1. 容器化docker

> **本质**：容器的本质就是**进程**，**容器**就是未来**云计算**系统中的进程，**镜像**就是这个系统中的**安装包**。**K8S就是操作系统**！
>
> 解决了最大的问题就是：环境无法跨平台

特点：

- **一次构建，到处运行**（Build Once，Run Anywhere）
- 对系统消耗不是特别多
- 可以快速启动
- 维护简单
- 扩展容易

1.1 三大组件

- **镜像（images）**：相对于模板，可以通过这个模板来创建容器服务，通过这个镜像可以创建多个容器
- **容器（container）**：简易的linux系统
- **仓库（repository）**：用于存储镜像
    - 私有的仓库
    - 共有的仓库：国外Docker hub（默认）、国内阿里云

1.2 底层实现原理（linux内核）

- Namespace：基于进程的隔离（容器的进程、网络、消息（IPC通信）、文件系统（UFS）、主机名、用户）
- Cgroups：控制资源的度量和配额
- Overlay2(UnionFS)：联合文件系统

1.3 安装与卸载

有`docker-ce`社区版、`docker-ee`商业版

1.3.1 安装

```
# 0. 查看当前系统版本
[root@huihui ~]# cat /etc/os-release

# 1. 要求CentOS系统的内核版本高于 3.10 ，查看本页面的前提条件来验证你的CentOS 版本是否支持Docker
[root@huihui ~]# uname -r

# 2. 使用root权限登录 Centos。确保 yum 包更新到最新
[root@huihui ~]# sudo yum update

# 3. 卸载旧版本(如果安装过旧版本的话)
[root@huihui ~]# sudo yum remove docker  docker-common docker-selinux docker-engine

# 4.下载关于Docker的依赖环境
[root@huihui ~]# yum -y install yum-utils device-mapper-persistent-data lvm2

# 5.设置下载Docker的镜像源(阿里云)
[root@huihui ~]# yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

# 6.更新yum软件包索引
[root@huihui ~]# yum makacache fast

# 7. 安装docker-ce最新版本
[root@huihui ~]# yum -y install docker-ce docker-ce-cli containerd.io
```

```
# 8. 启动Docker服务
[root@huihui ~]# systemctl start docker

# 9. 设置开机启动
[root@huihui ~]# systemctl enable docker

# 10. 查看docker版本，检查是否安装成功
[root@huihui ~]# docker version

# 11. 测试，若本地没有hello-world镜像，则会去docker hub下载该镜像
[root@huihui ~]# docker run hello-world

# 12. 查看hello-world镜像是否下载成功
[root@huihui ~]# docker images
```

1.3.2 卸载

```
# 1. 卸载依赖
[root@huihui ~]# yum remove docker-ce docker-ce-cli containerd.io

# 2. 删除默认的docker工作路径
[root@huihui ~]# rm -rf /var/lib/docker
```

1.4 配置国内镜像加速

`vim /etc/docker/daemon.json`，修改如下内容：

```
[root@huihui ~]# sudo mkdir -p /etc/docker
[root@huihui ~]# sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://1cfnvbel.mirror.aliyuncs.com"]
}
EOF
```

```
# 重启两个服务
[root@huihui ~]# systemctl daemon-reload
[root@huihui ~]# systemctl restart docker
```

1.5 常用指令集

1.5.1 帮助相关

- 显示docker版本信息

  ```
  [root@huihui ~]# docker version
  ```

- 显示docker 系统信息，包括镜像和容器的数量

```
[root@huihui ~]# docker info
```

- 帮助命令

```
[root@huihui ~]# docker <命令> --help
```

1.5.2 镜像相关

- 搜索镜像

```
[root@huihui ~]# docker search nginx

--filter:  通过所有过滤
```

- 下载镜像

```
docker pull 镜像名[:tag]

[root@huihui ~]# docker pull mysql
Using default tag: latest   # 默认下载最新版本,
latest: Pulling from library/mysql
d121f8d1c412: Pull complete   # 分层下载
f3cebc0b4691: Pull complete
1862755a0b37: Pull complete
489b44f3dbb4: Pull complete
690874f836db: Pull complete
baa8be383ffb: Pull complete
55356608b4ac: Pull complete
dd35ceccb6eb: Pull complete
429b35712b19: Pull complete
162d8291095c: Pull complete
5e500ef7181b: Pull complete
af7528e958b6: Pull complete
Digest:
sha256:e1bfe11693ed2052cb3b4e5fa356c65381129e87e38551c6cd6ec532ebe0e808 #
签名信息
Status: Downloaded newer image for mysql:latest
docker.io/library/mysql:latest # 真实地址


# --- 拉取指定版本
[root@huihui ~]# docker pull mysql:5.7
5.7: Pulling from library/mysql
d121f8d1c412: Already exists  # 增量更新，已下载的不会重新下载
f3cebc0b4691: Already exists
1862755a0b37: Already exists
489b44f3dbb4: Already exists
```

```
690874f836db: Already exists
baa8be383ffb: Already exists
55356608b4ac: Already exists
277d8f888368: Pull complete   # 增量更新
21f2da6feb67: Pull complete   # 增量更新
2c98f818bcb9: Pull complete   # 增量更新
031b0a770162: Pull complete   # 增量更新
Digest:
sha256:14fd47ec8724954b63d1a236d2299b8da25c9bbb8eacc739bb88038d82da4919 #
签名
Status: Downloaded newer image for mysql:5.7
docker.io/library/mysql:5.7 # 真实地址



docker pull mysql
# ======等价于======
docker pull docker.io/library/mysql:latest
```

- 查看全部镜像

```
[root@huihui ~]# docker images -aq

-a, --all     # 列出全部镜像
-q, --quiet   # 只显示镜像id
```

Docker 镜像是分层的

- 列出部分镜像

```
[root@huihui ~]# docker images nginx
```

- 查看镜像的元数据

```
[root@huihui ~]# docker inspect [ImageID]
```

- 删除一个或多个镜像

```
[root@huihui ~]# docker rmi <name|id>
```

- 删除全部镜像

```
[root@huihui ~]# docker rmi -f $(docker images -q)

# 想要删除untagged images，也就是那些id为的image的话可以用
[root@huihui ~]# docker rmi $(docker images | grep "^<none>" | awk "{print
$3}")
```

- 制作自己的镜像

```
[root@huihui ~]# docker commit -m 'commit消息' [容器ID] mycentos:latest
```

- 镜像的导入导出

```
# 导出镜像（官方不推荐）
[root@huihui ~]# docker save -o mynginx.tar.gz mycentos
或
[root@huihui ~]# docker save > mynginx.tar.gz nginx:latest

# 导入镜像（官方不推荐）
[root@huihui ~]# docker load -i mynginx.tar.gz
或
[root@huihui ~]# docker load < mynginx.tar.gz
```

1.5.3 容器相关

- 启动容器

```
# 简单操作
docker run <镜像的标识id|镜像名称[:tag]>

# 常用的参数
docker run -d -p <宿主机端口:容器端口> --name <容器名称> <镜像的标识id|镜像名称
[:tag]>

-d        : 代表后台运行容器
-p        : 宿主机端口:容器端口，为了映射当前Linux的端口和容器的端口
--name    : 指定容器名称
-v        : 宿主机目录:容器指定目录（将宿主机目录挂载到容器中）
-P        : 随机端口
-it       : 使用交互方式运行，进入容器查看内容
-e        : 设置环境变量
```

- 查看正在运行的容器

```
docker ps [-qa]

-a：查看全部的容器，包括没有运行
-q：只查看容器的标识id
```

- 进入容器

```
docker exec -it <容器id> bash

-i：将容器的标准输入保持打开
-t：创建一个虚拟终端
```

attach：直接进入容器正在运行的terminl窗口

exec：进入容器后新打开一个terminl窗口

- 退出容器

```
exit          # 直接停止容器并退出
ctrl + P + Q # 容器不停止退出
```

- 删除指定容器

```
docker rm -f <容器id>

-f: 强制删除，包括正在运行的容器
```

- 删除所有容器

```
docker rm -f $(docker ps -qa)
```

- 查看容器的日志

```
docker logs -f <容器id>

-f: 动态查看日志
```

- 停止指定容器

```
docker stop <容器id>
```

- 停止全部容器

```
docker stop $(docker ps -qa)
docker kill $(docker ps -qa)
```

- 重启容器

```
docker restart [容器ID]
```

- 移除所有已停止的容器

```
docker container prune
```

- 查看容器中进程信息

```
docker top <容器ID>
```

- 查看容器的元数据

```
docker inspect <容器ID>
```

- 将容器的文件拷贝到宿主机上

```
docker cp <容器id>:~/test.md ./
```

- 查看cpu的状态

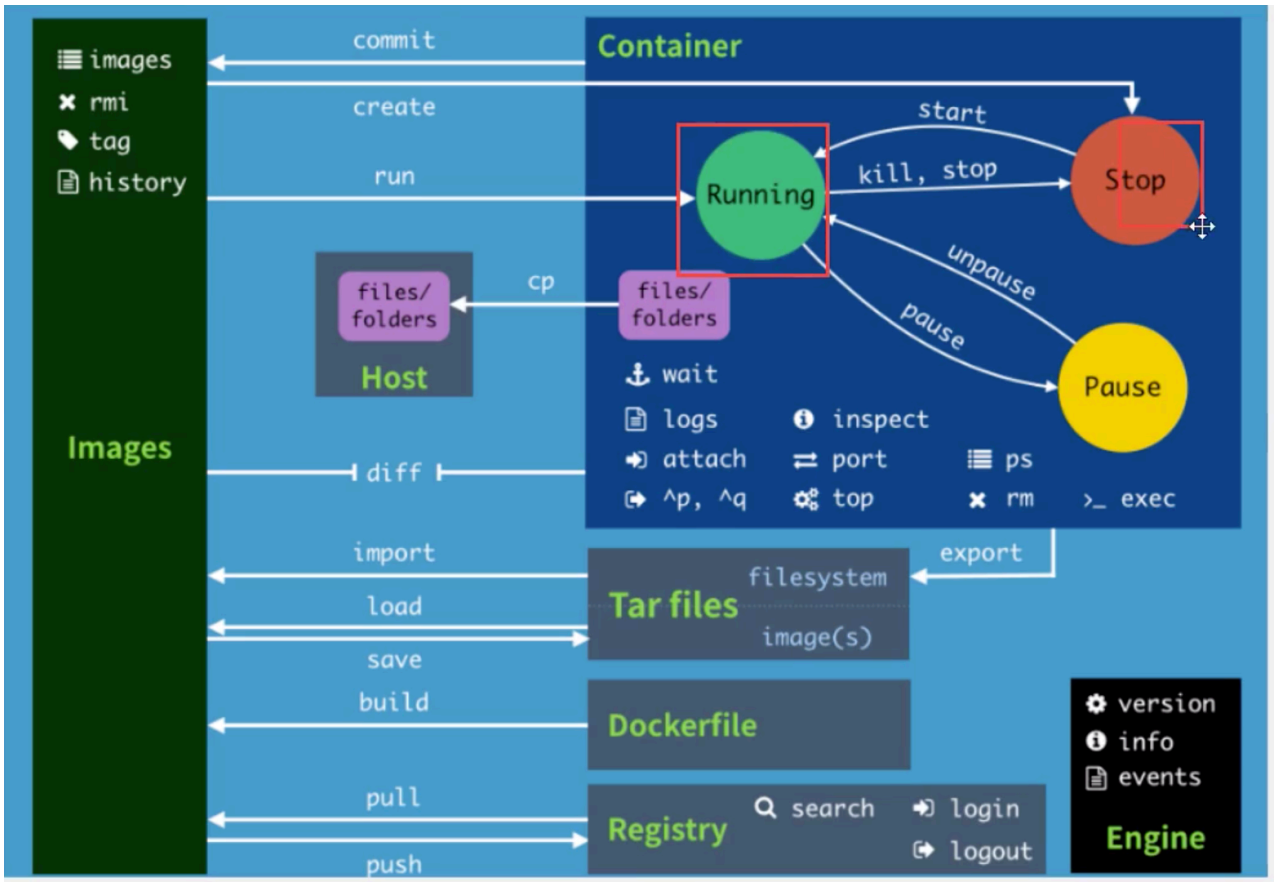
```
docker stats
```

- 容器的导入导出

```
# 导出容器
[root@huihui ~]# docker export -o test.tar [容器ID]

# 导入容器
[root@huihui ~]# docker import test.tar test:v1
```

小结：




```
attach  # 当前shell 下attach连接指定运行镜像
build   # 通过Dockfile定制镜像
commit  # 提交当前容器为新的镜像
cp      # 从容器中拷贝指定文件或者目录
create  # 创建一个新的容器，同run，但不启动容器
diff    # 查看docker容器变化
events  # 从docker服务获取容器实时事件
exec    # 在已存在的容器上运行命令
export  # 导出容器的内容流作为一个tar归档文件【对应import】
history # 显示一个镜像形成历史
images  # 列出系统当前镜像
import  # 从tar包中的内容创建一个新的文件系统镜像【对应export】
info    # 显示系统相关信息
inspect # 查看容器详细信息
kill    # 杀死指定的docker容器
```

```
load    # 从一个tar包中加载一个镜像【对应save】
login   # 注册或者登陆一个docker源服务器
logout  # 从当前Docker registry 退出
logs    # 输出当前容器日志信息
port    # 查看映射端口对应的容器内部源端口
pause   # 暂停容器
ps      # 列出容器列表
pull    # 从docker 镜像源服务器拉取指定镜像
push    # 推送指定镜像或库镜像至docker源服务器
restart # 重启运行的容器
rm      # 移除一个或者多个容器
rmi     # 移除一个或者多个镜像【该镜像没有关联容器，方可删除，否则需要使用-f参数，强制删除】
```

1.6 持久化volume

容器中数据持久化主要有两种方式：

1. 数据卷（Data Volumes）
2. 数据卷容器（Data Volumes Dontainers）

1.6.1 适用场景

- 多个容器之间共享数据
- 宿主机不保证存在固定的目录结构
- 持久化数据到远程主机或者云存储而非本地
- 需要备份、迁移、合并数据时。停止container，将volume整体复制，用于备份、迁移、合并等。

1.6.2 数据卷

本质和bind mount的操作方式一样，只不过在volume这种方式中，宿主机目录是由docker进行管理的，然后再挂载到容器中，不会覆盖容器中的对应目录文件，是共享的。

数据卷是一个可供一个或多个容器使用的特殊目录，可以绕过UFS（Unix File System）。

1. 数据卷可以在容器之间共享和重用
2. 对数据卷的修改会立马生效
3. 对数据卷的更新，不会影响镜像
4. 数据卷默认会一直存在，即使容器被删除
5. 一个容器可以挂载多个数据卷

注意：数据卷的使用，类似于 Linux 下对目录或文件进行 mount。

```
# 1. 将宿主机的/tmp/html路径 挂载到容器内的/etc/nginx/html中。
docker run --name nginx-data2 -v /tmp/html:/etc/nginx/html nginx

# 2. 查看是否挂载成功
docker inspect <容器ID>
```

具名和匿名挂载：

```
# 匿名挂载
docker run --name nginx01 -v /etc/nginx/html nginx

# 查看所有的 volume 的情况
docker volume ls

# 通过-v  卷名:容器内路径
# 查看一个卷 存储的位置
docker volume inspect <卷名>
```

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/<卷名>/_data`

```
# 如何确定是具名挂载还是匿名挂载，还是指定路径挂载!

-v 容器内路径              # 匿名挂载
-v 卷名:容器内路径          # 具名挂载
-v 宿主机路径:容器内路径    # 指定路径挂载
```

拓展：

```
# 通过 -v 容器内路径，  ro、rw 改变读写权限
ro    readonly      # 只读
rw    readwrite     # 可读可写


docker run --name nginx01 -v hginx:/etc/nginx/html:ro nginx
docker run --name nginx02 -v hginx:/etc/nginx/html: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，容器内是无法操作的
```

1.6.3 数据卷容器

```
docker run -it --name docker01 nginx
docker run -it --name docker02 --volume-from docker01 nginx
docker run -it --name docker03 --volume-from docker01 nginx
```

1.7 Dockerfile


```
FROM         # 基础镜镜像,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
WORKDIR      # 镜像的工作目录
VOLUME       # 挂载的目录
EXPOSE       # 暴露端口配置
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!
```

1.7.1 构建Tomcat镜像

- 编写`Dockerfile`文件

```
# 基础镜像
FROM centos:7
# 作者信息
MAINTAINER zouhui<1650941960@qq.com>

# 将宿主机的jdk文件拷至镜像的/usr/local目录下，会自动解压
ADD jdk-8u45-linux-x64.tar.gz /usr/local

# 设置环境变量
ENV JAVA_HOME /usr/local/jdk1.8.0_45

ADD apache-tomcat-8.0.46.tar.gz /usr/local

# 拷贝文件或目录到镜像中，用法同ADD，只是不支持自动下载和解压
```

```
COPY server.xml /usr/local/apache-tomcat-8.0.46/conf

# 删除tar包
RUN rm -rf /usr/local/*.tar.gz

# 当前工作目录
WORKDIR /usr/local/apache-tomcat-8.0.46

# 暴露端口
EXPOSE 8080

# 设置启动命令
ENTRYPOINT ["./bin/catalina.sh","run"]
```

- 执行以下指令集

```
# 构建tomcat镜像
docker build -t tomcat:v1 .
# 创建容器并启动
docker run -it -d --name=tomcat -p 8080:8080 \
    -v /app/webapps/:/usr/local/apache-tomcat-8.0.46/webapps/ \
    tomcat:v1
```

1.7.2 关键字全面详解

```
# 1. FROM（指明构建的新镜像是来自于哪个基础镜像）
FROM centos:7

# 2. MAINTAINER（用来指定镜像创建者信息）
MAINTAINER 1650941960@qq.com

# 3. RUN（安装软件用），该指令有两种格式：
RUN ["yum", "install", "httpd"]
RUN yum install httpd

# 4. CMD（启动容器时执行的Shell命令），该指令有三种格式：
CMD ["-C", "/start.sh"]
CMD ["/usr/sbin/sshd", "-D"]
CMD /usr/sbin/sshd -D

# 5. 当Dockerfile指定了ENTRYPOINT，那么使用下面的格式：
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)

# 6. ENTRYPOINT（启动容器时执行的Shell命令，同CMD类似，只是由ENTRYPOINT启动的程序不会被docker run命令行指定的参数所覆盖，而且，这些命令行参数会被当作参数传递给ENTRYPOINT指定指定的程序），两种格式：
ENTRYPOINT ["/bin/bash", "-C", "/start.sh"]
ENTRYPOINT /bin/bash -C '/start.sh'
# 注意：Dockerfile文件中也可以存在多个ENTRYPOINT指令，但仅有最后一个会生效。
```

```
# 7. USER（设置container容器的用户）
USER root

# 8. EXPOSE（指定容器需要映射到宿主机器的端口）
EXPOSE 80 443

# 9. ENV（用于设置环境变量）
ENV MYSQL_ROOT_PASSWORD 123456
ENV JAVA_HOME /usr/local/jdk1.8.0_45

# 10. ADD（拷贝文件或目录到镜像中）
ADD html.tar.gz /var/www/html
ADD https://xxx.com/html.tar.gz /var/www/html
# 注意：如果是URL或压缩包，会自动下载或自动解压。

# 11. COPY（拷贝文件或目录到镜像中，用法同ADD，只是不支持自动下载和解压）
COPY ./start.sh /start.sh

# 12. VOLUME（指定容器挂载点到宿主机自动生成的目录或其他容器））
VOLUME ["/var/lib/mysql"]
# 注意：一般不会在Dockerfile中用到，更常见的还是在docker run的时候指定-v数据卷。

# 13. WORKDIR（登陆容器默认目录）
WORKDIR /data

# 14. ONBUILD（在子镜像中执行）
格式：ONBUILD <Dockerfile关键字>
```

PS：补充（CMD和ENTRYPOINT的区别）

- CMD（启动容器时执行的Shell命令，CMD启动的程序**会被docker run命令行指定的参数所覆盖**，）因为CMD命令很容易被docker run命令的方式覆盖，所以，如果你希望你的docker镜像的功能足够灵活，建议在Dockerfile里调用CMD命令。
- ENTRYPOINT（启动容器时执行的Shell命令，同CMD类似，只是由ENTRYPOINT启动的程序**不会被docker run命令行指定的参数所覆盖**，而且，**这些命令行参数会被当作参数传递给ENTRYPOINT**，entrypoint会把/bin/bash当成一个echo的字符串参数，不会进入到容器中。）

  ENTRYPOINT的作用不同，如果你希望你的docker镜像只执行一个具体程序，不希望用户在执行docker run的时候随意覆盖默认程序. 建议用ENTRYPOINT.

```
[root@centos ~]# vim Dockerfile

FROM centos:centos7.1.1503
ENTRYPOINT ["/bin/echo", "This is test entrypoint"]


[root@centos ~]# docker run -it csphere/ent:0.1 /bin/bash
This is test entrypoint /bin/bash
```

注意：Dockerfile文件中也可以存在多个ENTRYPOINT指令，但仅有最后一个会生效。

- CMD 和 ENTRYPOINT组合使用

```
# 定义了 ENTRYPOINT 的话，CMD 只为 ENTRYPOINT 提供参数

FROM ubuntu:trusty
ENTRYPOINT ["/bin/ping","-c","3"]
CMD ["localhost"]

[root@centos ~]]# docker run demo
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.038 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.051 ms

--- localhost ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.025/0.038/0.051/0.010 ms


上面执行的命令是ENTRYPOINT和CMD指令拼接而成.，ENTRYPOINT和CMD同时存在时，docker把CMD的命令拼接到ENTRYPOINT命令之后，拼接后的命令才是最终执行的命令．但是由于上文说docker run命令行执行时，可以覆盖CMD指令的值．如果你希望这个docker镜像启动后不是ping localhost，而是ping其他服务器，可以这样执行docker run
```

`CMD` 和 `ENTRYPOINT` 命令的小总结:

Docker在很多情况下被用来打包一个程序. 想象你有一个用python脚本实现的程序, 你需要发布这个python程序. 如果用docker打包了这个python程序, 你的最终使用用户就不需要安装python解释器和python的库依赖，你可以把所有依赖工具打包进docker镜像里，然后用ENTRYPOINT指向你的Python脚本本身，当然你也可以用CMD命令指向Python脚本，但是通常用ENTRYPOINT可以表明你的docker镜像只是用来执行这个python脚本，也不希望最终用户用这个docker镜像做其他操作.

1.7.3 镜像优化

优化方法：

- 一次RUN指令形成新的一层，尽量Shell命令都写在一行，减少镜像层。
- 一次RUN形成新的一层，如果没有在同一层删除，无论文件是否最后删除，都会带到下一层，所以要在每一层清理对应的残留数据，减小镜像大小。

镜像优化的参考链接： https://blog.csdn.net/M2l0ZgSsVc7r69eFdTj/article/details/81977225

1.7.4 发布镜像到公有云

> Dockerhub

1. 注册自己的账号
2. 在我们服务器上提交自己的镜像

```
# 1. 登录自己的Dockerhub账号
[root@huihui ]# docker login -u <username>

# 2. 为本地镜像打Tag标签
[root@huihui ]# docker tag <镜像ID> <新标签名字>

# 3. 推送镜像到DockerHub
[root@huihui ]# docker push <新标签名字>
```

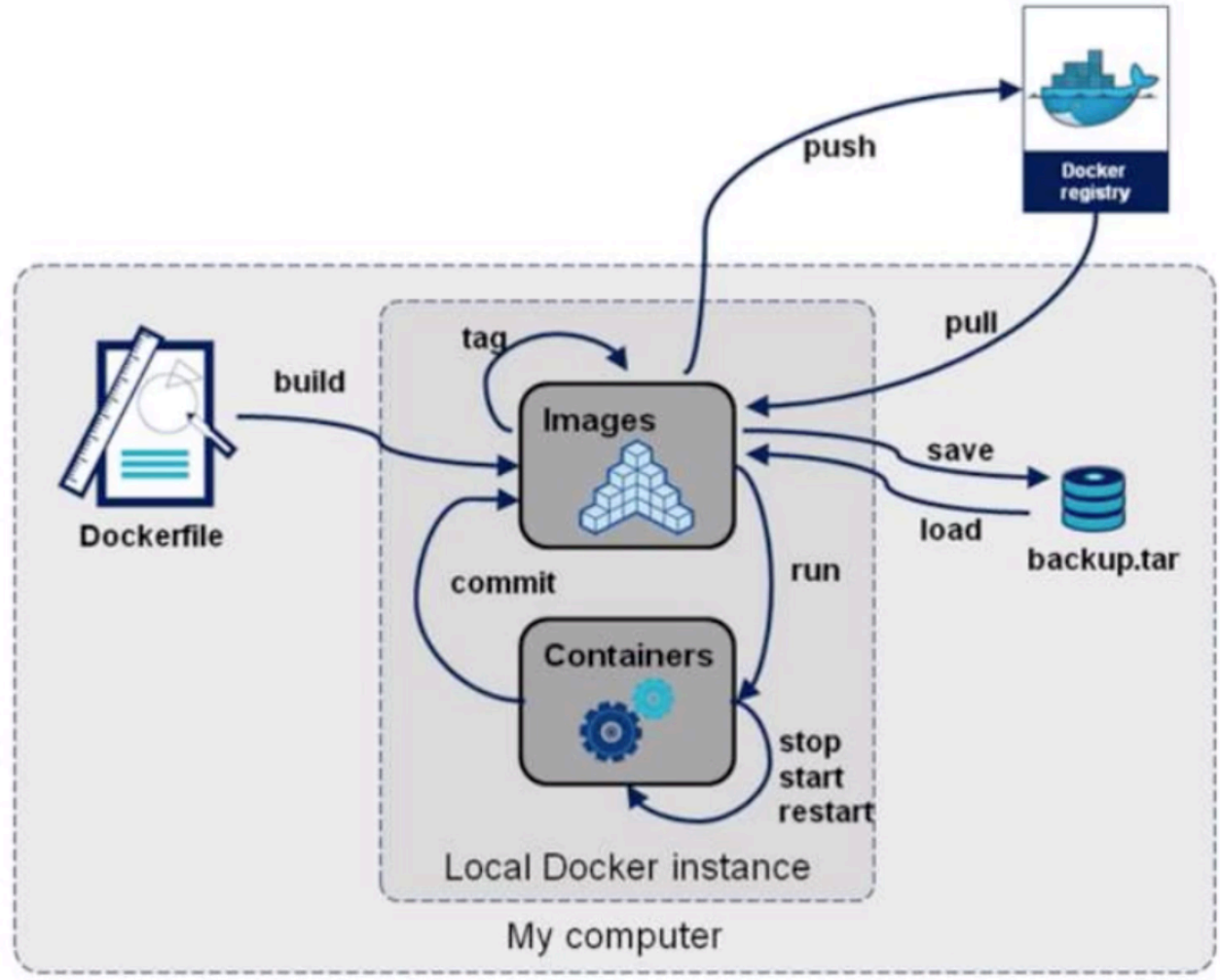
> 阿里云镜像

1. 登录阿里云
2. 找到容器镜像服务

```
# 1. 登录阿里云Docker Registry
$ sudo docker login --username=18988886666 registry.cn-qingdao.aliyuncs.com

# 2. 重命名
$ sudo docker tag [ImageId] registry.cn-qingdao.aliyuncs.com/test_wk/work:[镜像版本号]

# 3. 推送镜像到阿里云
$ sudo docker push registry.cn-qingdao.aliyuncs.com/test_wk/work:[镜像版本号]
```



1.8 网络Docker0

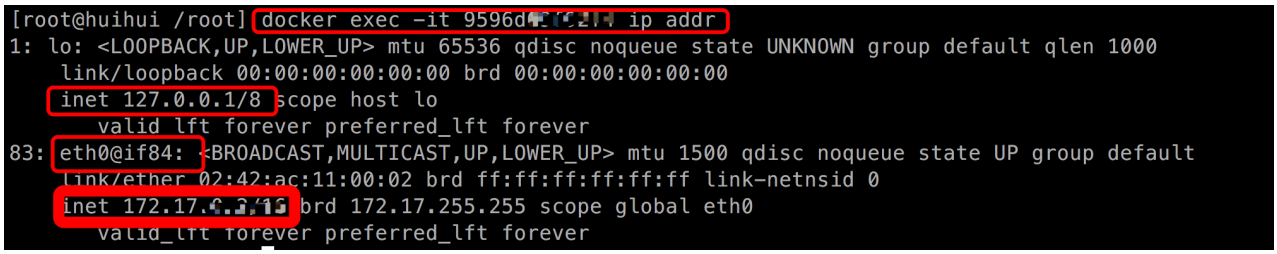
> Docker0 默认网桥模式，域名不能访问，--link可以打通连接！

前置条件：清空当前docker所有环境

- 宿主机网络

```
[root@huihui /root] 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                          本地回环地址
       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:04:ff:7a brd ff:ff:ff:ff:ff:ff      阿里云内网地址
    inet 172.31.[illegible] brd [illegible] scope global dynamic eth0
       valid_lft 285923637sec preferred_lft 285923637sec
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:5c:dc:d2:e7 brd ff:ff:ff:ff:ff:ff
    inet 172.17.[illegible] brd [illegible] scope global docker0   docker0地址
       valid_lft forever preferred_lft forever
```

- docker容器内网络



> 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，我们只要安装了docker，就会有一个网卡docker0
2. 底层依赖linux操作系统的veth pair技术，实现通信

--link

```
# 通过--link 就可以直接通过容器名实现容器内网路通信
docker run -d -P --name tomcat03 --link tomcat01 tomcat

# 此时tomcat03 可以ping 通 tomcat01，但反向则不可以，查看其原因

docker network ls
docker network inspect [NetworkID]

docker exec -it [ContainerID] cat /etc/hosts
```

1.8.1 自定义网络

```
# 1. 查看所有的docker网络
docker network ls

# 2. 创建网络
docker network create --driver bridge --subnet 192.168.0.0/16 --gateway
192.168.0.1 mynet

# 3. 创建并通过自定义网络启动两个容器
[root@huihui /root] docker run -d -P --name tomcat-net-0 --net mynet tomcat
[root@huihui /root] docker run -d -P --name tomcat-net-1 --net mynet tomcat

# 4. 此时容器内两端就可以通过域名连通
[root@huihui /root] docker exec -it tomcat-net-0 ping tomcat-net-1
PING tomcat-net-1 (192.168.0.3) 56(84) bytes of data.
64 bytes from tomcat-net-1.mynet (192.168.0.3): icmp_seq=1 ttl=64 time=0.079 ms
64 bytes from tomcat-net-1.mynet (192.168.0.3): icmp_seq=2 ttl=64 time=0.077 ms

[root@huihui /root] docker exec -it tomcat-net-1 ping tomcat-net-0
PING tomcat-net-0 (192.168.0.2) 56(84) bytes of data.
64 bytes from tomcat-net-0.mynet (192.168.0.2): icmp_seq=1 ttl=64 time=0.055 ms
```

```
64 bytes from tomcat-net-0.mynet (192.168.0.2): icmp_seq=2 ttl=64 time=0.117 ms
64 bytes from tomcat-net-0.mynet (192.168.0.2): icmp_seq=4 ttl=64 time=0.089 ms

# 5. 查看自定义网络元信息
[root@huihui /root] docker network inspect <NetworkID>
[
    {
        "Name": "mynet",
        "Id":
"e18b013b7b7e90d927f61b5ea6a6be276a0b200a0f6488099debbf59cb104d53",
        "Created": "2020-11-29T11:16:38.881828435+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": {
            "1f0e4e69e3757e289618fee0d062fec05a2aa1ad9a3e74b4a665d454c21317e7":
{
                "Name": "tomcat-net-0",
                "EndpointID":
"e772c0f70d862b89099f4e70dc191105a5102709998666bff3ff9330aecc51cd",
                "MacAddress": "02:42:c0:a8:00:02",
                "IPv4Address": "192.168.0.2/16",
                "IPv6Address": ""
            },
            "3007f43e20456455f298ec7c91c3a6fb5b0ca22278ddfef9cbc3726c4784dd4d":
{
                "Name": "tomcat-net-1",
                "EndpointID":
"a04621cd3c59612791d1176b241124db16c5c026cac1c78cf0d185a515345b46",
                "MacAddress": "02:42:c0:a8:00:03",
                "IPv4Address": "192.168.0.3/16",
                "IPv6Address": ""
```

```
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

网络模式

host模式

- 众所周知，Docker使用了Linux的Namespaces技术来进行资源隔离，如PID Namespace隔离进程，Mount Namespace隔离文件系统，Network Namespace隔离网络等。一个Network Namespace提供了一份独立的网络环境，包括网卡、路由、Iptable规则等都与其他的Network Namespace隔离。一个Docker容器一般会分配一个独立的Network Namespace。但如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。
- 例如，我们在10.10.101.105/24的机器上用host模式启动一个含有web应用的Docker容器，监听tcp80端口。当我们在容器中执行任何类似ifconfig命令查看网络环境时，看到的都是宿主机上的信息。而外界访问容器中的应用，则直接使用10.10.101.105:80即可，不用任何NAT转换，就如直接跑在宿主机中一样。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。

container模式（一般不用）

- 在理解了host模式后，这个模式也就好理解了。这个模式指定新创建的容器和已经存在的一个容器共享一个Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的IP，而是和一个指定的容器共享IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过lo网卡设备通信。

none模式（一般不用）

- 这个模式和前两个不同。在这种模式下，Docker容器拥有自己的Network Namespace，但是，并不为Docker容器进行任何网络配置。也就是说，这个Docker容器没有网卡、IP、路由等信息。需要我们自己为Docker容器添加网卡、配置IP等。

bridge模式（默认）

1. 当Docker进程启动时，会在主机上创建一个名为docker0的虚拟网桥，此主机上启动的Docker容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。
2. 从docker0子网中分配一个IP给容器使用，并设置docker0的IP地址为容器的默认网关。在主机上创建一对虚拟网卡veth pair设备，Docker将veth pair设备的一端放在新创建的容器中，并命名为eth0（容器的网卡），另一端放在主机中，以vethxxx这样类似的名字命名，并将这个网络设备加入到docker0网桥中。可以通过brctl show命令查看。
3. bridge模式是docker的默认网络模式，不写--net参数，就是bridge模式。使用docker run -p时，docker实际是在iptables做了DNAT规则，实现端口转发功能。可以使用iptables -t nat -vnL查看。

1.8.2 网络连通

> 将某个容器连接到某个网络

```
# 1. 将tomcat01容器连接到mynet网络上，一个容器两个ip
[root@huihui /root] docker network connect mynet tomcat01

# 2. 测试tomcat01的网络连通性
[root@huihui /root] docker exec -it tomcat01 ping tomcat-net-1
PING tomcat-net-1 (192.168.0.3) 56(84) bytes of data.
64 bytes from tomcat-net-1.mynet (192.168.0.3): icmp_seq=1 ttl=64 time=0.109 ms
64 bytes from tomcat-net-1.mynet (192.168.0.3): icmp_seq=2 ttl=64 time=0.075 ms
64 bytes from tomcat-net-1.mynet (192.168.0.3): icmp_seq=3 ttl=64 time=0.118 ms

# 3. 分析其原理
[root@huihui /root] docker network inspect mynet
[
    {
        "Name": "mynet",
        "Id":
"e18b013b7b7e90d927f61b5ea6a6be276a0b200a0f6488099debbf59cb104d53",
        "Created": "2020-11-29T11:16:38.881828435+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": {
            "1f0e4e69e3757e289618fee0d062fec05a2aa1ad9a3e74b4a665d454c21317e7":
{
                "Name": "tomcat-net-0",
                "EndpointID":
"e772c0f70d862b89099f4e70dc191105a5102709998666bff3ff9330aecc51cd",
```

```
                "MacAddress": "02:42:c0:a8:00:02",
                "IPv4Address": "192.168.0.2/16",
                "IPv6Address": ""
            },
            "3007f43e20456455f298ec7c91c3a6fb5b0ca22278ddfef9cbc3726c4784dd4d":
{
                "Name": "tomcat-net-1",
                "EndpointID":
"a04621cd3c59612791d1176b241124db16c5c026cac1c78cf0d185a515345b46",
                "MacAddress": "02:42:c0:a8:00:03",
                "IPv4Address": "192.168.0.3/16",
                "IPv6Address": ""
            },
            "ad5a139dfd0372ba26e609a52fb8f9ed46bef6296c5bcf20eace20196a14ca76":
{
                "Name": "tomcat01",
                "EndpointID":
"60571be4d35a0643560c2a27dc4de47e21bee55cb7f1bd507656cbb2943a2f8c",
                "MacAddress": "02:42:c0:a8:00:04",
                "IPv4Address": "192.168.0.4/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```


1.9 Web UI工具

- portainer

```
$ docker run -d -p 9000:9000 --name portainer --restart=always  -v
/var/run/docker.sock:/var/run/docker.sock -v  portainer_data:/data
portainer/portainer
```

- Rancher（CI/CD再用）

2. 解决各种坑

- 问题描述

```
doker启动时，报错：docker: Error response from daemon: OCI runtime create failed:
container_linux.go:348: starting container process caused
"process_linux.go:301: running exec setns process for init caused \"exit status
23\"": unknown.

环境：Ubuntu 14.04
```

- 产生原因

docker的版本和linux的内核版本不兼容

- 解决办法

```
升级linux内核，执行下列命令

apt-get install --install-recommends linux-generic-lts-xenial
注意，更新了内核后，需要重启系统。
```