago     448MB
nginx               latest         602e111c06b6      3 weeks ago      127MB
centos              latest         470671670cac      3 months ago     237MB
[root@kuangshen docker-test-volume]#

```

启动自己写的镜像

```
[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
```

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

查看一下卷挂载

```
docker inspect 容器id
```

```
{
  "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": ""
    }
  ]
}
```

测试一下刚才的文件是否同步出去了！

这种方式使用的十分多，因为我们通常会构建自己的镜像！

假设构建镜像时候没有挂载卷，要手动镜像挂载 -v 卷名：容器内路径！

6.数据卷容器

多个MySQL同步数据！

命名的容器挂载数据卷！



```
--volumes-from list          Mount volumes from the specified container(s)
# 测试，我们通过刚才启动的
```

```
[root@kuangshen /]# docker run -it --name docker02 --volumes-from docker01 kuangshen/centos:1.0
[root@af993368f540 /]# ls -l
total 56
lrwxrwxrwx  1 root root    7 May 11  2019 bin -> usr/bin
drwxr-xr-x  5 root root  360 May 15 12:19 dev
drwxr-xr-x  1 root root 4096 May 15 12:19 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 116 root root    0 May 15 12:19 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 12:18 volume01
drwxr-xr-x  2 root root 4096 May 15 12:18 volume02
[root@af993368f540 /]# cd volume01
[root@af993368f540 volume01]# ls
docker01
[root@af993368f540 volume01]#
```

docker01 创建的内容同步到了 docker02 上面

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

```
→ ~ docker run -d -p 3306:3306 -v /home/mysql/conf:/etc/mysql/conf.d -v
   /home/mysql/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 --name mysql01
   mysql:5.7
→ ~ docker run -d -p 3307:3306 -e MYSQL_ROOT_PASSWORD=123456 --name
   mysql02 --volumes-from mysql01 mysql:5.7
# 这个时候，可以实现两个容器数据同步！
```

◦ 结论：

容器之间的配置信息的传递，数据卷容器的生命周期一直持续到没有容器使用为止。

但是一旦你持久化到了本地，这个时候，本地的数据是不会删除的！

七、DockerFile

1. 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 构建称为一个镜像
- 3、 docker run 运行镜像
- 4、 docker push 发布镜像（DockerHub、阿里云仓库）

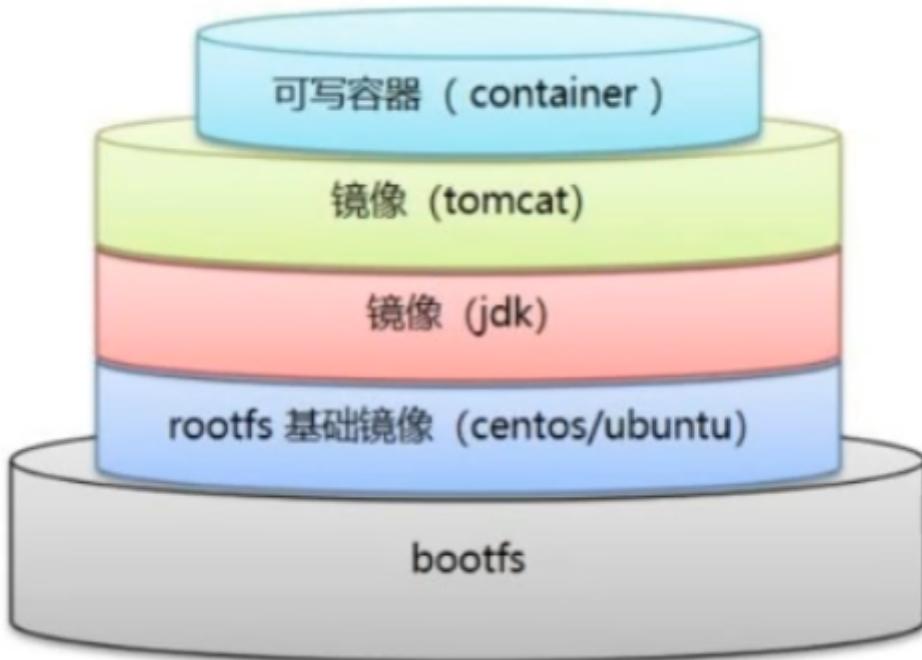
但是很多官方镜像都是基础包，很多功能没有，我们通常会自己搭建自己的镜像！

官方既然可以制作镜像，那我们也可以！

2.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
- 3、#表示注释
- 4、每一个指令都会创建提交一个新的镜像曾，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，最终发布和运行产品。

Docker容器：容器就是镜像运行起来提供服务。

```
# DockerFile常用指令
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！添加内容 添加同目录
WORKDIR      # 镜像的工作目录
VOLUME        # 挂载的目录
EXPOSE       # 保留端口配置
CMD           # 指定这个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   # 指定这个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 这个时候就会运行ONBUILD的指令，触发指令。
COPY          # 类似ADD，将我们文件拷贝到镜像中
ENV           # 构建的时候设置环境变量！
```

3.实战测试

- 创建一个自己的centos

```
```shell
1.编写Dockerfile文件
vim mydockerfile-centos
FROM centos
MAINTAINER cheng<1204598429@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 "----end----"
CMD /bin/bash
```

```shell
2、通过这个文件构建镜像
命令 docker build -f 文件路径 -t 镜像名:[tag] .
docker build -f mydockerfile-centos -t mycentos:0.1 .
```

**测试运行**



我们可以列出本地进行的变更历史


```

- CMD 和 ENTRYPOINT区别

| | |
|------------|-------------------------------------|
| CMD | # 指定这个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 | # 指定这个容器启动的时候要运行的命令，可以追加命令 |

测试cmd

```
# 编写dockerfile文件
$ vim dockerfile-test-cmd
FROM centos
CMD ["ls", "-a"]
# 构建镜像
$ docker build -f dockerfile-test-cmd -t cmd-test:0.1 .
# 运行镜像
$ docker run cmd-test:0.1
.
.
.dockerenv
```

```

bin
dev

# 想追加一个命令 -l 成为ls -al
$ docker run cmd-test:0.1 -l
docker: Error response from daemon: OCI runtime create failed:
container_linux.go:349: starting container process caused "exec: \"-l\":
executable file not found in $PATH": unknown.
ERRO[0000] error waiting for container: context canceled
# cmd的情况下 -l 替换了CMD["ls","-l"]。 -l 不是命令所有报错

```

测试ENTRYPOINT

```

# 编写dockerfile文件
$ vim dockerfile-test-entrypoint
FROM centos
ENTRYPOINT ["ls","-a"]
$ docker run entrypoint-test:0.1

.
.

.dockerenv
bin
dev
etc
home
lib
lib64
lost+found ...

# 我们的命令，是直接拼接在我们得ENTRYPOINT命令后面的
$ docker run entrypoint-test:0.1 -l
total 56
drwxr-xr-x  1 root root 4096 May 16 06:32 .
drwxr-xr-x  1 root root 4096 May 16 06:32 ..
-rwxr-xr-x  1 root root    0 May 16 06:32 .dockerenv
lrwxrwxrwx  1 root root    7 May 11 2019 bin -> usr/bin
drwxr-xr-x  5 root root  340 May 16 06:32 dev
drwxr-xr-x  1 root root 4096 May 16 06:32 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 ....

```

Dockerfile中很多命令都十分的相似，我们需要了解它们的区别，我们最好的学习就是对比他们然后测试效果！

4.实战：Tomcat镜像

1、准备镜像文件

准备tomcat 和 jdk到当前目录，编写好README。

```

→ docker-tomcat ls
apache-tomcat-9.0.35.tar.gz  dockerfile  jdk-8u231-linux-x64.tar.gz  README
→ docker-tomcat

```

2、编写dokerfile

```
FROM centos #
MAINTAINER cheng<1204598429@qq.com>
COPY README /usr/local/README #复制文件
ADD jdk-8u231-linux-x64.tar.gz /usr/local/ #复制解压
ADD apache-tomcat-9.0.35.tar.gz /usr/local/ #复制解压
RUN yum -y install vim
ENV MYPATH /usr/local #设置环境变量
WORKDIR $MYPATH #设置工作目录
ENV JAVA_HOME /usr/local/jdk1.8.0_231 #设置环境变量
ENV CATALINA_HOME /usr/local/apache-tomcat-9.0.35 #设置环境变量
ENV PATH $PATH:$JAVA_HOME/bin:$CATALINA_HOME/lib #设置环境变量 分隔符是:
EXPOSE 8080 #设置暴露的端口
CMD /usr/local/apache-tomcat-9.0.35/bin/startup.sh && tail -F /usr/local/apache-tomcat-9.0.35/logs/catalina.out # 设置默认命令
```

3、构建镜像

```
# 因为dockerfile命名使用默认命名 因此不用使用-f 指定文件
$ docker build -t mytomcat:0.1 .
```

4、run镜像

```
$ docker run -d -p 8080:8080 --name tomcat01 -v
/home/kuangshen/build/tomcat/test:/usr/local/apache-tomcat-9.0.35/webapps/test -
v /home/kuangshen/build/tomcat/tomcatlogs:/usr/local/apache-tomcat-9.0.35/logs
mytomcat:0.1
```

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！)

发现：项目部署成功，可以直接访问！

我们以后开发的步骤：需要掌握Dockerfile的编写！我们之后的一切都是使用docker镜像来发布运行！

5.发布自己的镜像

1、地址 <https://hub.docker.com/>

2、确定这个账号可以登录

3、登录

```
$ 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
```

4、提交 push镜像

```
→ tomcatlogs docker login -u chengcode
Password:
Error response from daemon: Get https://registry-1.docker.io/v2/: unauthorized: incorrect username or password
→ tomcatlogs docker login -u chengcoder
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
→ tomcatlogs docker push mytomcat
The push refers to repository [docker.io/library/mymtomcat]
0175410fd8ce: Preparing
005704bc69a: Preparing
e13eaca5a438: Preparing
030692781086: Preparing
0683de282177: Preparing
```

会发现push不上去，因为如果没有前缀的话默认是push到 官方的library

解决方法

第一种 build的时候添加你的dockerhub用户名，然后在push就可以放到自己的仓库了

```
$ docker build -t chengcoder/mymtomcat:0.1 .
```

第二种 使用docker tag #然后再次push

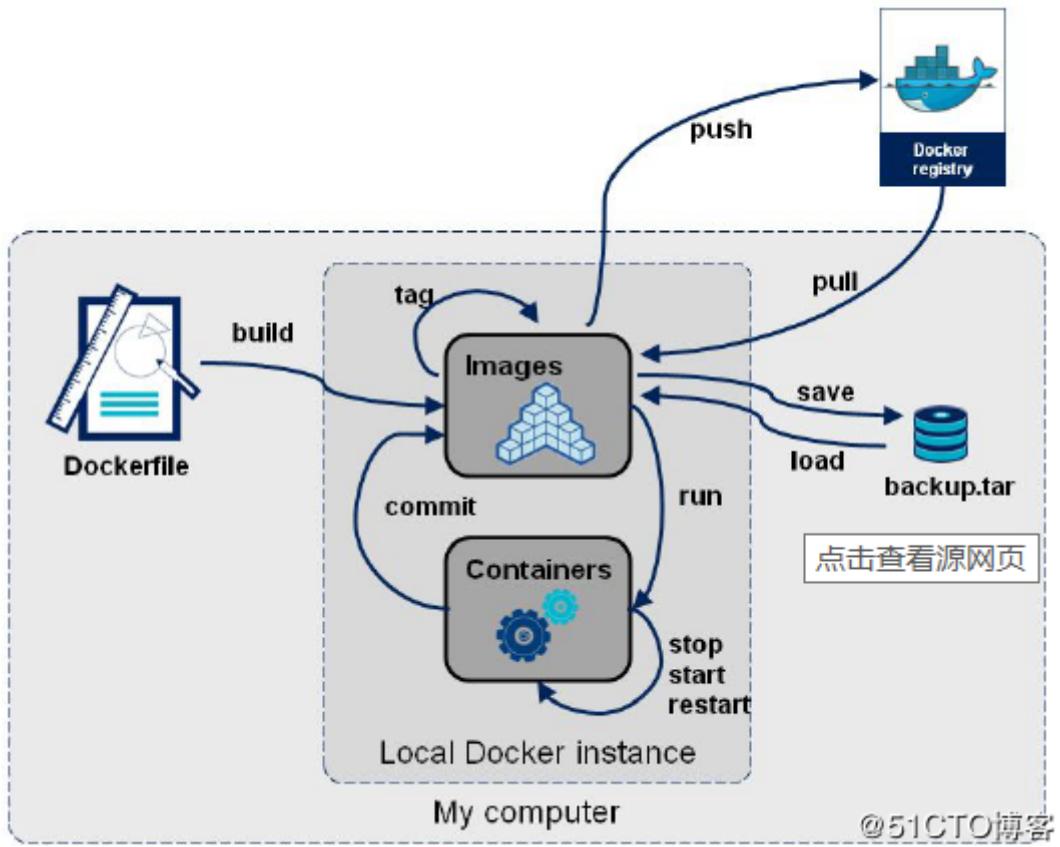
```
$ docker tag 容器id chengcoder/mymtomcat:1.0 #然后再次push
```

- 阿里云镜像服务上

看官网 很详细<https://cr.console.aliyun.com/repository/>

```
```shell
$ sudo docker login --username=zchengx registry.cn-shenzhen.aliyuncs.com
$ sudo docker tag [ImageId] registry.cn-shenzhen.aliyuncs.com/dsadxzc/cheng:[镜像
版本号]
修改id 和 版本
sudo docker tag a5ef1f32aaaae registry.cn-shenzhen.aliyuncs.com/dsadxzc/cheng:1.0
修改版本
$ sudo docker push registry.cn-shenzhen.aliyuncs.com/dsadxzc/cheng:[镜像版本号]
```
```

6.小结



八、Docker 网络

1.理解Docker 0

清空所有网络

```
[root@kuangshen tomcat]# 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
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:16:3e:30:27:f4 brd ff:ff:ff:ff:ff:ff
    inet 172.17.90.138/20 brd 172.17.95.255 scope global dynamic eth0
        valid_lft 310744886sec preferred_lft 310744886sec
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:bb:71:07:06 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global docker0
        valid_lft forever preferred_lft forever
```

三个网络

- 问题：docker 是如何处理容器网络访问的？

```
# 测试 运行一个tomcat
$ docker run -d --name tomcat01 tomcat

$ 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
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
551: vethbfc37e3@if550: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether 1a:81:06:13:ec:a1 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

```

inet6 fe80::1881:6ff:fe13:eca1/64 scope link
  valid_lft forever preferred_lft forever

$ docker exec -it 容器id
$ ip addr
# 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@if551 ip地址, docker分配!
550: eth0@if551: <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能不能ping通容器内部! 可以 容器内部可以ping通外界吗? 可以!
$ 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=0.069 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.074 ms

```

○ 原理

1、我们每启动一个docker容器，docker就会给docker容器分配一个ip，我们只要按照了docker，就会有一个docker0桥接模式，使用的技术是veth-pair技术！

<https://www.cnblogs.com/bakari/p/10613710.html>

再次测试ip add

2、在启动一个容器测试，发现又多了一对网络

我们发现这个容器带来网卡，都是一对对的

veth-pair 就是一对的虚拟设备接口，他们都是成对出现的，一端连着协议，一端彼此相连
正因为有这个特性 veth-pair 充当一个桥梁，连接各种虚拟网络设备的
OpenStac,Docker容器之间的连接，OVS的连接，都是使用veth-pair技术

3、我们来测试下tomcat01和tomcat02是否可以ping通

```

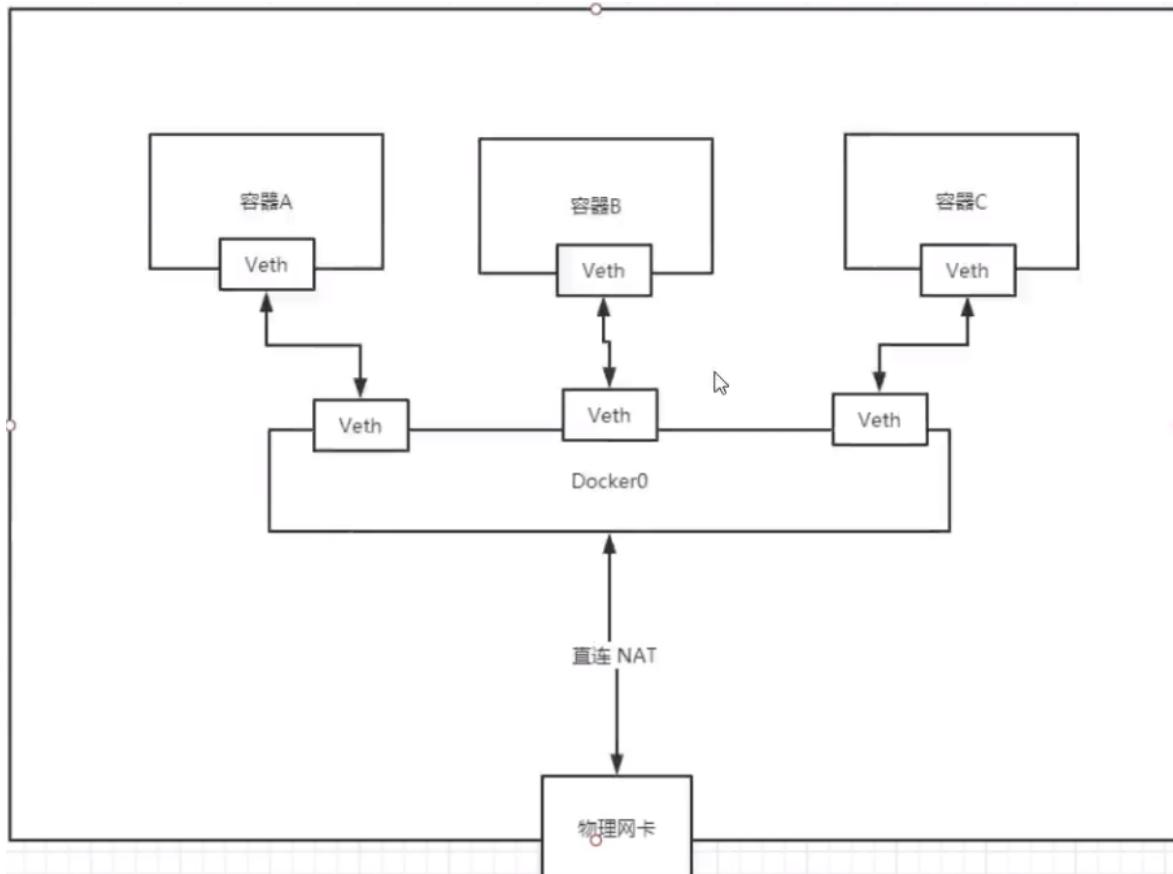
```shell
$ docker-tomcat docker exec -it tomcat01 ip addr #获取tomcat01的ip
172.17.0.2
550: eth0@if551: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
 link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
 inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
 valid_lft forever preferred_lft forever
$ docker-tomcat docker exec -it tomcat02 ping 172.17.0.2#让tomcat02ping
tomcat01
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=0.098 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.071 ms
可以ping通
```

```


结论: tomcat01和tomcat02公用一个路由器, docker0。

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

小结: Docker使用的是Linux的桥接, 宿主机是一个Docker容器的网桥 docker0



Docker中所有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, 对应的网桥一对就没了!

- 思考一个场景: 我们编写了一个微服务, database url=ip: 项目不重启, 数据ip换了, 我们希望可以处理这个问题, 可以通过名字来进行访问容器?

2.-link

```
$ docker exec -it tomcat02 ping tomcat01 # ping不通
ping: tomcat01: Name or service not known
# 运行一个tomcat03 --link tomcat02
$ docker run -d -P --name tomcat03 --link tomcat02 tomcat
5f9331566980a9e92bc54681caaac14e9fc993f14ad13d98534026c08c0a9aef
# 用tomcat03 ping tomcat02 可以ping通
$ 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=0.115 ms
64 bytes from tomcat02 (172.17.0.3): icmp_seq=2 ttl=64 time=0.080 ms

# 用tomcat02 ping tomcat03 ping不通
```

探究: docker network inspect 网络id 网段相同

docker inspect tomcat03

```
"Links": [
    "/tomcat02:/tomcat03/tomcat02"
],
```

查看tomcat03里面的/etc/hosts发现有tomcat02的配置

```
→ docker-tomcat docker exec 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 9a9f30d83b57
172.17.0.4      5f9331566980
```

-link 本质就是在hosts配置中添加映射

现在使用Docker已经不建议使用-link了！

自定义网络，不适用docker0！

docker0问题：不支持容器名连接访问！

3.自定义网络

```
docker network
connect      -- Connect a container to a network
create       -- Creates a new network with a name specified by the
disconnect   -- Disconnects a container from a network
inspect      -- Displays detailed information on a network
ls           -- Lists all the networks created by the user
prune        -- Remove all unused networks
rm           -- Deletes one or more networks
```

- 查看所有的docker网络

```
→ ~ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
ad5ada6a106f    bridge    bridge      local
19315b394ac7    host      host       local
8203922769a9    none     null       local
```

网络模式

bridge：桥接 docker（默认，自己创建也是用bridge模式）

none：不配置网络，一般不用

host：和所主机共享网络

container：容器网络连通（用得少！局限很大）

测试

```

# 我们直接启动的命令 --net bridge, 而这个就是我们得docker0
# bridge就是docker0
$ docker run -d -P --name tomcat01 tomcat
等价于 => docker run -d -P --name tomcat01 --net bridge tomcat

# docker0, 特点: 默认, 域名不能访问。 --link可以打通连接, 但是很麻烦!
# 我们可以 自定义一个网络
$ docker network create --driver bridge --subnet 192.168.0.0/16 --gateway
192.168.0.1 mynet

```

```

→ ~ docker network create --driver bridge --subnet 192.168.0.0/16 --
gateway 192.168.0.1 mynet
aabfa0c0a0bdc266facb15e1e00e6e4d68c5ecec5c64919c0f8d04da9637ff0f
→ ~ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
ad5ada6a106f    bridge    bridge      local
19315b394ac7    host      host       local
aabfa0c0a0bd    mynet    bridge      local
8203922769a9    none     null       local
→ ~

```

```
$ docker network inspect mynet;
```

```

→ ~ docker network inspect mynet
[{"Name": "mynet",
 "Id": "aabfa0c0a0bdc266facb15e1e00e6e4d68c5ecec5c64919c0f8d04da9637ff0f",
 "Created": "2020-05-16T19:11:09.183843491+08:00",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
     "Driver": "default",
     "Options": {},
     "Config": [
         {
             "Subnet": "192.168.0.0/16",
             "Gateway": "192.168.0.1"
         }
     ]
 },
 "Internal": false,
 "Attachable": false,
 "Ingress": false,
 "ConfigFrom": {
     "Network": ""
 },
 "ConfigOnly": false,
 "Containers": {},
 "Options": {},
 "Labels": {}
}
]
```

启动两个tomcat,再次查看网络情况

```

→ ~ docker run -d -P --name tomcat-net-01 --net mynet tomcat
1c3fbb2f22df905d80e8c447839822d78ddc98220ce66fd599c572ad235fa13a
→ ~ docker run -d -P --name tomcat-net-02 --net mynet tomcat
524f7b9526347398c63f058a68886f780728d9c09e87f83c92d6a0be73ae00ac

```

```
→ ~ docker network inspect mynet
[{"Name": "mynet",
 "Id": "aabfa0c0a0bdc266facb15e1e00e6e4d68c5ecec5c64919c0f8d84da9637ffff",
 "Created": "2020-05-16T19:11:09.183843491+08:00",
 "Scope": "local",
 "Driver": "bridge",
 "EnableIPv6": false,
 "IPAM": {
     "Driver": "default",
     "Options": {},
     "Config": [
         {
             "Subnet": "192.168.0.0/16",
             "Gateway": "192.168.0.1"
         }
     ]
 },
 "Internal": false,
 "Attachable": false,
 "Ingress": false,
 "ConfigFrom": {
     "Network": ""
 },
 "ConfigOnly": false,
 "Containers": {
     "1c3fb2f22df905d80e8c447839822d78ddc98220ce66fd599c572ad235fa13a": {
         "Name": "tomcat-net-01",
         "EndpointID": "cablabeece4854019ebfffc3211f2e708c6829fe2add01e15f5668ala0bcb5e",
         "MacAddress": "02:42:c0:a8:00:02",
         "IPv4Address": "192.168.0.2/16",
         "IPv6Address": ""
     },
     "524f7b9526347398c63f058a68886f780728d9c09e87f83c92d6a0be73ae00ac": {
         "Name": "tomcat-net-02",
         "EndpointID": "63a8b3a7178c244b7131663430fa6f4a20bf9d96df4d18d7b1a940a82369795d",
         "MacAddress": "02:42:c0:a8:00:03",
         "IPv4Address": "192.168.0.3/16",
         "IPv6Address": ""
     }
 },
 "Options": {},
 "Labels": {}
}
]
→ ~
```

在自定义的网络下，服务可以互相ping通，不用使用-link

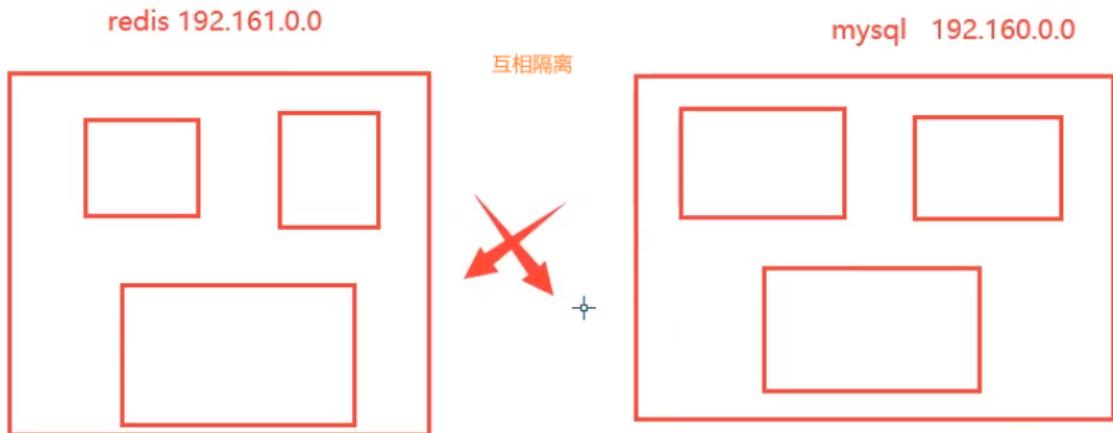
```
→ ~ docker exec tomcat-net-02 ping tomcat-net-01
PING tomcat-net-01 (192.168.0.2) 56(84) bytes of data.
64 bytes from tomcat-net-01.mynet (192.168.0.2): icmp_seq=1 ttl=64 time=0.133 ms
^C
→ ~ docker exec tomcat-net-01 ping tomcat-net-02
PING tomcat-net-02 (192.168.0.3) 56(84) bytes of data.
64 bytes from tomcat-net-02.mynet (192.168.0.3): icmp_seq=1 ttl=64 time=0.114 ms
^C
→ ~
```

我们自定义的网络docker当我们维护好了对应的关系，推荐我们平时这样使用网络！

好处：

redis -不同的集群使用不同的网络，保证集群是安全和健康的

mysql-不同的集群使用不同的网络，保证集群是安全和健康的



4. 网络连通

```
→ ~ docker network

Usage: docker network COMMAND

Manage networks

Commands:
  connect      Connect a container to a network
  create       Create a network
  disconnect   Disconnect a container from a network
  inspect      Display detailed information on one or more networks
  ls           List networks
  prune        Remove all unused networks
  rm           Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.

→ ~ docker network connect --help

Usage: docker network connect [OPTIONS] NETWORK CONTAINER

Connect a container to a network

Options:
  --alias strings      Add network-scoped alias for the container
  --driver-opt strings  driver options for the network
  --ip string          IPv4 address (e.g., 172.30.100.104)
  --ip6 string         IPv6 address (e.g., 2001:db8::33)
  --link list          Add link to another container
  --link-local-ip strings  Add a link-local address for the container
```

```
# 测试两个不同的网络连通 再启动两个tomcat 使用默认网络, 即docker0
$ docker run -d -P --name tomcat01 tomcat
$ docker run -d -P --name tomcat02 tomcat
# 此时ping不通
```

要将tomcat01 连通 tomcat—net-01 , 连通就是将 tomcat01加到 mynet网络
一个容器两个ip (tomcat01)

```

→ ~ docker network connect tomcat01 mynet
Error response from daemon: No such container: mynet
→ ~ docker network connect mynet tomcat01
→ ~ docker exec tomcat-net-01 ping tomcat01
PING tomcat01 (192.168.0.4) 56(84) bytes of data.
64 bytes from tomcat01.mynet (192.168.0.4): icmp_seq=1 ttl=64 time=0.100 ms
64 bytes from tomcat01.mynet (192.168.0.4): icmp_seq=2 ttl=64 time=0.104 ms
^C
→ ~ docker exec tomcat-net-01 ping tomcat02
ping: tomcat02: Name or service not known
→ ~ 

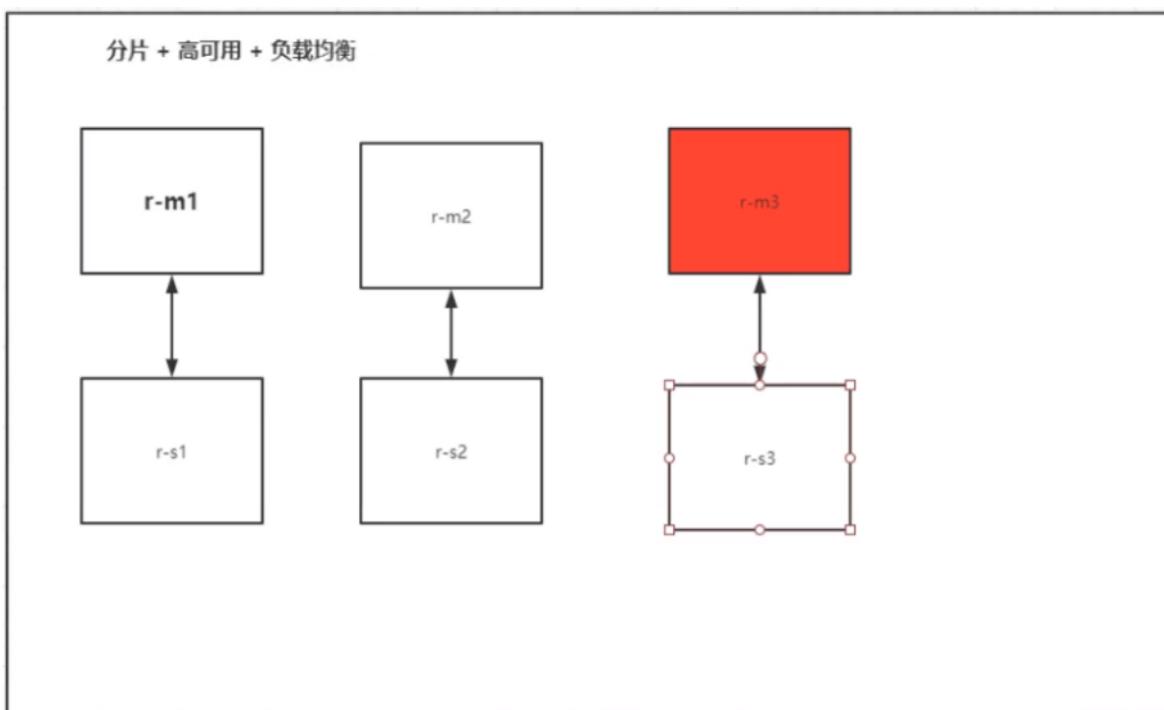
```

01连通，加入后此时，已经可以tomcat01 和 tomcat-01-net ping通了

02是依旧不通的

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

5.实战：部署Redis集群



```

# 创建网卡
docker network create redis --subnet 172.38.0.0/16
# 通过脚本创建六个redis配置
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

```

```

EOF
done

# 通过脚本运行六个redis
for port in $(seq 1 6); \
do
  docker run -p 637${port}:6379 -p 1667${port}:16379 --name redis-${port} \
  -v /mydata/redis/node-$port:/data:/data \
  -v /mydata/redis/node-$port/conf/redis.conf:/etc/redis/redis.conf \
  -d --net redis --ip 172.38.0.1${port} redis:5.0.9-alpine3.11 redis-server \
  /etc/redis/redis.conf
  docker exec -it redis-1 /bin/sh #redis默认没有bash
  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

```

```

/data # 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
9 --cluster-replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 172.38.0.15:6379 to 172.38.0.11:6379
Adding replica 172.38.0.16:6379 to 172.38.0.12:6379
Adding replica 172.38.0.14:6379 to 172.38.0.13:6379
M: 0fd135f258b39add0ef9bc39a1228f8243d43516 172.38.0.11:6379
  slots:[0-5460] (5461 slots) master
M: b6b05fd69b5911c836b5cba43d978bfdd74f5596 172.38.0.12:6379
  slots:[5461-10922] (5462 slots) master
M: e98d326f8d74b29db274f395cbf364d9dc6ad2f2 172.38.0.13:6379
  slots:[10923-16383] (5461 slots) master
S: 526f4365cca8a@1d658d659f1908881b218f7929 172.38.0.14:6379
  replicates e98d326f8d74b29db274f395cbf364d9dc6ad2f2
S: fd58b402cd6f3e41d2c945617d8a0aa448eb83f 172.38.0.15:6379
  replicates 0fd135f258b39add0ef9bc39a1228f8243d43516
S: 5bd1e3ddf4c7ea3aa10f128615a66d623a33fb8b 172.38.0.16:6379
  replicates b6b05fd69b5911c836b5cba43d978bfdd74f5596
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
...
>>> Performing Cluster Check (using node 172.38.0.11:6379)
M: 0fd135f258b39add0ef9bc39a1228f8243d43516 172.38.0.11:6379
  slots:[0-5460] (5461 slots) master
  1 additional replica(s)
S: fd58b402cd6f3e41d2c945617d8a0aa448eb83f 172.38.0.15:6379
  slots: (0 slots) slave
  replicates 0fd135f258b39add0ef9bc39a1228f8243d43516
S: 5bd1e3ddf4c7ea3aa10f128615a66d623a33fb8b 172.38.0.16:6379
  slots: (0 slots) slave
  replicates b6b05fd69b5911c836b5cba43d978bfdd74f5596
S: 526f4365cca8a@1d658d659f1908881b218f7929 172.38.0.14:6379
  slots: (0 slots) slave
  replicates e98d326f8d74b29db274f395cbf364d9dc6ad2f2
M: e98d326f8d74b29db274f395cbf364d9dc6ad2f2 172.38.0.13:6379
  slots:[10923-16383] (5461 slots) master
  1 additional replica(s)
M: b6b05fd69b5911c836b5cba43d978bfdd74f5596 172.38.0.12:6379
  slots:[5461-10922] (5462 slots) master
  1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
/data #

```

docker搭建redis集群完成！

我们使用docker之后，所有的技术都会慢慢变得简单起来！

九、SpringBoot项目打包Docker镜像

1、构建SpringBoot项目

2、打包运行

```
mvn package
```

3、编写dockerfile

```
FROM java:8
COPY *.jar /app.jar
CMD ["--server.port=8080"]
EXPOSE 8080
ENTRYPOINT ["java","-jar","app.jar"]
```

4. 构建镜像

```
# 1. 复制jar和Dockerfile到服务器
# 2. 构建镜像
$ docker build -t xxxxx:xx .
```

5. 发布运行

以后我们使用了Docker之后，给别人交付就是一个镜像即可！

十、SpringBoot项目打包Docker镜像（添加）

1、认识docker

- Docker定义：Docker 是一个开源的应用容器引擎，它可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制（沙箱是一个虚拟系统程序，沙箱提供的环境相对于每一个运行的程序都是独立的，而且不会对现有的系统产生影响），相互之间不会有任何接口，更重要的是容器性能开销极低。
- Docker的优点

Docker 是一个用于开发，交付和运行应用程序的开放平台。Docker 使您能够将应用程序与基础架构分开，从而可以快速交付软件。借助 Docker，您可以与管理应用程序相同的方式来管理基础架构。通过利用 Docker 的方法来快速交付，测试和部署代码，您可以大大减少编写代码和在生产环境中运行代码之间的延迟。

- Docker应用场景
- Web 应用的自动化打包和发布。
 - 自动化测试和持续集成、发布。
 - 在服务型环境中部署和调整数据库或其他的后台应用。
 - 从头编译或者扩展现有的 OpenShift 或 Cloud Foundry 平台来搭建自己的 PaaS 环境。

2.Docker容器与虚拟机

我们用的传统虚拟机如 VMware 之类的需要模拟整台机器包括硬件，每台虚拟机都需要有自己的操作系统，虚拟机一旦被开启，预分配给它的资源将全部被占用。每一台虚拟机包括应用，必要的二进制和库，以及一个完整的用户操作系统。

而容器技术是和我们的宿主机共享硬件资源及操作系统，可以实现资源的动态分配。容器包含应用和其所有的依赖包，但是与其他容器共享内核。容器在宿主机操作系统中，在用户空间以分离的进程运行。

容器技术是实现操作系统虚拟化的一种途径，可以让您在资源受到隔离的进程中运行应用程序及其依赖关系。通过使用容器，我们可以轻松打包应用程序的代码、配置和依赖关系，将其变成容易使用的构建块，从而实现环境一致性、运营效率、开发人员生产力和版本控制等诸多目标。容器可以帮助保证应用程序快速、可靠、一致地部署，其间不受部署环境的影响。容器还赋予我们对资源更多的精细化控制能力，让我们的基础设施效率更高。

Docker 属于 Linux 容器的一种封装，提供简单易用的容器使用接口。它是目前最流行的 Linux 容器解决方案。

Docker 将应用程序与该程序的依赖，打包在一个文件里面。运行这个文件，就会生成一个虚拟容器。程序在这个虚拟容器里运行，就好像在真实的物理机上运行一样。有了 Docker，就不用担心环境问题。

总体来说，Docker 的接口相当简单，用户可以方便地创建和使用容器，把自己的应用放入容器。容器还可以进行版本管理、复制、分享、修改，就像管理普通的代码一样。

- Docker的几个重要概念，如下图所示：



- 镜像就类似于在创建虚拟机前需要下载的系统镜像文件，比如iso文件、img文件等一些镜像文件。

镜像是 Docker 运行容器的前提，仓库是存放镜像的场所，可见镜像更是 Docker 的核心。

- 容器可以类比于正在运行中的虚拟机。
- 你可以将你的镜像 save 为一个 tar 文件，别人就可以通过 load 来获取你的镜像。
- 仓库中则保存了很多公共的常用的镜像，比如常用的 JDK 镜像、MySQL 镜像、tomcat 镜像、Ubuntu 镜像、nginx 镜像等等。你可以通过 pull 来拉取获得这些镜像，你也可以自定义一些镜像通过 push 推送到仓库中。
- Dockerfile 就是一个 build 镜像的文件，它描述并指定了应该如何构建一个镜像。

Dockerfile 是自动构建 docker 镜像的配置文件，用户可以使用 **Dockerfile** 快速创建自定义的镜像。**Dockerfile** 中的命令非常类似于 Linux 下的 shell 命令。

一般来说，我们可以将 **Dockerfile** 分为四个部分：

- 基础镜像(父镜像)信息指令 FROM
- 维护者信息指令 MAINTAINER
- 镜像操作指令 RUN 、 ENV 、 ADD 和 WORKDIR 等
- 容器启动指令 CMD 、 ENTRYPOINT 和 USER 等

3.Docker 命令

1.Docker 容器使用

获取镜像

启动容器

查看容器

启动已经停止的容器

后台运行

停止一个容器

进入容器

退出容器

导出容器

导入容器

容器快照

删除容器

清理掉所有处于终止状态的容器

2.Docker Web应用

运行一个web应用

以端口5000开启端口

关闭web应用

重启web应用

移除web应用

3.Docker 镜像应用

列出镜像列表

获取镜像

4.Dockerfile

构建镜像

4.SpringBoot项目容器化步骤

step1：添加Docker的maven的插件，配置Dockerfile的path；

```
<!-- Docker maven plugin -->
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.13</version>
    <configuration>
        <imageName>${docker.image.prefix}/${project.artifactId}</imageName>
        <dockerDirectory>${project.basedir}/src/main/docker</dockerDirectory>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>
                <include>${project.build.finalName}.jar</include>
            </resource>
        </resources>
    </configuration>
</plugin>
```

step2：在配置的Dockerfile的path处添加Dockerfile文件；

Step3：文件中添加配置：

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
#把当前项目下dockertest-0.0.1-SNAPSHOT.jar 改名为test.jar 添加到镜像中
ADD web-app-template-1.0.0.jar test.jar
#指定端口，最好写与项目配置的端口
EXPOSE 8080
#在镜像中运行/test.jar包，这样在运行镜像的时候就已经启动好了test.jar
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/test.jar"]
```

Step4: mvn clean package -Dmaven.test.skip=true (表示不执行测试用例，也不编译测试用例类。)

step5: mvn package docker:build 打镜像

step6: docker images 查看镜像

step7: docker run -p 8081:8081 -t springboot/web-app-template 运行

step8: 查看运行结果: [http://localhost:8081/....](http://localhost:8081/)

step9: docker push