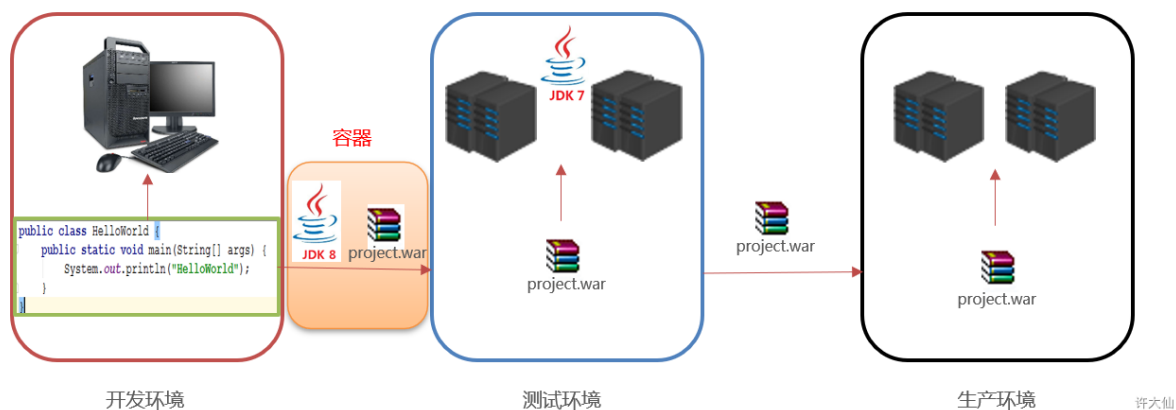


初识Docker

1 Docker出现的背景

- 在实际开发过程中，会出现很多环境：开发环境、测试环境以及生产环境。

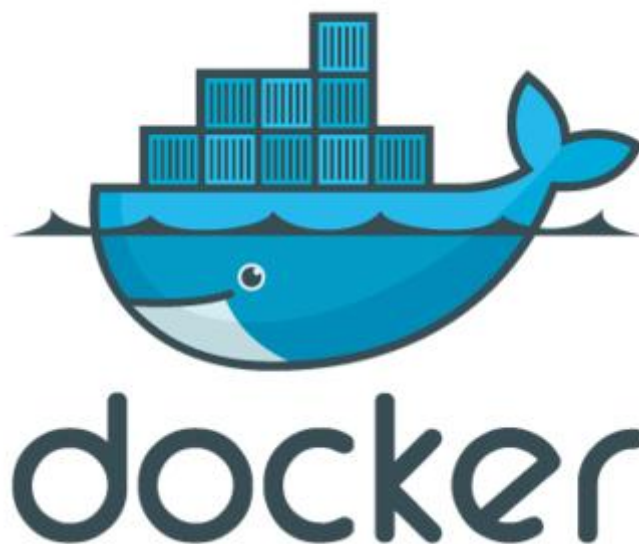


- 一款产品从开发到上线，从操作系统，到运行环境，再到应用配置。作为开发+运维之间的协作我们需要关心很多东西，这也是很多互联网公司都不得不面对的问题，特别是各种版本的迭代之后，不同版本环境的兼容，对运维人员都是考验。
- Docker之所以如此迅速，也是因为它对此给出了一个标准化的解决方案。
- 环境配置如此麻烦，换一台机器，就要重来一次，费时费力。很多人想到，能不能从根本上解决问题，软件可以带环境安装？也就是说，安装的时候，把原始环境一模一样地复制过来。开发人员利用Docker可以消除协作编码时“在我的机器上可以正常运行”的问题。
- 之前在服务器配置一个应用的运行环境，需要安装各种软件，比如JavaEE项目，至少需要安装JDK、MySQL、Maven、Tomcat等，安装和配置这些东西有多麻烦不说，关键是其不能跨平台。比如我们在Windows上安装的这些环境，到了Linux又得重新安装。况且就算不跨操作系统，换另一台同样操作系统的服务器，要移植应用也是非常麻烦的。
- 传统上认为，软件编码开发/测试结束后，所产出的成果即程序或是能够编译执行的二进制字节码等（以java为例）。而为了让这些程序可以顺利执行，开发团队也得准备完整的部署文件，让运维团队得以部署应用程序，开发需要清楚的告诉运维部署团队，用的全部配置文件+所有软件环境。不过，即便如此，仍然常常发生部署失败的情况。Docker镜像的设计，使得Docker得以打破过去“程序即应用”的观念。通过镜像将作业系统核心除外，运行应用程序所需要的系统环境，由下而上打包，达到应用程序跨平台间的无缝接轨运作。

Docker是解决了运行环境和配置问题的软件容器，方便做持续集成并有助于整体发布的容器虚拟化技术。

2 Docker的概念

- Docker是一个开源的应用容器引擎。
- 诞生于2013年初，基于Go语言实现，dotCloud公司出品（后改名为Docker Inc）。
- Docker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。
- 容器是完全使用沙箱机制，相互隔离。
- 容器性能开销极低。
- Docker从17.03版本之后分为CE版（社区版）和EE版（企业版）。



Docker的安装

1 前提说明

- 目前，CentOS仅发行版中的内核支持Docker。
- Docker运行在CentOS7上，要求系统为64位，系统内核版本是3.10以上。
- Docker运行在CentOS6.5或更高版本的CentOS上，要求系统为64位，系统内核版本是2.6.32-431或者更高版本。
- 查看内核：

```
uname -r
```

```
[root@xuweiwei opt]# uname -r
2.6.32-754.25.1.el6.x86_64
[root@xuweiwei opt]#
```

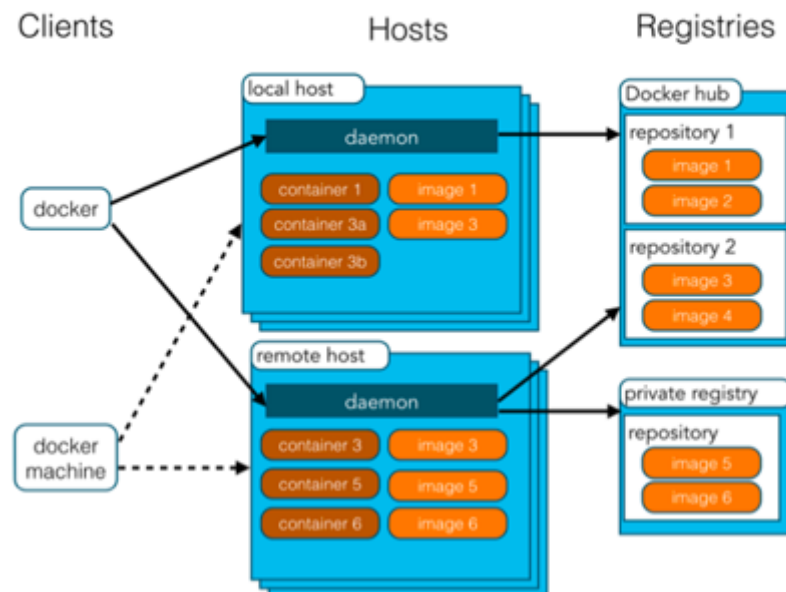
- 查看已安装的CentOS的版本信息：

```
cat /etc/redhat-release
```

```
[root@localhost ~]# cat /etc/redhat-release
CentOS Linux release 7.8.2003 (Core)
[root@localhost ~]#
```

2 Docker的组成

2.1 Docker的架构图



2.2 镜像 (image)

- Docker的镜像image就是一个只读的模板。镜像可以用来创建Docker的容器，一个镜像可以创建很多容器。

2.3 容器 (container)

- 镜像 (image) 和容器 (container) 的关系，就像是面向对象程序设计中的类和对象一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

2.4 仓库 (repository)

- 仓库repository是集中存放镜像文件的场所。
- 仓库repository和仓库注册服务器registry是有区别的。仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签tag。
- 仓库分为公开仓库和私有仓库两种形式。
- 最大的公开仓库是Docker Hub，存放了数量庞大的镜像供用户下载。
- 国内的公开仓库包括阿里云、网易云等。

2.5 总结

- Docker本身是一个容器运行载体或者称之为管理引擎。我们把应用程序和配置依赖打包好形成一个可交付的运行环境，这个打包好的运行环境就是image镜像文件。只有通过这个镜像文件才能生成Docker容器。image文件可以看做是容器的模板。Docker根据image文件生成容器的实例。同一个image文件，可以生成多个同时运行的容器实例。
- image文件生成的容器实例，本身也是一个文件，称为镜像文件。
- 一个容器运行一种服务，当我们需要的时候，就可以通过docker客户端创建一个对应的运行实例，也就是我们的容器。
- 至于仓库，就是放了一堆镜像的地方，我们可以把镜像发布到仓库中，需要的时候从仓库中拉下来就可以了。

3 Docker的安装

本次安装是在CentOS7版本上。

3.1 yum安装gcc相关

```
yum -y install gcc
yum -y install gcc-c++
```

```
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]#
```

3.2 卸载旧版本

```
sudo yum remove docker \
    docker-client \
    docker-client-latest \
    docker-common \
    docker-latest \
    docker-latest-logrotate \
    docker-logrotate \
    docker-engine
```

```
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]#
```

3.3 安装所需要的软件包

```
sudo yum install -y yum-utils \  
device-mapper-persistent-data \  
lvm2
```

```
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]#
```

3.4 设置stable镜像仓库

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

```
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]#
```

3.5 更新yum软件包索引

```
yum makecache fast
```

```
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]#
```

3.6 安装Docker

```
yum -y install docker-ce
```

```
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]#
```


3.7 启动Docker

```
systemctl start docker
```

```
[root@iZuf60x6sld5e5y4iy7c3oZ ~]#
```



3.8 验证docker是否安装成功

```
docker version
```

```
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]# docker version
Client: Docker Engine - Community
Version:      19.03.5
API version:  1.40
Go version:   go1.12.12
Git commit:   633a0ea
Built:        Wed Nov 13 07:25:41 2019
OS/Arch:      linux/amd64
Experimental: false

Server: Docker Engine - Community
Engine:
Version:      19.03.5
API version:  1.40 (minimum version 1.12)
Go version:   go1.12.12
Git commit:   633a0ea
Built:        Wed Nov 13 07:24:18 2019
OS/Arch:      linux/amd64
Experimental: false
containerd:
Version:      1.2.10
GitCommit:    b34a5c8af56e510852c35414db4c1f4fa6172339
runc:
Version:      1.0.0-rc8+dev
GitCommit:    3e425f80a8c931f88e6d94a8c831b9d5aa481657
docker-init:
Version:      0.18.0
GitCommit:    fec3683
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]#
```

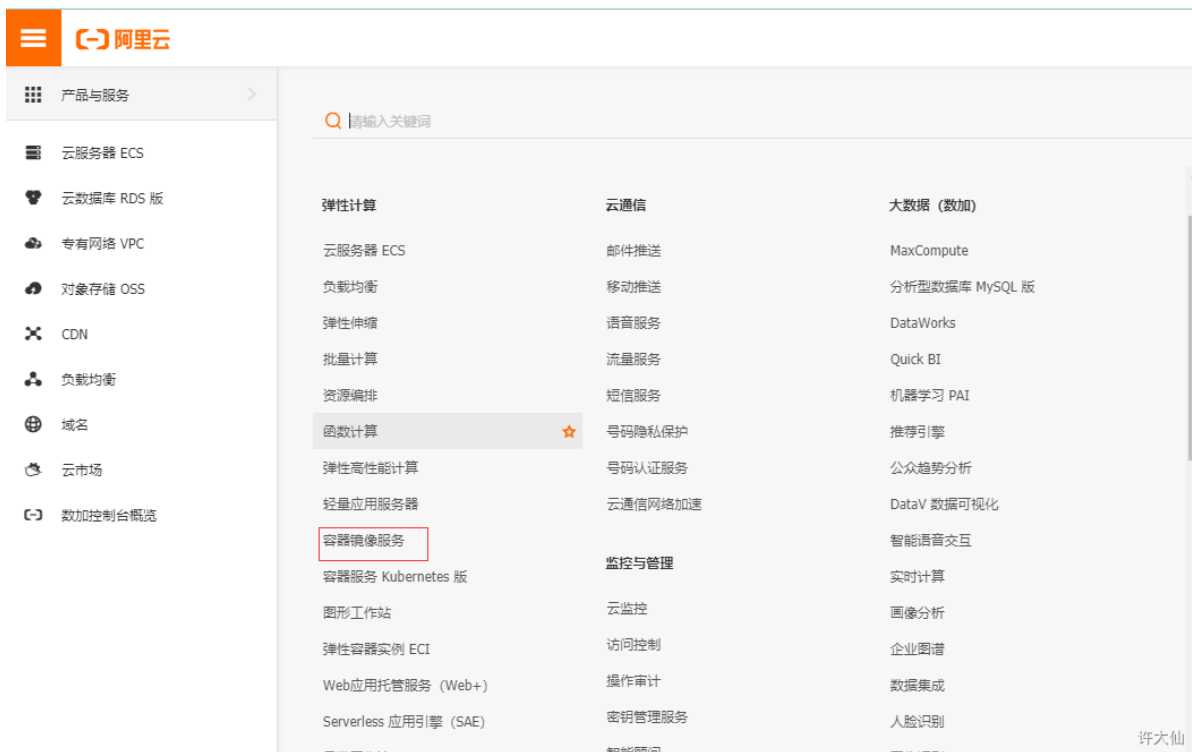
许大仙

4 卸载Docker

```
systemctl stop docke
yum -y remove docker-ce
rm -rf /var/lib/docker
```

5 配置阿里云镜像加速

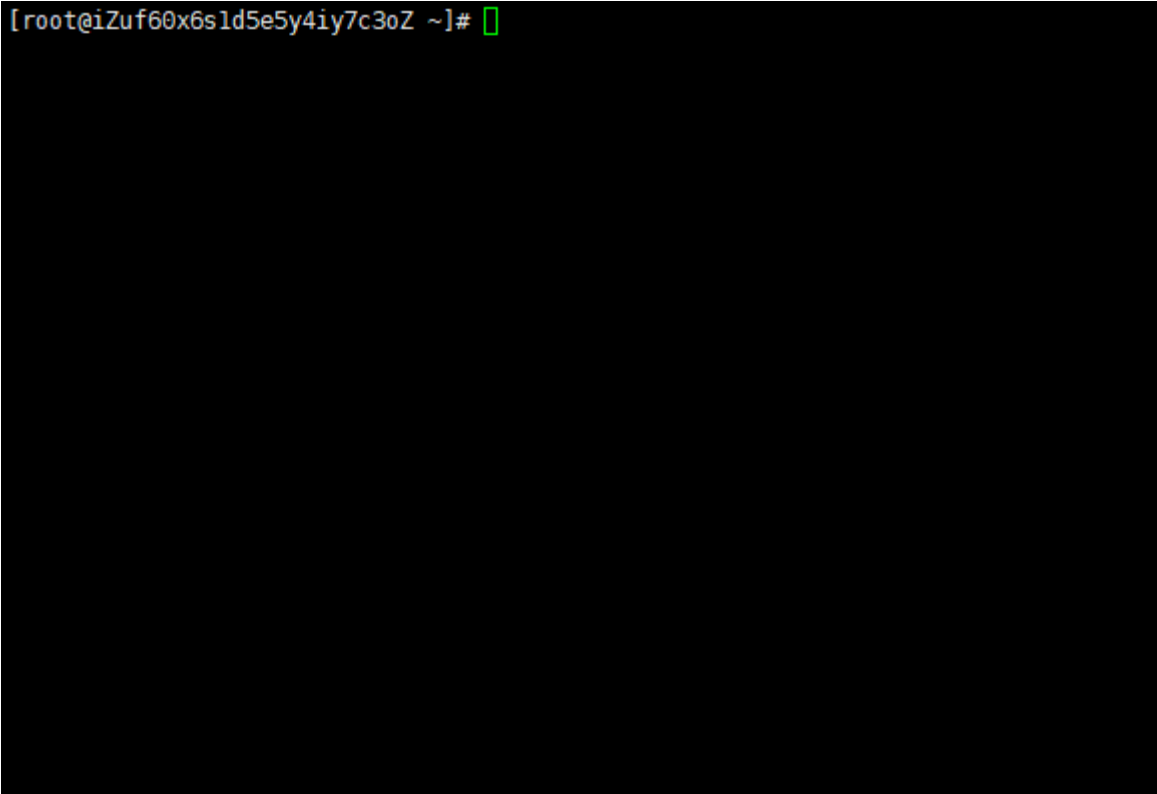
5.1 获取阿里云镜像加速地址



5.2 CentOS7.x下的Docker配置镜像加速

5.2.1 创建/etc/docker目录

```
mkdir -pv /etc/docker
```

A terminal window with a black background. The prompt is [root@iZuf60x6s1d5e5y4iy7c3oZ ~]# followed by a green cursor. The rest of the terminal area is empty.

```
[root@iZuf60x6s1d5e5y4iy7c3oZ ~]#
```

5.2.2 在/etc/docker目录下创建daemon.json文件来配置阿里云镜像加速，并将以下内容复制进去

```
{
  "registry-mirrors": ["自己的阿里云镜像加速地址"],
  "live-restore": true,
  "log-driver": "json-file",
  "log-opts": {"max-size": "500m", "max-file": "3"}
}
```

```
[root@iZuf60x6s1d5e5y4iy7c3oZ docker]#
```

live-restore: Docker重启之后，容器不退出。

log-driver和log-opts: 用来限制Docker容器的日志的大小。

max-size=500m, 意味着一个容器日志大小上限是500M。

max-file=3, 意味着一个容器有三个日志，分别是id+.json、id+1.json、id+2.json。

5.2.3 重启Docker服务

```
systemctl daemon-reload  
systemctl restart docker
```

```
[root@iZuf60x6s1d5e5y4iy7c3oZ docker]#
```

6 通过脚本安装

6.1 yum更新

```
yum -y update
```

6.2 脚本安装

```
curl -sSL https://get.docker.com/ | sh
```

6.3 启动Docker

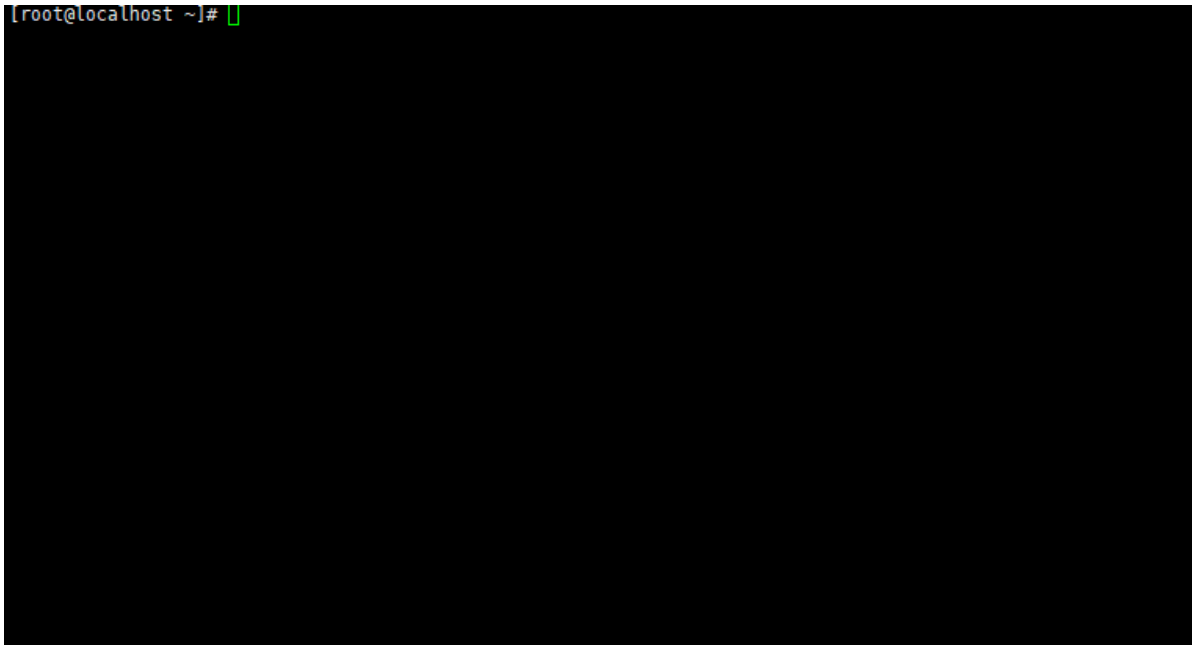
```
systemctl start docker
```

Docker命令

1 Docker进程相关命令

1.1 启动Docker服务

```
systemctl start docker
```



```
[root@localhost ~]#
```

1.2 停止Docker服务

```
systemctl stop docker
```

```
[root@localhost ~]#
```

1.3 重启Docker服务

```
systemctl restart docker
```

```
[root@localhost ~]#
```


1.4 查看Docker服务状态

```
systemctl status docker
```

```
[root@localhost ~]#
```

1.5 开机自启动Docker服务

```
systemctl enable docker
```

```
[root@localhost ~]#
```

2 镜像相关命令

2.1 搜索镜像

```
# 从网络上查找需要的镜像  
docker search 镜像名称
```

```
[root@localhost ~]#
```

2.2 拉取镜像

从Docker的仓库下载镜像到本地，镜像名称格式为名称:版本号，如果版本号不指定则是最新的版本。如果不知道镜像版本，可以去Docker Hub搜索对应镜像查看即可。

```
docker pull 镜像名称
```

```
[root@localhost ~]#
```

2.3 查看镜像

查看本地镜像

```
docker images
```

查看本地所有镜像

```
docker images -a
```

查看本地镜像的id

```
docker images -q
```

```
[root@localhost ~]#
```

2.4 删除镜像

```
# 删除镜像 -f表示强制删除  
docker rmi [-f] 镜像id[镜像名称]
```

```
# 删除所有镜像  
docker rmi -f $(docker images -qa)
```

```
[root@localhost ~]#
```

3 容器相关命令

3.1 查看运行的容器

```
# 查看正在执行的容器
docker ps
```

```
# 查看所有容器
docker ps -a
```

```
[root@localhost ~]#
```

3.2 创建并启动容器

```
docker run 参数
```

参数说明：

- i: 保持容器运行。通过和-t同时使用。加入-it这两个参数以后，容器创建后会自动进入容器中，退出容器后，容器会自动关闭。
- t: 为容器重新分配一个伪输入终端，通常和-i同时使用。
- d: 以守护（后台）模式运行容器。创建一个容器在后台运行，需要使用docker exec 进入容器。
- it: 创建的容器一般称为交互式容器。
- id: 创建的容器一般称为守护式容器、
- name: 为创建的容器命名。
- p: 映射端口 外部端口:容器内部暴露的端口

```
[root@localhost ~]#
```

3.3 进入容器

```
docker exec -it 容器id[容器名称] /bin/bash
```

```
[root@localhost ~]#
```

3.4 查看容器信息

```
docker inspect 容器id[容器名称]
```

```
[root@localhost ~]#
```

3.5 停止容器

```
docker stop 容器id[容器名称]
```

```
[root@localhost ~]#
```

3.6 启动容器

```
docker start 容器id[容器名称]
```

```
[root@localhost ~]#
```

3.7 重启容器

```
docker restart 容器id[容器名称]
```

```
[root@localhost ~]#
```

3.8 强制停止容器

```
docker kill 容器id[容器名称]
```



```
[root@localhost ~]#
```

3.9 删除容器

```
# 需要先停止容器，然后再删除  
docker rm 容器id[容器名称]
```

```
# 强制删除容器  
docker rm -f 容器id[容器名称]
```

```
# 强制删除所有容器  
docker rm -f $(docker ps -qa)
```

```
[root@localhost ~]#
```

3.10 查看容器日志

```
docker logs -f 容器id[容器名称]
```

```
[root@localhost ~]#
```

4 常用命令

attach	Attach to a running container	# 当前 shell 下 attach 连接指定运行镜像
build	Build an im 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]
run	Run a command in a new container	# 创建一个新的容器并运行一个命令

save	Save an image to a tar archive [对应 load]	# 保存一个镜像为一个 tar 包
search	Search for an image on the Docker Hub	# 在 docker hub 中搜索镜像
start	Start a stopped containers	# 启动容器
stop	Stop a running containers	# 停止容器
tag	Tag an image into a repository	# 给源中镜像打标签
top	Lookup the running processes of a container	# 查看容器中运行的进程信息
unpause	Unpause a paused container	# 取消暂停容器
version	Show the docker version information	# 查看 docker 版本号
wait	Block until a container stops, then print its exit code	# 截取容器停止时的退出状态值
age	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	

Docker容器的数据卷

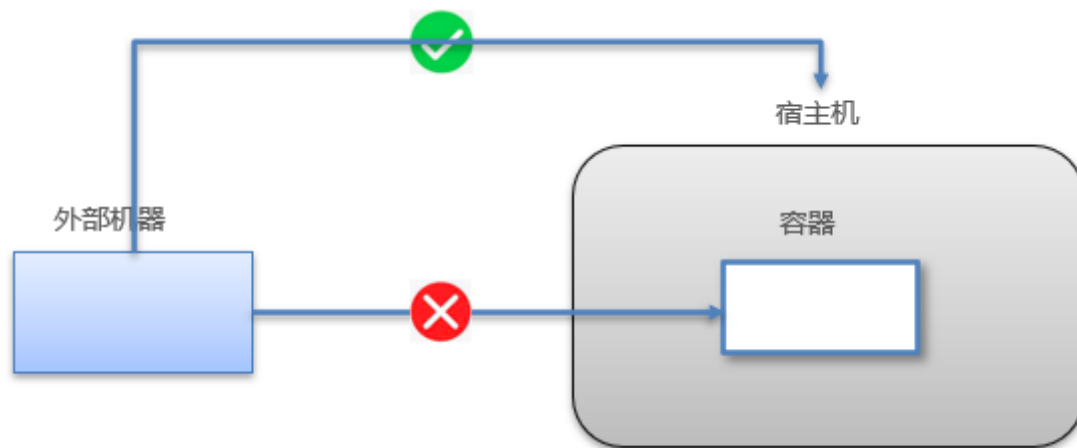
1 数据卷

1.1 思考

- Docker容器删除后，在容器中产生的数据还在吗？



- Docker容器和外部机器可以直接交换文件吗?

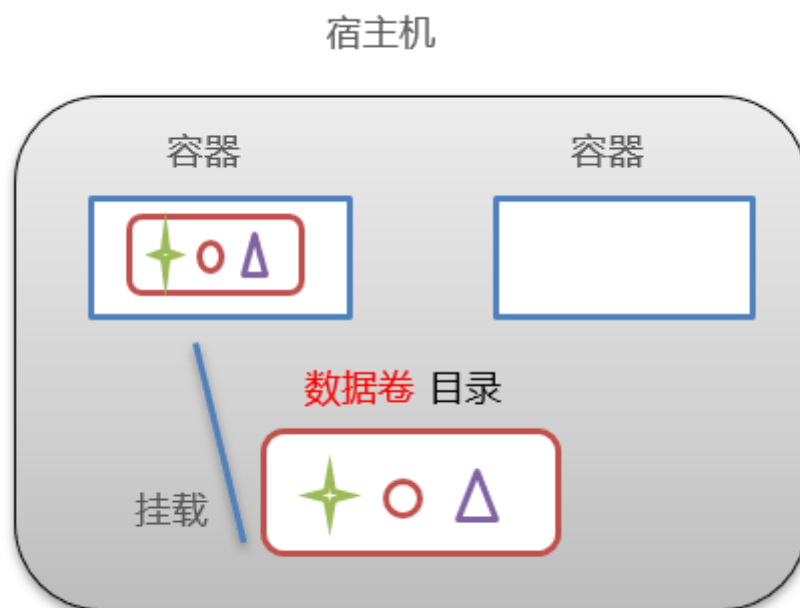


- 容器之间能进行数据交互?



1.2 数据卷概念

- 数据卷是宿主机中的一个目录或文件。
- 当容器目录和数据卷目录绑定后，对方修改会立即同步。
- 一个数据卷可以同时被多个容器同时挂载。
- 一个容器也可以被挂载多个数据卷。



1.3 数据卷作用

- 容器数据持久化。
- 外部机器和容器间接通信。
- 容器之间数据交换。

1.4 配置数据卷

1.4.1 命令

```
docker run ... -v 宿主机目录(文件):容器内目录(文件) ...
```

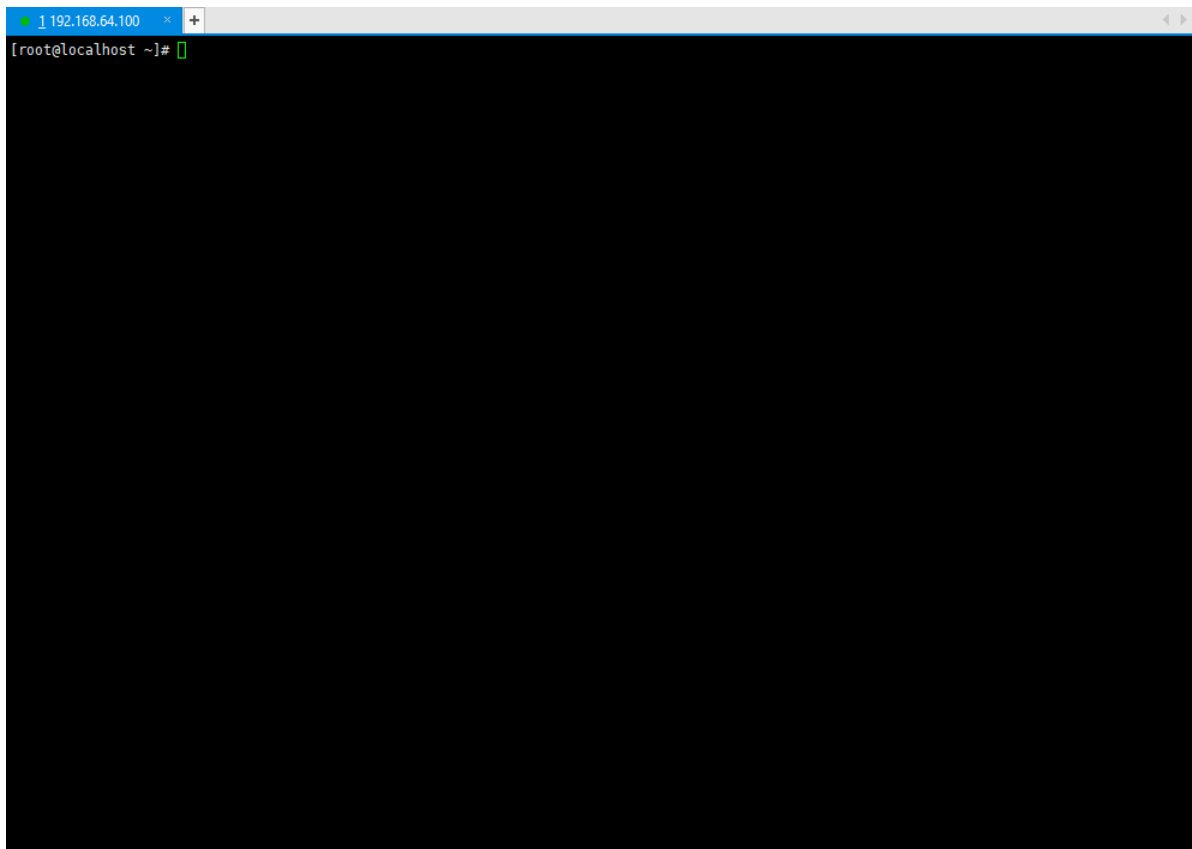
注意事项：

- ①目录必须是绝对路径。
- ②如果目录不存在，则会自动创建。
- ③可以挂载多个数据卷。

1.4.2 应用示例

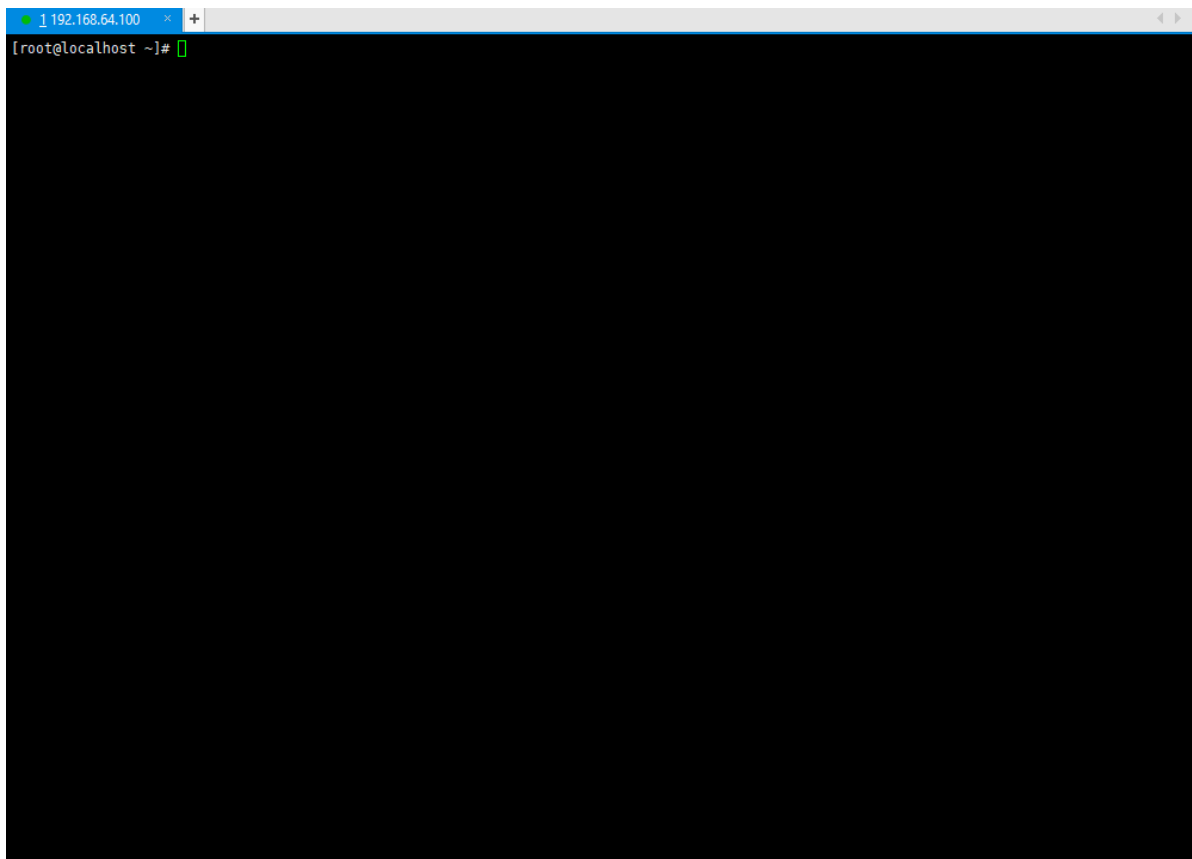
1.4.2.1 一个容器挂载一个数据卷

```
docker run -id --name c1 -v /root/data:/root/data_container centos:7
```



1.4.2.2 两个容器挂载同一个数据卷

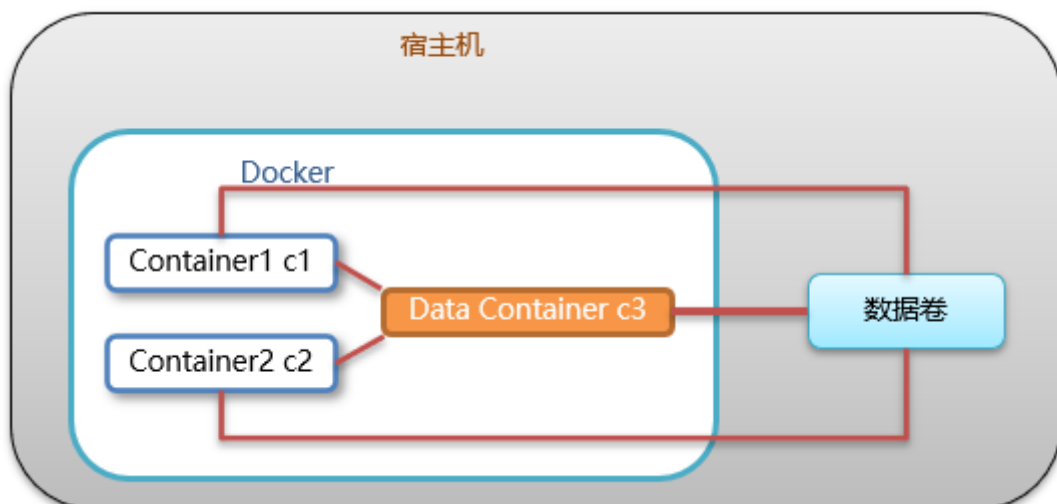
```
docker run -id --name c1 -v /root/data:/root/data_container centos:7  
docker run -id --name c2 -v /root/data:/root/data_container centos:7
```



2 数据卷容器

2.1 概念

- 多容器进行数据交换。
 - 多个容器挂载同一个数据卷。
 - 数据卷容器。



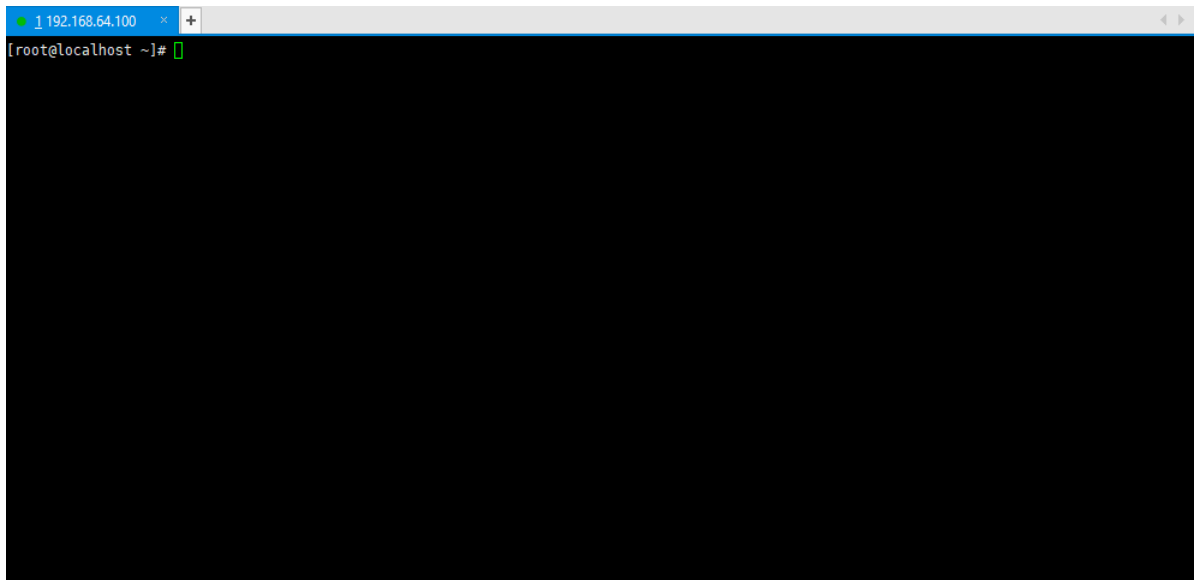
2.2 配置数据卷容器

- 创建启动c3数据卷容器，使用-v参数设置数据卷。

```
docker run -id -v /volume --name c3 centos:7
```

- 创建启动c1、c2容器，使用--volumes-from参数设置数据卷。

```
docker run -id --volumes-from c3 --name c1 centos:7  
docker run -id --volumes-from c3 --name c2 centos:7
```



Docker应用部署

1 Docker安装MySQL

1.1 需求

- 在Docker容器中部署MySQL，并通过外部MySQL客户端操作MySQL服务器。

1.2 实现步骤

- ①搜索MySQL镜像。
- ②拉取MySQL镜像。

- ③创建容器。
- ④操作容器中的MySQL。

1.3 应用示例

1.3.1 搜索MySQL镜像

```
docker search mysql
```

```
[root@localhost ~]#
```

1.3.2 拉取MySQL镜像

```
docker pull mysql:5.7
```

```
[root@localhost ~]#
```

1.3.3 创建容器，设置端口映射、目录映射

```
docker run -id -p 3306:3306 --name mysql5.7 -v  
/var/mysql5.7/conf:/etc/mysql/conf.d -v /var/mysql5.7/logs:/logs -v  
/var/mysql5.7/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 --restart=always  
mysql:5.7 --lower_case_table_names=1
```

```
[root@localhost ~]#
```

2 Docker安装Tomcat

2.1 需求

- 在Docker容器中部署Tomcat，并通过外部机器访问Tomcat部署的项目。

2.2 实现步骤

- ①搜索Tomcat镜像。
- ②拉取Tomcat镜像。
- ③创建容器。
- ④部署项目。
- ⑤测试访问。

2.3 应用示例

2.3.1 搜索Tomcat镜像

```
docker search tomcat
```

```
[root@localhost ~]#
```

2.3.2 拉取tomcat镜像

```
docker pull tomcat
```

```
[root@localhost ~]#
```

2.3.3 创建容器，设置端口映射、目录映射

```
docker run -id --name tomcat -p 8080:8080 -v /var/tomcat:/usr/local/tomcat/webapps tomcat
```

```
[root@localhost ~]#
```

3 Docker安装Nginx

3.1 需求

- 在Docker容器中部署Nginx，并通过外部机器访问Nginx。

3.2 实现步骤

- ①搜索Nginx镜像。
- ②拉取Nginx镜像。
- ③创建容器。
- ④测试访问。

3.3 应用示例

3.3.1 搜索Nginx镜像

```
docker search nginx
```

```
[root@localhost ~]#
```

3.3.2 拉取Nginx镜像

```
docker pull nginx
```

```
[root@localhost ~]#
```

3.3.3 创建容器，设置端口映射、目录映射

- nginx的配置文件

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include        /etc/nginx/mime.types;
    default_type   application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;
```



```
    keepalive_timeout 65;

    #gzip on;

    include /etc/nginx/conf.d/*.conf;
}
```

```
[root@localhost ~]#
```

```
docker run -id --name=nginx \
-p 80:80 \
-v /var/nginx/conf/nginx.conf:/etc/nginx/nginx.conf \
-v /var/nginx/logs:/var/log/nginx \
-v /var/nginx/html:/usr/share/nginx/html \
nginx
```

```
[root@localhost ~]#
```

4 Docker安装Redis

4.1 需求

- 在Docker容器中部署Redis，并通过外部机器访问Redis。

4.2 实现步骤

- ①搜索Redis镜像。
- ②拉取Redis镜像。
- ③创建容器。
- ④测试访问。

4.3 应用示例

4.3.1 搜索Redis镜像

```
docker search redis
```

```
[root@localhost ~]#
```

4.3.2 拉取Redis镜像

```
docker pull redis:5.0
```

```
[root@localhost ~]#
```

4.3.3 创建容器，设置端口映射

```
docker run -id --name redis5.0 -p 6379:6379 redis:5.0
```

```
[root@localhost ~]#
```

Dockerfile

1 镜像原理

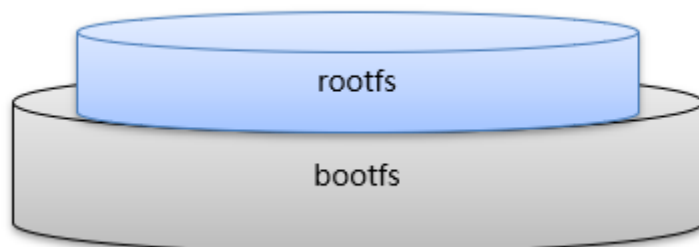
1.1 思考

- Docker镜像的本质是什么？
- Docker中一个CentOS镜像为什么只有200MB，而一个CentOS操作系统的iso文件要几个G？
- Docker中的一个Tomcat镜像为什么有500MB，而一个Tomcat安装包只有70多MB。

1.2 Linux文件系统

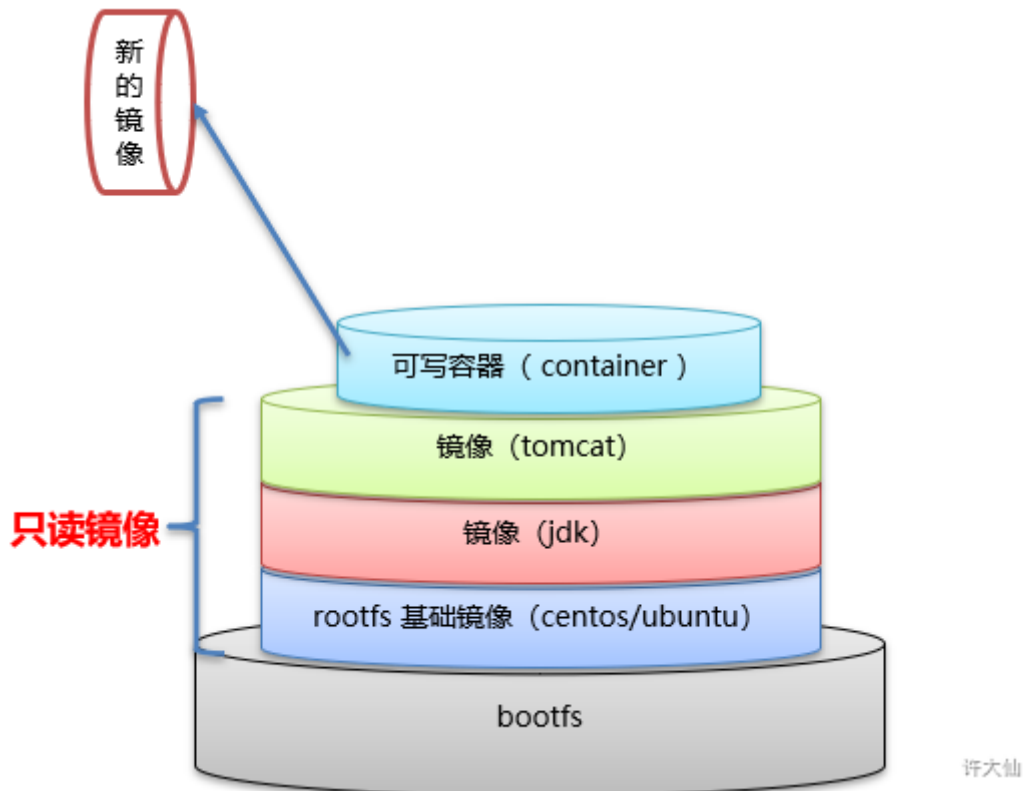
- 操作系统组成部分：
 - 进程调度子系统。
 - 进程通信子系统。
 - 内存管理子系统。
 - 设备管理子系统。
 - 文件管理子系统。
 - 网络通信子系统。
 - 作业控制子系统。

- Linux文件系统由bootfs和rootfs两部分组成。
 - bootfs: 包含bootloader（引导加载程序）和kernel（内核）。
 - rootfs: root文件系统，包含的就是典型的Linux系统中的/dev, /proc, /etc等标准目录和文件。
 - 不同的Linux发行版，bootfs基本一样，而rootfs不同，如Ubuntu和CentOS等。



1.3 Docker镜像原理

- Docker镜像是由特殊的文件系统叠加而成。
- 最低端是bootfs，并使用宿主机的bootfs。
- 第二层是root文件系统的rootfs，称为base image。
- 然后再往上可以叠加其他的镜像文件。
- 统一文件系统（Union File System）技术能够将不同的层整合成一个文件系统，为这些层提供一个统一的视角，这样就隐藏了多层的存在，在用户的角度来看，只存在一个文件系统。
- 一个镜像可以放在另一个镜像的上面。位于下面的镜像称为父镜像，最底部的镜像称为基础镜像。
- 当从一个镜像启动容器时，Docker会在最顶层加载一个读写文件系统作为容器。



1.4 总结

- 问：Docker镜像的本质是什么？
- 答：是一个分层文件系统。
- 问：Docker中一个CentOS镜像为什么只有200MB，而一个CentOS操作系统的iso文件要几个G？
- 答：CentOS的iso镜像包含bootfs和rootfs，而Docker的centos镜像复用操作系统的bootfs，只有rootfs和其它镜像层。
- 问：Docker中的一个Tomcat镜像为什么有500MB，而一个Tomcat安装包只有70多MB？
- 答：Docker中镜像是分层的，Tomcat虽然只有70多MB，但他需要依赖于父镜像和基础镜像，所以整个队外暴露的Tomcat镜像大小约500多MB。

2 镜像制作

2.1 容器转为镜像

- 容器转为镜像

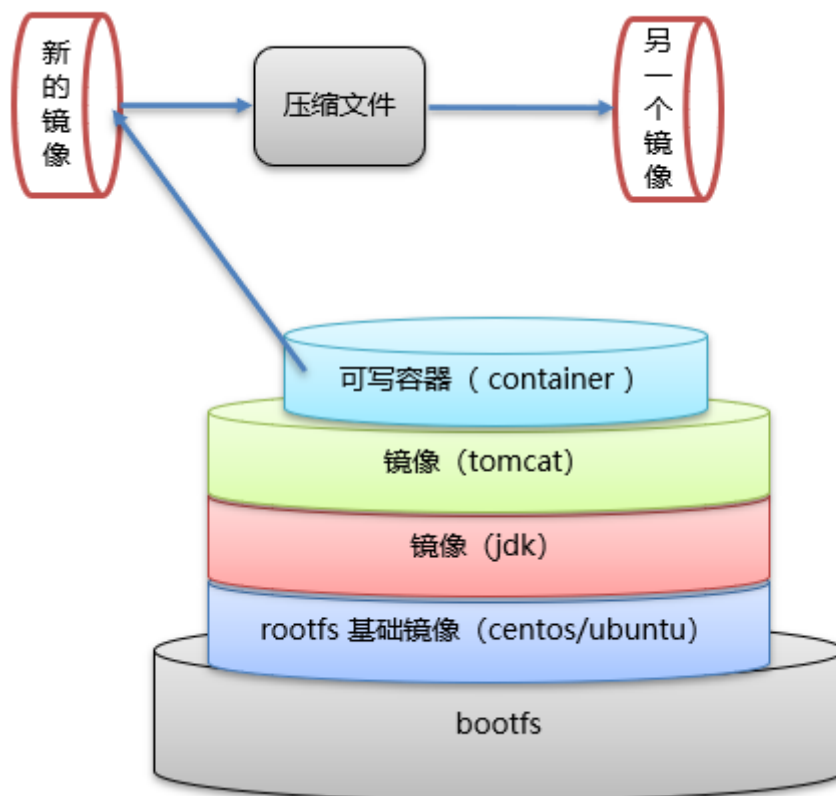
```
docker commit 容器id 镜像名称:版本号
```

- 将镜像压缩

```
docker save -o 压缩文件名称 镜像名称:版本号
```

- 将压缩文件还原为镜像

```
docker save -i 压缩文件名称
```



许大仙

2.2 Dockerfile

2.2.1 概念

- Dockerfile是一个文本文件。
- 包含了一条条的指令。
- 每一条指令构建一层，基于基础镜像，最终构建一个新的镜像。
- 对于开发人员，可以为开发团队提供一个完全一致的开发环境。

- 对于测试人员，可以直接拿开发时所构建的镜像或者通过Dockerfile文件构建一个新的镜像。
- 对于运维人员，在部署的时候，可以实现应用的无缝移植。

2.2.2 Dockerfile指令

关键字	作用	备注
FROM	指定父镜像	指定Dockerfile基于那个image构建
MAINTAINER	作者信息	用来标明这个Dockerfile谁写的
LABEL	标签	用来标明Dockerfile的标签，可以使用Label代替Maintainer，最终都是在docker image基本信息中可以查看
RUN	执行命令	执行一段命令，默认是/bin/sh 格式: RUN command 或者 RUN ["command", "param1","param2"]
CMD	容器启动命令	提供启动容器时候的默认命令 和ENTRYPOINT配合使用.格式 CMD command param1 param2 或者 CMD ["command", "param1","param2"]
ENTRYPOINT	入口	一般在制作一些执行就关闭的容器中会使用
COPY	复制文件	build的时候复制文件到image中
ADD	添加文件	build的时候添加文件到image中 不仅仅局限于当前build上下文可以来源于远程服务
ENV	环境变量	指定build时候的环境变量 可以在启动的容器的时候 通过-e覆盖 格式ENV name=value
ARG	构建参数	构建参数 只在构建的时候使用的参数 如果有ENV 那么ENV的相同名字的值始终覆盖arg的参数
VOLUME	定义外部可以挂载的数据卷	指定build的image那些目录可以启动的时候挂载到文件系统中 启动容器的时候使用 -v 绑定 格式 VOLUME ["目录"]
EXPOSE	暴露端口	定义容器运行的时候监听的端口 启动容器的使用-p来绑定暴露端口 格式: EXPOSE 8080 或者 EXPOSE 8080/udp
WORKDIR	工作目录	指定容器内部的工作目录 如果没有创建则自动创建 如果指定/使用的是绝对地址 如果不是/开头那么是在上一条workdir的路径的相对路径
USER	指定执行用户	指定build或者启动的时候 用户 在RUN CMD ENTRYPOINT执行的时候的用户
HEALTHCHECK	健康检查	指定监测当前容器的健康监测的命令 基本上没用 因为很多时候应用本身有健康监测机制
ONBUILD	触发器	当存在ONBUILD关键字的镜像作为基础镜像的时候 当执行FROM完成之后 会执行 ONBUILD的命令 但是不影响当前镜像用处也不怎么大
STOPSIGNAL	发送信号量到宿主机	该STOPSIGNAL指令设置将发送到容器的系统调用信号以退出。

关键字	作用	备注
SHELL	指定执行脚本的shell	指定RUN CMD ENTRYPOINT 执行命令的时候 使用的shell

2.2.3 自定义CentOS镜像

- Dockerfile

```
# 定义父镜像
FROM centos:7
# 定义作者信息
MAINTAINER weiwei.xu <1900919313@qq.com>
# 执行安装vim命令
RUN yum -y install vim
# 定义默认的工作目录
WORKDIR /usr
# 定义容器启动执行的命令
CMD /bin/bash
```

- 通过Dockerfile构建镜像

```
docker build -f Dockerfile文件路径 -t 镜像名称:版本号 镜像存放的绝对路径
```

- 示例:

```
[root@localhost ~]#
```

```
[root@localhost ~]#
```

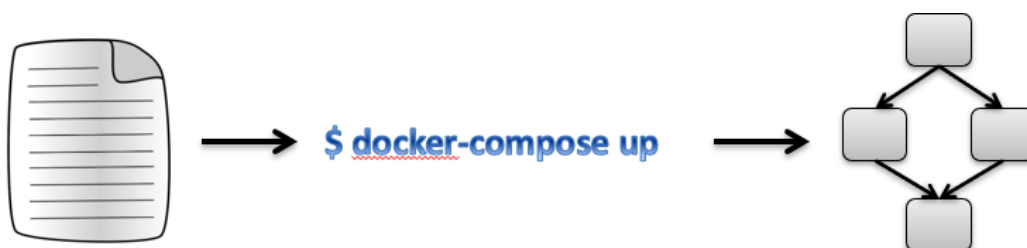
Docker服务编排（已过时）

1 服务编排概念

- 微服务架构的应用系统中一般包含若干个微服务，每个微服务一般都会部署多个实例，如果每个微服务都要手动启停，维护的工作量会很大。
 - 要从Docker build image 或者去 Docker Hub拉取image。
 - 要创建多个container。
 - 要管理这些container（启动、停止、删除等）。
- 服务编排：按照一定的业务规则批量管理容器。

2 Docker Compose概念

- Docker Compose是一个编排多容器分布式部署的工具，提供命令管理容器化应用的完整开发周期，包括服务创建构建、启动和停止。
- 使用步骤如下：
 - ①利用Dockerfile定义运行环境镜像。
 - ②使用docker-compose.yml定义组成应用的各个服务。
 - ③运行docker-compose up启动应用。



3 Docker Compose的安装和卸载

3.1 Docker Compose的安装

```
# Compose目前已经完全支持Linux、Mac OS和Windows，在我们安装Compose之前，需要先安装
Docker。下面我们 以编译好的二进制包方式安装在Linux系统中。
curl -L https://github.com/docker/compose/releases/download/1.26.2/docker-
compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
# 国内的地址
curl -L https://get.daocloud.io/docker/compose/releases/download/1.26.2/docker-
compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
# 设置文件可执行权限
chmod 777 /usr/local/bin/docker-compose
# 创建软链接
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
# 查看版本信息
docker-compose -version
```

3.2 Docker Compose的卸载

```
# 二进制包方式安装的，删除二进制文件即可  
rm /usr/local/bin/docker-compose
```

4 Docker Compose的应用示例

- 新建一个SpringBoot项目，随便建。
- 编写Dockerfile文件。

```
# 指定基础镜像，在其上进行定制  
FROM java:8  
#这里的 /tmp 目录就会在运行时自动挂载为匿名卷，任何向 /tmp 中写入的信息都不会记录进容器存储层  
VOLUME /tmp  
# 指定在创建容器后，终端默认登陆进来的工作目录，一个落脚点  
WORKDIR /  
#复制上下文目录下的/target/demo-1.0.jar到容器里，并将文件名称修改为demo.jar  
ADD /target/demo-1.0.jar demo.jar  
#bash方式执行，使robot.jar可访问  
#RUN新建一层，在其上执行这些命令，执行结束后， commit 这一层的修改，构成新的镜像。  
RUN bash -c "touch /demo.jar"  
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && echo  
'Asia/Shanghai' >/etc/timezone  
#声明运行时容器提供服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务  
EXPOSE 8080  
#指定容器启动程序及参数    <ENTRYPOINT> "<CMD>"tail  
ENTRYPOINT ["java","-Dfile.encoding=UTF-8","-  
Djava.security.egd=file:/dev/./urandom","-jar","/demo.jar"]
```

- 使用docker build构建镜像。

```
docker build -t demo
```

- 新建/var/nginx/nginx.conf.d目录，并新建nginx.conf配置文件：

```
server {  
    listen      80;  
    server_name 192.168.64.100;  
  
    location / {  
        proxy_pass http://app:8080/;  
    }  
  
    error_page 500 502 503 504 /50x.html;  
    location = /50x.html {  
        root    html;  
    }  
}
```

- 编写docker-compose.yml。

```
version: '3'  
services:  
  nginx:  
    image: nginx  
    ports:  
      - "80:80"  
    links:  
      - app  
    volumes:  
      - "/var/nginx/nginx.conf.d:/etc/nginx/conf.d"  
  app:  
    image: demo
```

- 使用如下命令启动：

```
docker-compose up -d
```

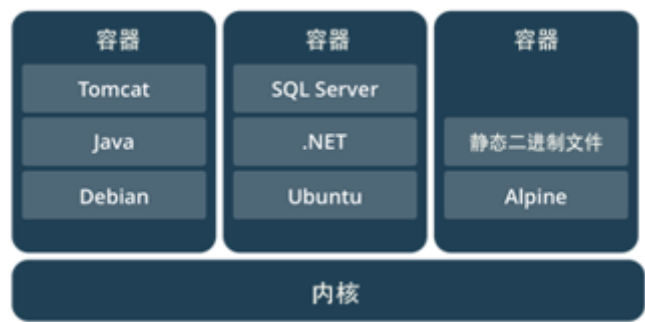
- 访问：<http://192.168.64.100>。

Docker相关概念

1 容器

- 容器就是将软件打包成标准化单元，以便开发、交付和部署。
- 容器镜像是轻量的、可执行的独立软件包，包含软件运行所需要的所有内容：代码、运行时环境、系统工具、系统库和设置。
- 容器化软件在任何环境中都能够始终如一地运行。

- 容器赋予了软件独立性，使其免受外在环境差异的影响，从而有助于减少团队建设在相同基础设施上运行不同软件时的冲突。



2 Docker容器虚拟化 VS 传统虚拟机

- 相同：容器和虚拟机具有相似的资源隔离和分配优势。
- 不同：
 - 容器虚拟化的是操作系统，虚拟机虚拟化是硬件。
 - 传统虚拟机可以运行不同的操作系统，容器只能运行同一类型操作系统。



Docker Swarm（已过时）

1 Docker Swarm介绍

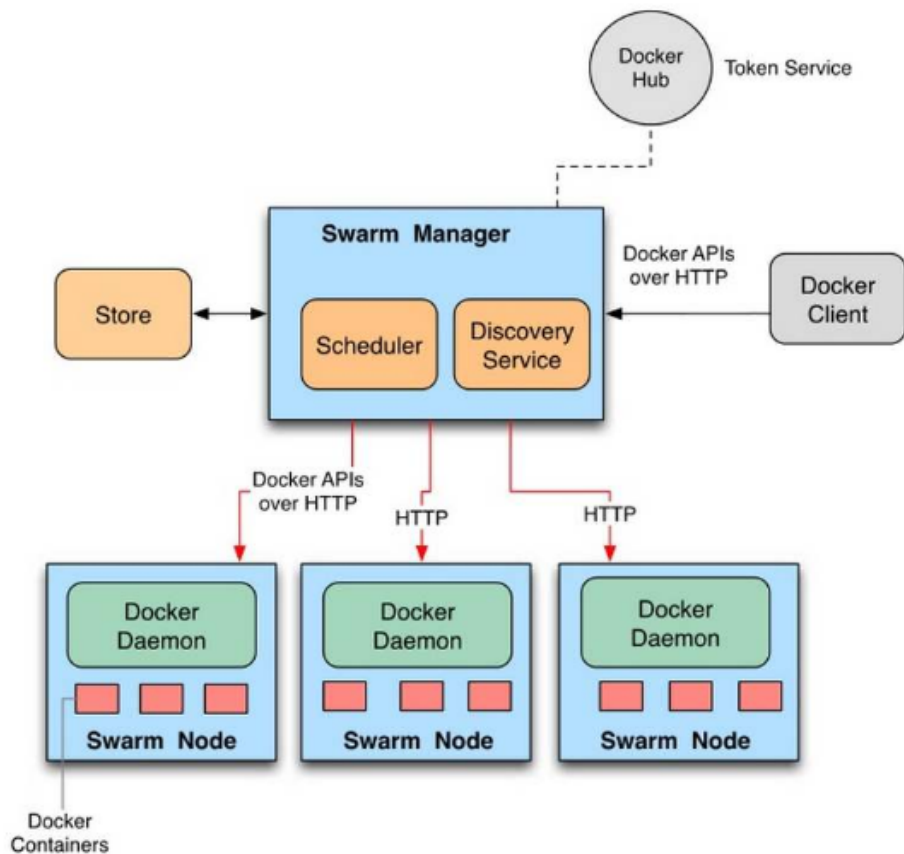
- 在Wiki的解释中，Swarm behavior是指动物的群集行为。比如我们常见的蜂群、鱼群，秋天往南飞的燕群都可以称作为Swarm behavior。Swarm项目正是如此，通过把多个Docker Engine聚集在一起，形成一个大的Docker Engine，对外提供容器的集群服务。同时这个集群对外提供Swarm API，用户可以像使用Docker Engine一样使用Docker集群。
- Swarm是Docker公司在2014年12月初发布的容器管理工具，和Swarm一起发布的Docker管理工具还有Machine以及Compose。Swarm是一套较为简单的工具，用以管理Docker集群，使得Docker集群暴露给用户时相当于一个虚拟的整体。Swarm将一群Docker宿主机变成一个单一的、虚拟的主机。Swarm使用标准的Docker API接口作为其前端访问入口，换言之，各种形式的Docker Client均可以直接和Swarm直接通信。Swarm几乎全部用Go语言完成开发，Swarm0.2版本增加了一个新的策略来调度集群中的容器，使得在可用的节点上传播它们以及支持更多的Docker命令以及集群驱动。Swarm daemon只是一个调度器（Scheduler）加上路由器（Router），Swarm自己不运行容器，它只是接受Docker客户端发送过来的请求，调度适合的节点来运行容器，这意味着，即使Swarm由于某些原因挂掉了，集群中的节点也会照样运行，当Swarm恢复运行之后，它会收集重建集群信息。

2 Docker Swarm特点

- 对外以Docker API接口呈现，这样带来的好处是，如果现有系统使用Docker Engine，则可以平滑将Docker Engine切到Swarm上，无需改动现有系统。
- Swarm对用户来说，之前使用Docker的经验可以继承过来。非常容易上手，学习成本和二次开发成本都比较低。同时Swarm本身专注于Docker集群管理，非常轻量，占用资源也非常少。简单的说，就是插件化机制，Swarm中的各个模块都抽象出了API，可以根据自己的一些特点进行定制实现。
- Swarm自身对Docker命令参数支持的比较完善，Swarm目前和Docker是同步发布的。Docker的新功能都会第一时间在Swarm中体现。

3 Docker Swarm架构

- Swarm作为一个管理Docker集群的工具，首先需要将其部署起来，可以单独将Swarm部署于一个节点。另外，自然需要一个Docker集群，集群上每一个节点均安装Docker。



许大仙

4 Docker Swarm的使用

4.1 环境准备

- 准备三台已经安装Docker Engine的CentOS系统主机（Docker的版本必须是1.12以上的版本，老版本不支持Swarm）。
- Docker容器主机的ip地址固定，集群中所有工作节点必须能访问该管理节点。
- 集群管理节点必须使用相应的协议并且保证端口可用。

集群管理通信：TCP，端口2377
节点通信：TCP和UDP，端口7946
覆盖型网络：UDP，端口4789

备注：

192.168.219.100 管理节点
192.168.219.101 工作节点
192.168.219.102 工作节点

4.2 Docker Swarm集群搭建实现

注意：在测试时，最好关闭防火墙。

- 在192.168.219.100机器上创建Docker Swarm集群。

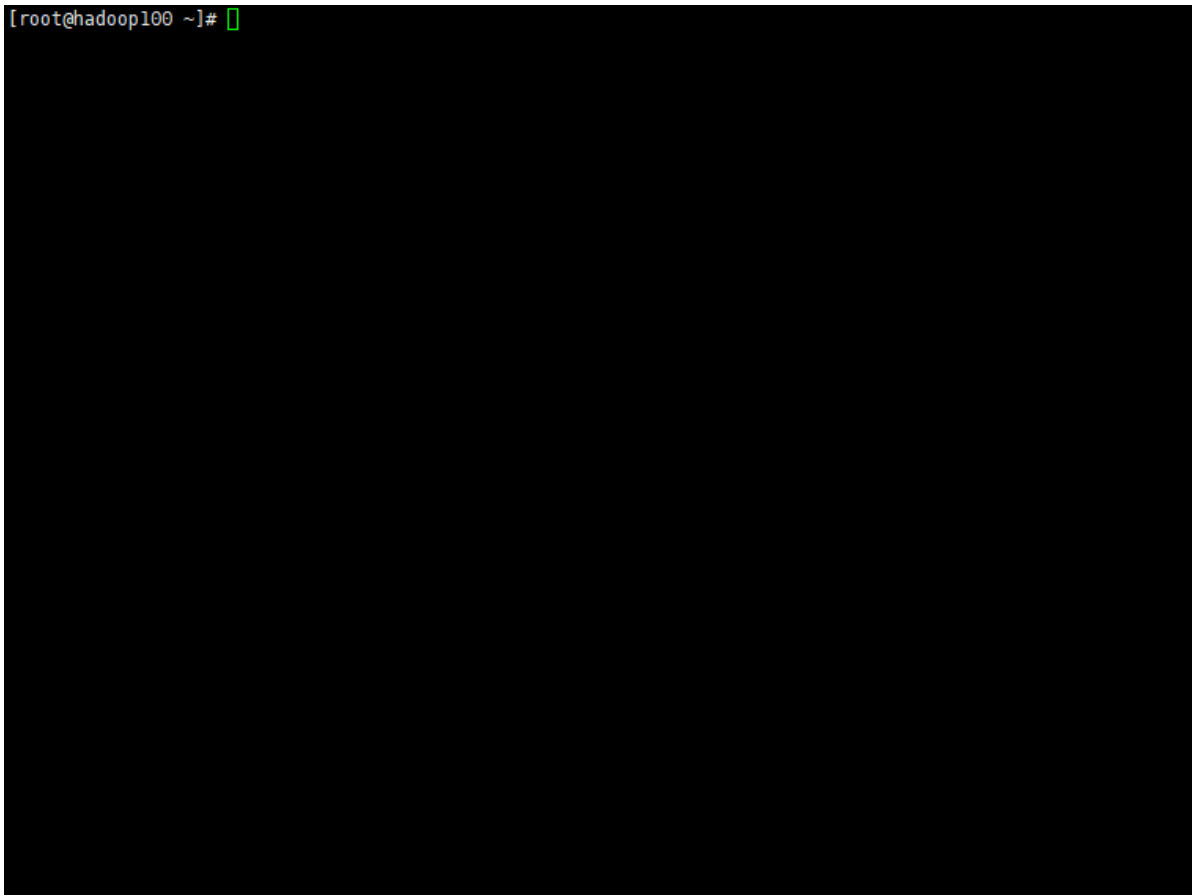
```
docker swarm init --advertise-addr 192.168.219.100
```

备注：

--advertise-addr将该IP地址的机器设置为集群管理节点，如果是单节点，无需该参数

- 查看管理节点集群信息：

```
docker node ls
```



```
[root@hadoop100 ~]#
```

- 向Docker Swarm中添加工作节点：在两个工作节点中分别执行如下的命令，ip地址是manager节点的

```
# 在192.168.219.101中执行如下的命令
docker swarm join --token SWMTKN-1-
1b8cgc1f61a9j7a84voptpfm15afnmfp8hpcizvichqid5ko11-cf094t5ymy0hzaharru9ocoty
192.168.219.100:2377
```

备注:

--token xxx: 向指定集群中加入工作节点的认证信息, xxx认证信息是在创建Docker Swarm时产生的

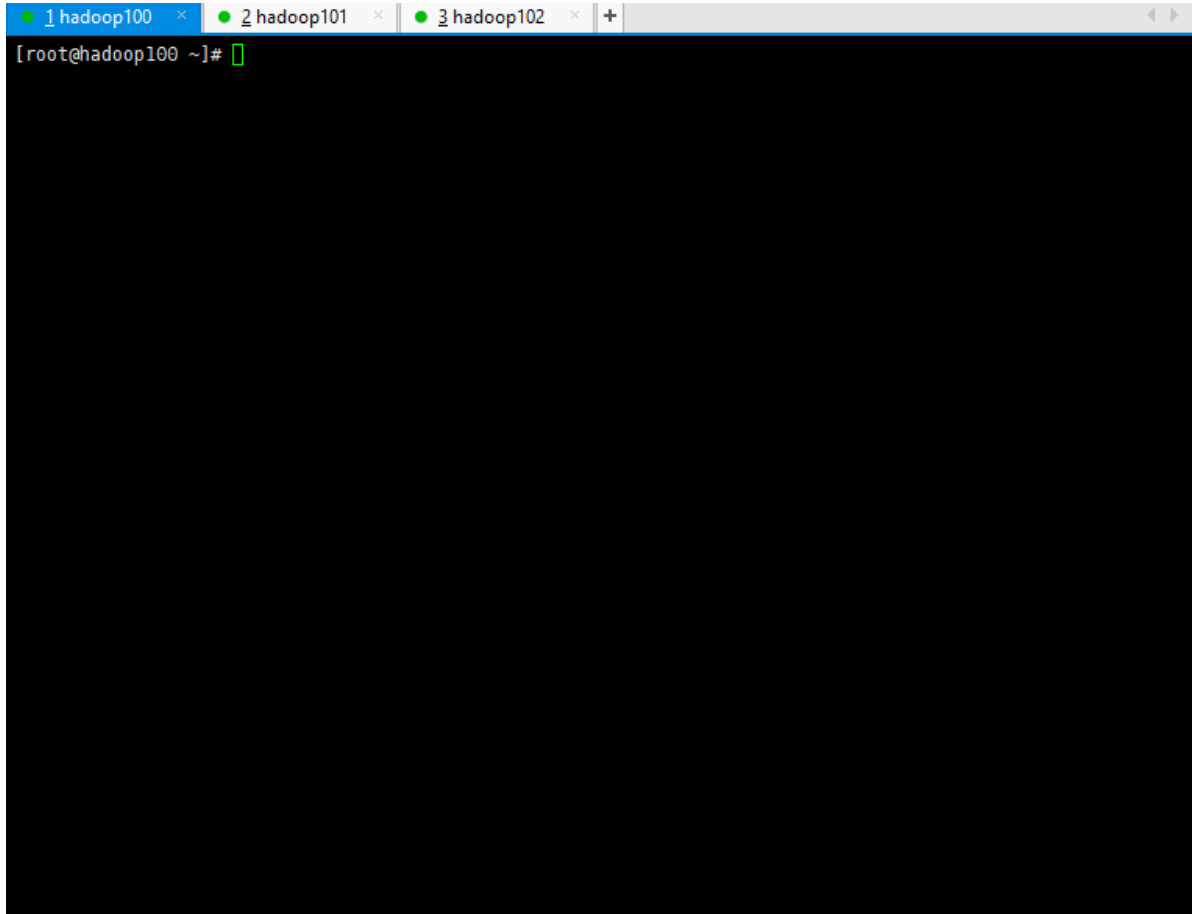
```
# 在192.168.219.102中执行如下的命令
docker swarm join --token SWMTKN-1-
1b8cgc1f61a9j7a84voptpfm15afnmfp8hpcizvichqid5ko11-cf094t5ymy0hzaharru9ocoty
192.168.219.100:2377
```

备注:

--token xxx: 向指定集群中加入工作节点的认证信息, xxx认证信息是在创建Docker Swarm时产生的

- 查看管理节点集群信息:

```
docker node ls
```



4.3 Docker Swarm中部署alpine服务

4.3.1 部署服务

```
docker service create --replicas 1 --name helloworld alpine ping docker.com
```

备注:

`docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]`: 用于在Swarm集群中创建一个基于alpine镜像的服务

`--replicas num`: 指定了该服务只有一个副本实例

`--name 服务名称`: 创建成功后的服务名称

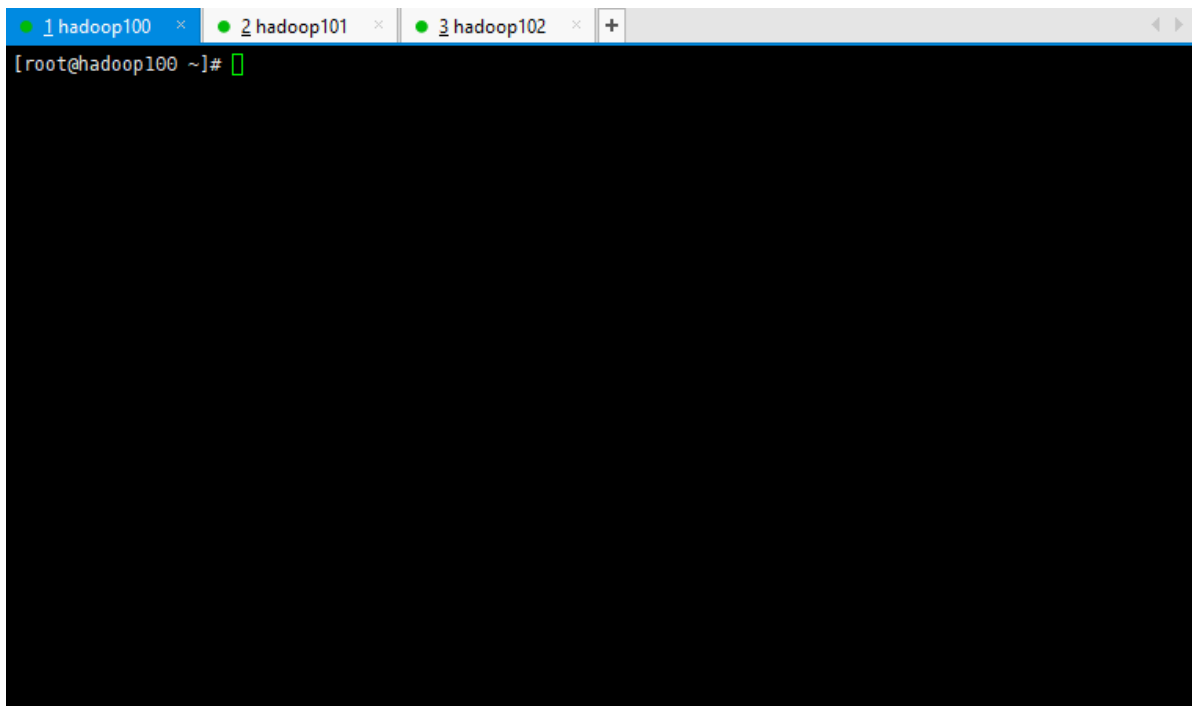
`ping docker.com`: 表示服务启动执行的命令

```
[root@hadoop100 ~]# []
```

4.3.2 查看服务

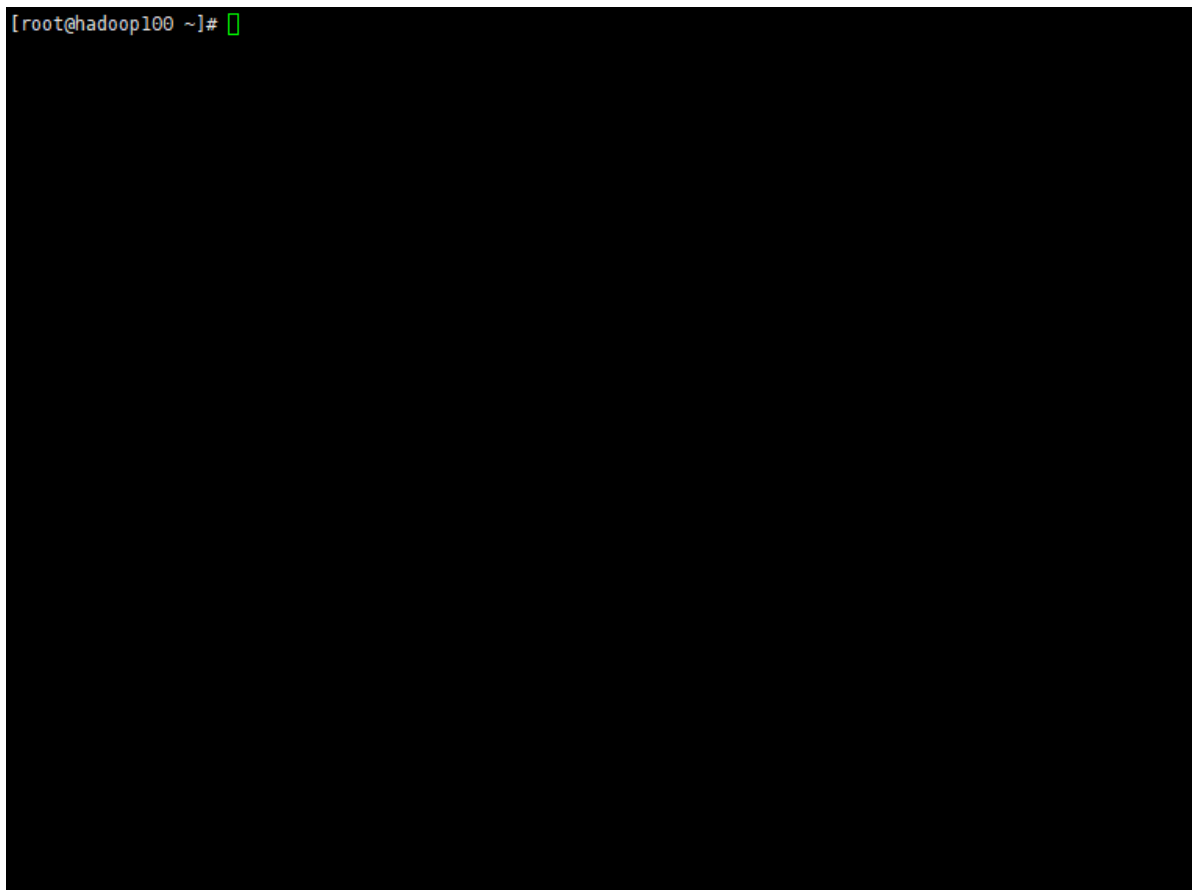
- 查看服务列表:

```
docker service ls
```



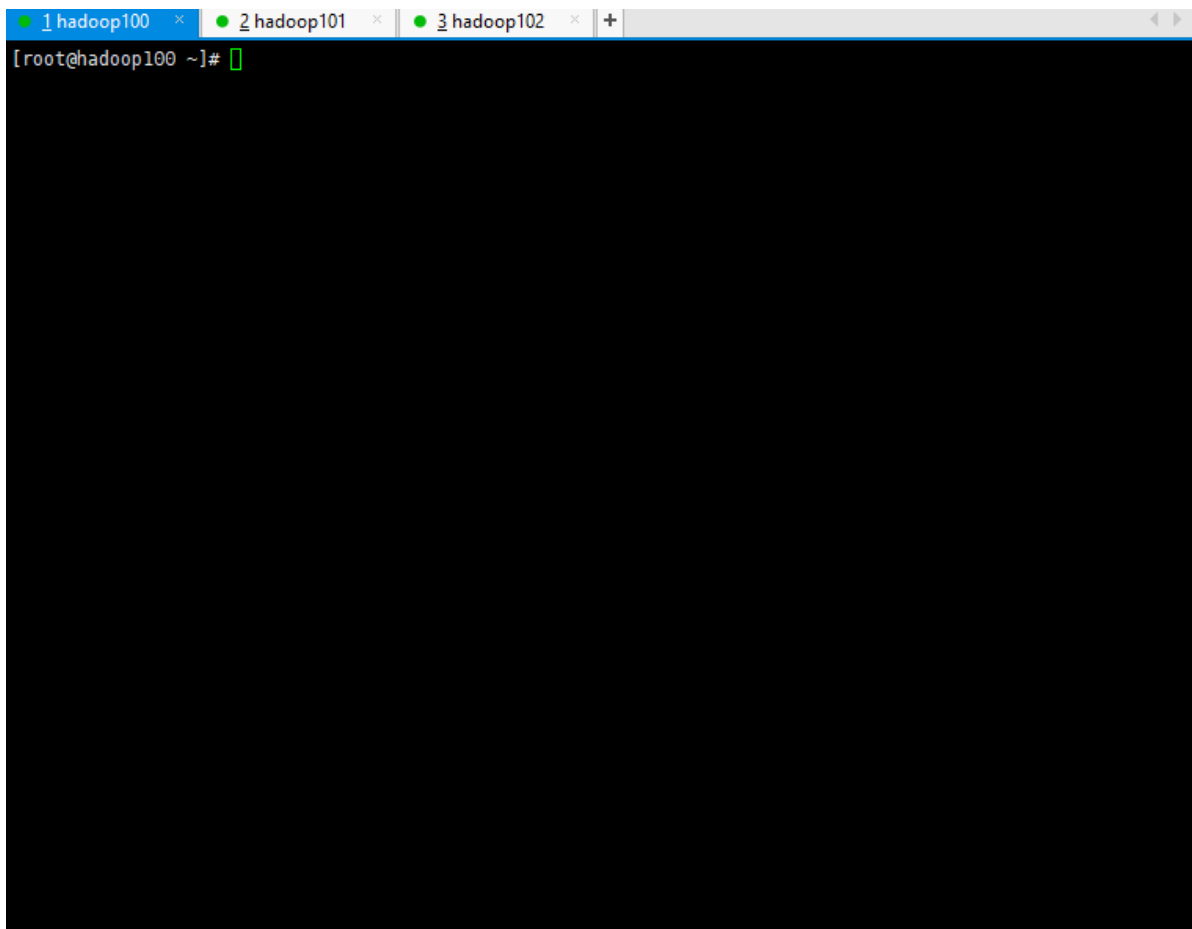
- 查看部署具体服务的详细信息：

```
docker service inspect 服务名称
```



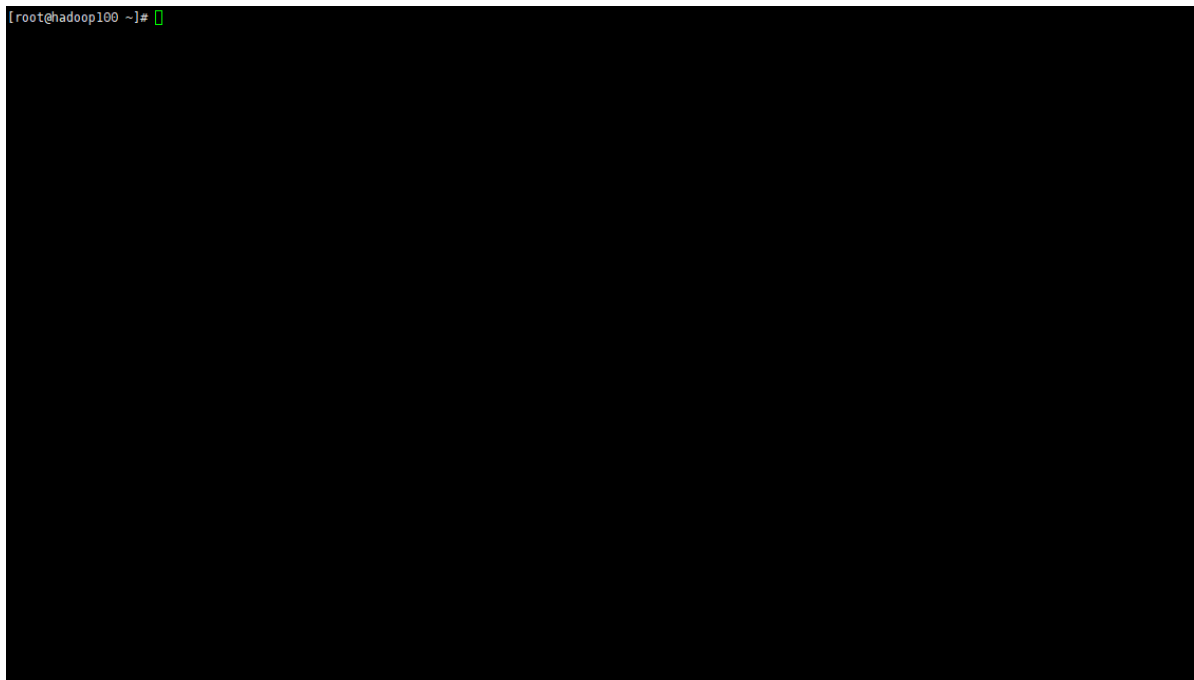
- 查看服务在集群节点上的分配以及运行情况：

```
docker service ps 服务名称
```



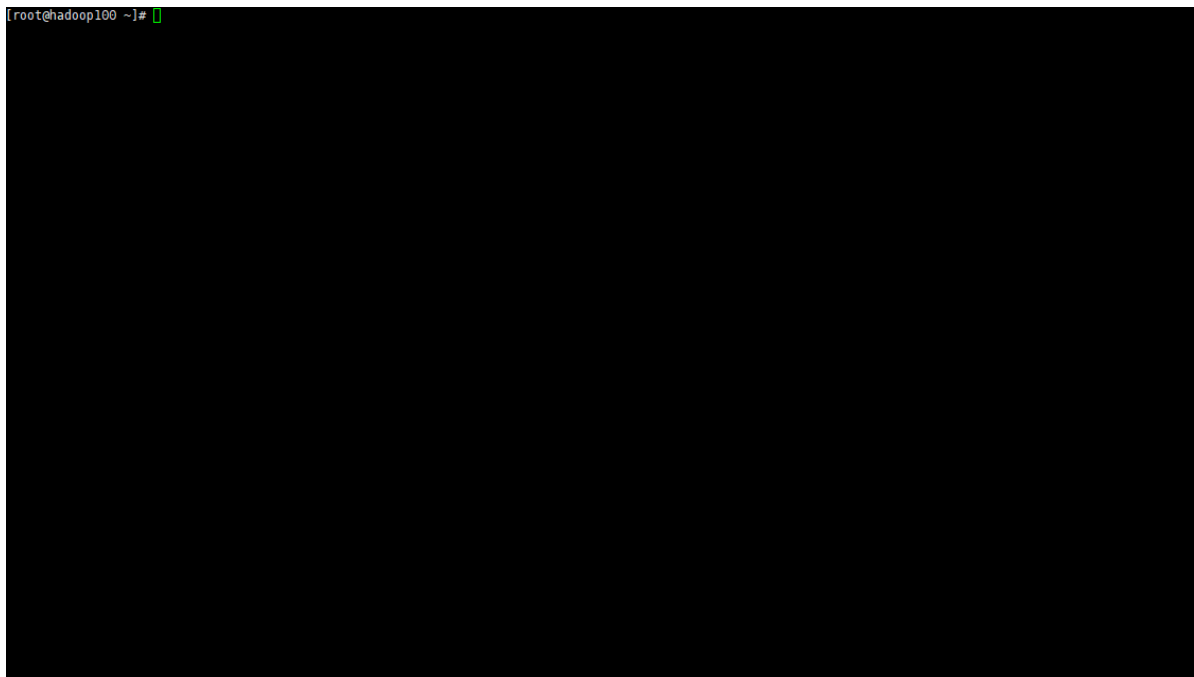
4.3.3 修改副本数量

```
docker service scale helloworld=5
```



4.3.4 删除服务

```
docker service rm 服务名称
```



Docker仓库

1 官方仓库构建

1.1 仓库服务器的配置

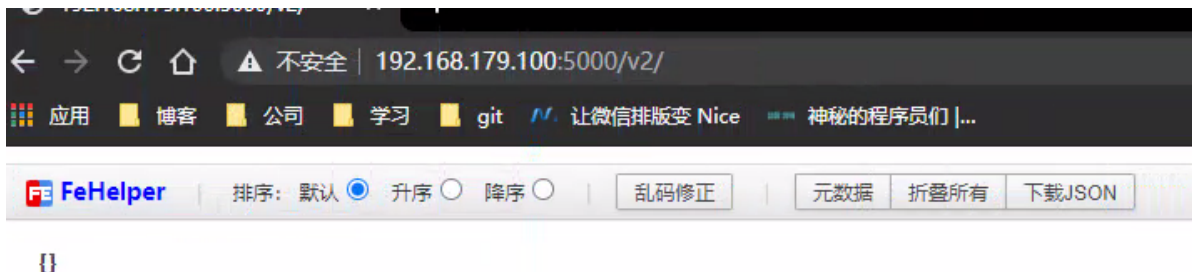
```
docker run -d -v /var/docker/registry:/var/lib/registry -p 5000:5000 --restart=always registry
```

```
vim /etc/docker/daemon.json
{
  "registry-mirrors": ["https://du3ia00u.mirror.aliyuncs.com"],
  "insecure-registries": ["192.168.179.100:5000"]
}
systemctl daemon-reload
systemctl restart docker
```

192.168.179.100是仓库服务器的地址。

- 验证是否安装成功：

```
http://192.168.179.100:5000/v2/
```



- 从Docker Hub上拉取镜像：

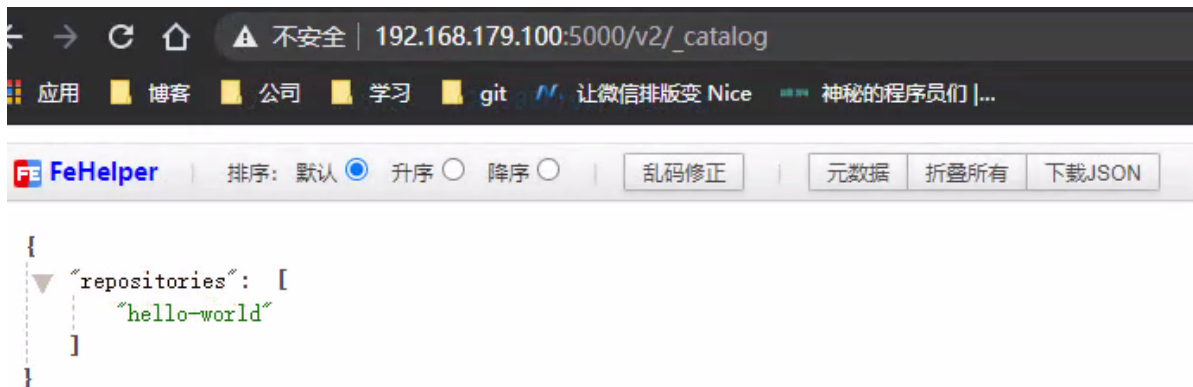
```
docker pull hello-world
```

- 将拉取到本地的镜像改为ip地址:端口号/用户名/镜像名称:标签：

```
docker tag hello-world:latest 192.168.179.100:5000/hello-world:v1.0
```

- 推送镜像到本地仓库:

```
docker push 192.168.179.100:5000/hello-world:v1.0
```



1.2 客户端设置

```
vim /etc/docker/daemon.json
{
  "registry-mirrors": ["https://du3ia00u.mirror.aliyuncs.com"],
  "insecure-registries": ["192.168.179.100:5000"]
}
systemctl daemon-reload
systemctl restart docker
```

```
docker pull 192.168.179.100:5000/hello-world:v1.0
```

2 Harbor仓库构建

2.1 安装底层需求

- Python应该为2.7或更高版本。
- Docker引擎应该为1.10或更高版本。

- Docker Compose应该为1.6.0或更高版本。

2.2 Harbor的安装

2.2.1 解压压缩包

```
wget https://github.com/goharbor/harbor/releases/download/v1.2.0/harbor-offline-installer-v1.2.0.tgz
```

```
tar -zxvf harbor-offline-installer-v1.2.0.tgz
```

```
[root@localhost ~]# tar -zxvf harbor-offline-installer-v1.2.0.tgz
harbor/common/templates/
harbor/common/templates/ui/
harbor/common/templates/ui/private_key.pem
harbor/common/templates/ui/env
harbor/common/templates/ui/app.conf
harbor/common/templates/clair/
harbor/common/templates/clair/config.yaml
harbor/common/templates/clair/postgresql-init.d/
harbor/common/templates/clair/postgresql-init.d/README.md
harbor/common/templates/clair/postgres_env
harbor/common/templates/nginx/
harbor/common/templates/nginx/notary_server.conf
harbor/common/templates/nginx/notary_upstream.conf
harbor/common/templates/nginx/nginx_https.conf
harbor/common/templates/nginx/nginx_http.conf
harbor/common/templates/db/
harbor/common/templates/db/env
harbor/common/templates/adminserver/
harbor/common/templates/adminserver/env
harbor/common/templates/jobservice/
harbor/common/templates/jobservice/env
harbor/common/templates/jobservice/app.conf
harbor/common/templates/notary/
harbor/common/templates/notary/signer-config.json
harbor/common/templates/notary/notary-signer.key
harbor/common/templates/notary/mysql-initdb.d/
harbor/common/templates/notary/mysql-initdb.d/initial-notaryserver.sql
harbor/common/templates/notary/mysql-initdb.d/initial-notarysigner.sql
harbor/common/templates/notary/signer_env
harbor/common/templates/notary/server-config.json
harbor/common/templates/notary/notary-signer-ca.crt
harbor/common/templates/notary/notary-signer.crt
harbor/common/templates/registry/
harbor/common/templates/registry/root.crt
harbor/common/templates/registry/config.yml
harbor/harbor.v1.2.0.tar.gz
harbor/prepare
harbor/NOTICE
harbor/upgrade
harbor/harbor_1_1_0_template
harbor/LICENSE
harbor/install.sh
harbor/harbor.cfg
harbor/docker-compose.yml
harbor/docker-compose.notary.yml
harbor/docker-compose.clair.yml
```

许大仙

2.2.2 移动harbor到/usr/local目录下

```
mv harbor /usr/local
```

```

[root@localhost ~]# ll
总用量 479008
drwxr-xr-x. 2 root root    4096  8月 25 10:36 公共
drwxr-xr-x. 2 root root    4096  8月 25 10:36 模板
drwxr-xr-x. 2 root root    4096  8月 25 10:36 视频
drwxr-xr-x. 2 root root    4096  8月 25 10:36 图片
drwxr-xr-x. 2 root root    4096  8月 25 10:36 文档
drwxr-xr-x. 2 root root    4096  8月 25 10:36 下载
drwxr-xr-x. 2 root root    4096  8月 25 10:36 音乐
drwxr-xr-x. 2 root root    4096  8月 25 10:36 桌面
-rw-r--r--. 1 root root     978  8月 25 10:28 anaconda-ks.cfg
drwxr-xr-x. 3 root root    4096  9月  9 14:28 harbor
-rw-r--r--. 1 root root 490451083  9月  9 14:27 harbor-offline-installer-v1.2.0.tgz
-rw-r--r--. 1 root root    1221  8月 25 10:36 initial-setup-ks.cfg
[root@localhost ~]# mv harbor /usr/local/
[root@localhost ~]# cd /usr/local/
[root@localhost local]# ll
总用量 44
drwxr-xr-x. 2 root root 4096  9月  9 08:42 bin
drwxr-xr-x. 2 root root 4096  1月 29 2020 etc
drwxr-xr-x. 2 root root 4096  1月 29 2020 games
drwxr-xr-x. 3 root root 4096  9月  9 14:28 harbor
drwxr-xr-x. 2 root root 4096  1月 29 2020 include
drwxr-xr-x. 2 root root 4096  1月 29 2020 lib
drwxr-xr-x. 3 root root 4096  4月 23 06:36 lib64
drwxr-xr-x. 2 root root 4096  1月 29 2020 libexec
drwxr-xr-x. 2 root root 4096  1月 29 2020 sbin
drwxr-xr-x. 5 root root 4096  4月 23 06:36 share
drwxr-xr-x. 2 root root 4096  1月 29 2020 src
[root@localhost local]#

```

许大仙

2.2.3 创建https证书以及配置相关目录权限

```
openssl genrsa -des3 -out server.key 2048
```

```

[root@localhost ~]# openssl genrsa -des3 -out server.key 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
Enter pass phrase for server.key:
Verifying - Enter pass phrase for server.key:
[root@localhost ~]#

```

输入密码

```
openssl req -new -key server.key -out server.csr
```

```
[root@localhost ~]# openssl req -new -key server.key -out server.csr
Enter pass phrase for server.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [XX]:CN
State or Province Name (full name) []:BJ
Locality Name (eg, city) [Default City]:BJ
Organization Name (eg, company) [Default Company Ltd]:sunxiaping
Organizational Unit Name (eg, section) []:sunxiaping
Common Name (eg, your name or your server's hostname) []:hub.sunxiaping.com
Email Address []:1900919313@qq.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

直接Enter即可

许大仙

```
cp server.key server.key.org
```

```
[root@localhost ~]# ll
总用量 479012
drwxr-xr-x. 2 root root    4096  8月 25 10:36 公共
drwxr-xr-x. 2 root root    4096  8月 25 10:36 模板
drwxr-xr-x. 2 root root    4096  8月 25 10:36 视频
drwxr-xr-x. 2 root root    4096  8月 25 10:36 图片
drwxr-xr-x. 2 root root    4096  8月 25 10:36 文档
drwxr-xr-x. 2 root root    4096  8月 25 10:36 下载
drwxr-xr-x. 2 root root    4096  8月 25 10:36 音乐
drwxr-xr-x. 2 root root    4096  8月 25 10:36 桌面
-rw-r--r--. 1 root root      978  8月 25 10:28 anaconda-ks.cfg
-rw-r--r--. 1 root root 490451083  9月  9 14:27 harbor-offline-installer-v1.2.0.tgz
-rw-r--r--. 1 root root    1221  8月 25 10:36 initial-setup-ks.cfg
-rw-r--r--. 1 root root    1062  9月  9 14:36 server.csr
-rw-r--r--. 1 root root    1751  9月  9 14:33 server.key
[root@localhost ~]# cp server.key server.key.org
[root@localhost ~]# ll
总用量 479016
drwxr-xr-x. 2 root root    4096  8月 25 10:36 公共
drwxr-xr-x. 2 root root    4096  8月 25 10:36 模板
drwxr-xr-x. 2 root root    4096  8月 25 10:36 视频
drwxr-xr-x. 2 root root    4096  8月 25 10:36 图片
drwxr-xr-x. 2 root root    4096  8月 25 10:36 文档
drwxr-xr-x. 2 root root    4096  8月 25 10:36 下载
drwxr-xr-x. 2 root root    4096  8月 25 10:36 音乐
drwxr-xr-x. 2 root root    4096  8月 25 10:36 桌面
-rw-r--r--. 1 root root      978  8月 25 10:28 anaconda-ks.cfg
-rw-r--r--. 1 root root 490451083  9月  9 14:27 harbor-offline-installer-v1.2.0.tgz
-rw-r--r--. 1 root root    1221  8月 25 10:36 initial-setup-ks.cfg
-rw-r--r--. 1 root root    1062  9月  9 14:36 server.csr
-rw-r--r--. 1 root root    1751  9月  9 14:33 server.key
-rw-r--r--. 1 root root    1751  9月  9 14:37 server.key.org
[root@localhost ~]#
```

许大仙

```
openssl rsa -in server.key.org -out server.key
```

```
[root@localhost ~]# openssl rsa -in server.key.org -out server.key
Enter pass phrase for server.key.org:
writing RSA key
[root@localhost ~]#
```

```
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

```
[root@localhost ~]# openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
Signature ok
subject=C = CN, ST = BJ, L = BJ, O = sunxiaping, OU = sunxiaping, CN = hub.sunxiaping.com, emailAddress = 1900919313@qq.com
Getting Private key
[root@localhost ~]#
```

```
mkdir -pv /data/cert
```

```
[root@localhost ~]# mkdir -pv /data/cert
mkdir: 已创建目录 '/data'
mkdir: 已创建目录 '/data/cert'
[root@localhost ~]#
```

```
chmod -R 777 /data/cert
```

```
[root@localhost ~]# chmod -R 777 /data/cert
[root@localhost ~]#
```

```
mv server.* /data/cert/
```

```
root@localhost ~]# mv server.* /data/cert/
root@localhost ~]#
```

2.2.4 修改harbor.cfg配置文件

```
cd /usr/local/harbor
```

```
vim harbor.cfg
```

```
hostname = hub.sunxiaping.com  
ui_url_protocol = https
```

```
[root@localhost ~]# ll  
总用量 479008  
drwxr-xr-x. 2 root root    4096 8月 25 10:36 公共  
drwxr-xr-x. 2 root root    4096 8月 25 10:36 模板  
drwxr-xr-x. 2 root root    4096 8月 25 10:36 视频  
drwxr-xr-x. 2 root root    4096 8月 25 10:36 图片  
drwxr-xr-x. 2 root root    4096 8月 25 10:36 文档  
drwxr-xr-x. 2 root root    4096 8月 25 10:36 下载  
drwxr-xr-x. 2 root root    4096 8月 25 10:36 音乐  
drwxr-xr-x. 2 root root    4096 8月 25 10:36 桌面  
-rw-r--r--. 1 root root     978 8月 25 10:28 anaconda-ks.cfg  
drwxr-xr-x. 3 root root    4096 9月 9 14:28 harbor  
-rw-r--r--. 1 root root 490451083 9月 9 14:27 harbor-offline-installer-v1.2.0.tgz  
-rw-r--r--. 1 root root    1221 8月 25 10:36 initial-setup-ks.cfg  
[root@localhost ~]# mv harbor /usr/local/  
[root@localhost ~]# cd /usr/local/  
[root@localhost local]# ll  
总用量 44  
drwxr-xr-x. 2 root root 4096 9月 9 08:42 bin  
drwxr-xr-x. 2 root root 4096 1月 29 2020 etc  
drwxr-xr-x. 2 root root 4096 1月 29 2020 games  
drwxr-xr-x. 3 root root 4096 9月 9 14:28 harbor  
drwxr-xr-x. 2 root root 4096 1月 29 2020 include  
drwxr-xr-x. 2 root root 4096 1月 29 2020 lib  
drwxr-xr-x. 3 root root 4096 4月 23 06:36 lib64  
drwxr-xr-x. 2 root root 4096 1月 29 2020 libexec  
drwxr-xr-x. 2 root root 4096 1月 29 2020 sbin  
drwxr-xr-x. 5 root root 4096 4月 23 06:36 share  
drwxr-xr-x. 2 root root 4096 1月 29 2020 src  
[root@localhost local]#
```

许大仙

2.2.5 安装

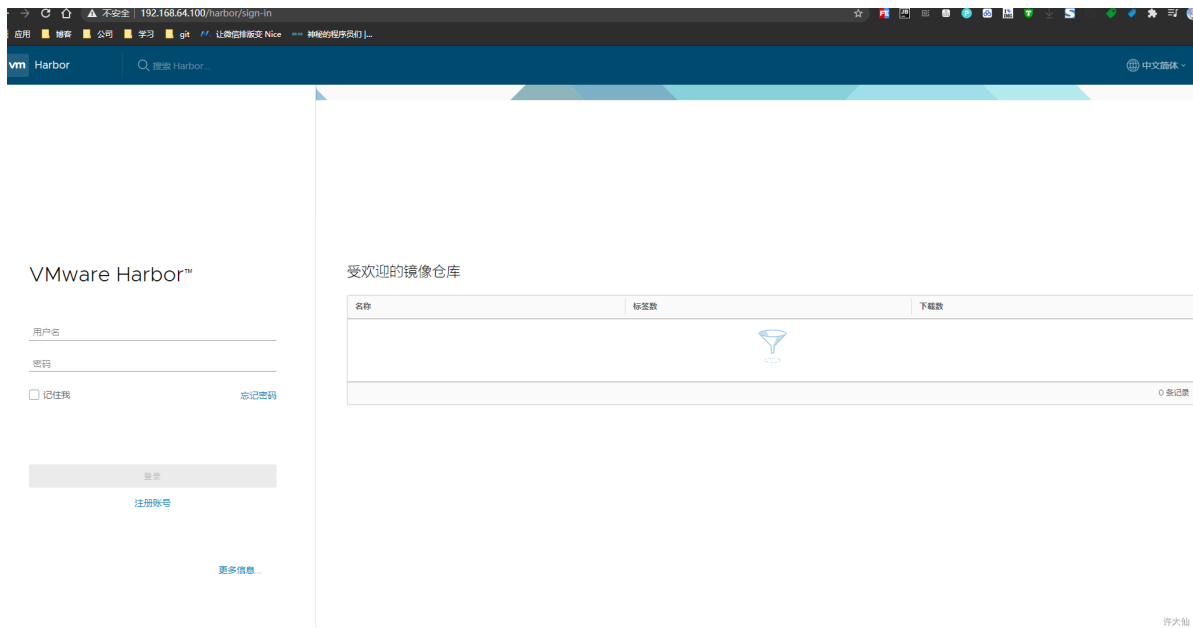
- 在/usr/local/harbor目录下执行如下的命令：

```
./install.sh
```

```
Creating network "harbor_harbor" with the default driver  
Creating harbor-log ... done  
Creating harbor-adminserver ... done  
Creating harbor-db ... done  
Creating registry ... done  
Creating harbor-ui ... done  
Creating nginx ... done  
Creating harbor-jobservice ... done  
  
✓ ----Harbor has been installed and started successfully.----  
  
Now you should be able to visit the admin portal at https://hub.sunxiaping.com.  
For more details, please visit https://github.com/vmware/harbor .
```

2.3 Harbor的访问测试

- 在浏览器中输入<https://192.168.64.100/harbor/sign-in>，并输入admin/Harbor12345进行登录。



2.4 Harbor指定镜像仓库的地址

```
vim /etc/docker/daemon.json
{
  "registry-mirrors": ["https://du3ia00u.mirror.aliyuncs.com"],
  "insecure-registries": ["192.168.64.100", "hub.sunxiaping.com"]
}
systemctl daemon-reload
systemctl restart docker
```

```
vim /etc/hosts

192.168.64.100 hub.sunxiaping.com
```

2.5 Harbor下载测试镜像

```
docker pull hello-world
```


2.6 Harbor给镜像重新打上标签

```
docker tag hello-world:latest hub.sunxiaping.com/sy/hello-world:v1.0
```

2.7 Harbor登录

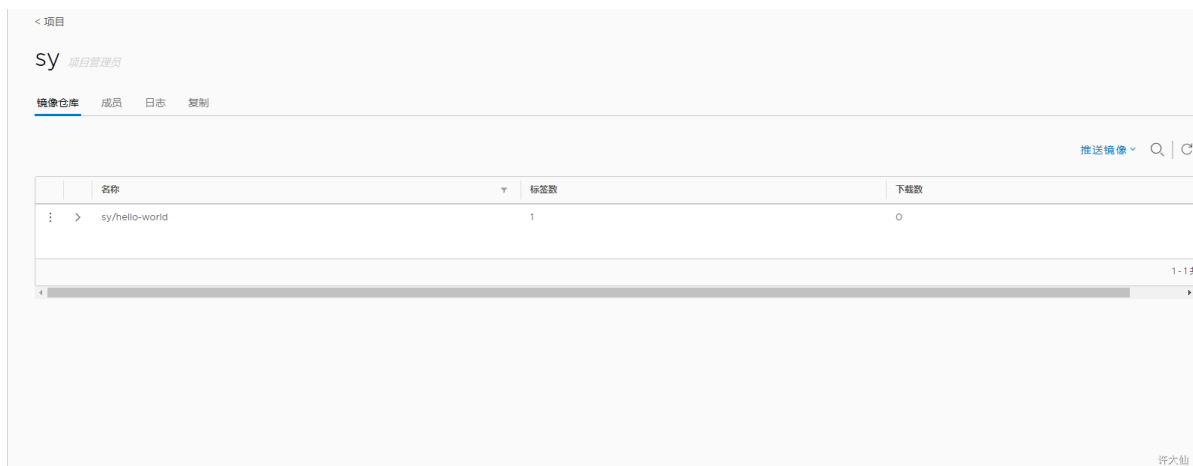
```
docker login 192.168.64.100
```

或

```
docker login hub.sunxiaping.com
```

2.8 Harbor推送镜像

```
docker push hub.sunxiaping.com/sy/hello-world:v1.0
```



2.9 其他Docker客户端下载镜像

2.9.1 指定镜像仓库地址

```
vim /etc/docker/daemon.json
{
  "registry-mirrors": ["https://du3ia00u.mirror.aliyuncs.com"],
  "insecure-registries": ["192.168.64.100", "hub.sunxiaping.com"]
}
systemctl daemon-reload
systemctl restart docker
```

```
vim /etc/hosts

192.168.64.100 hub.sunxiaping.com
```

2.9.2 登录

```
docker login 192.168.64.100
或
docker login hub.sunxiaping.com
```

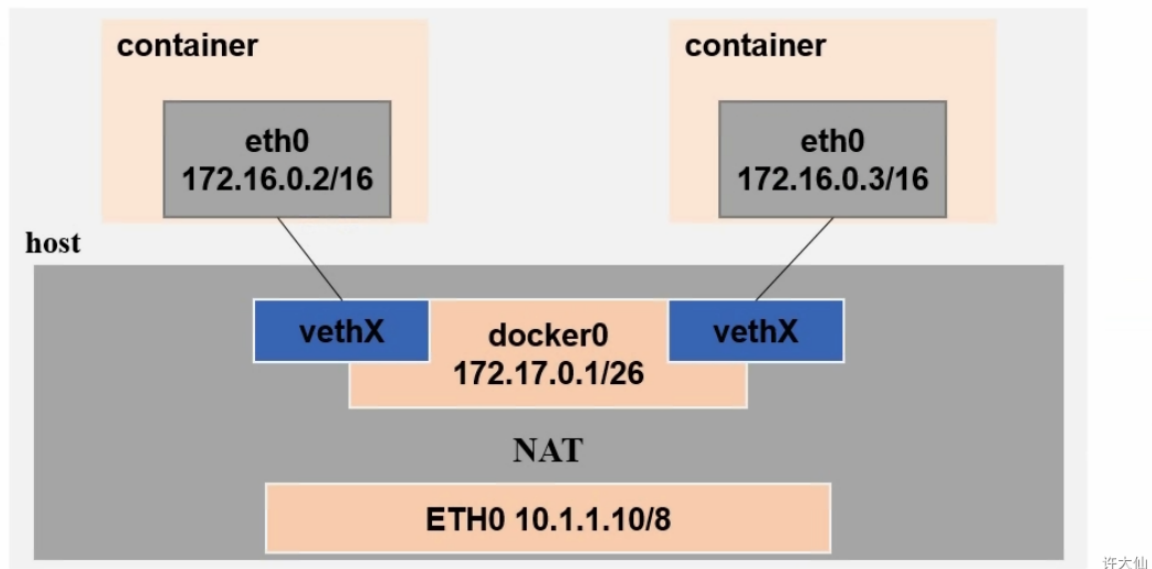
2.9.3 下载镜像

```
docker pull hub.sunxiaping.com/sy/hello-world:v1.0
```

Docker网络管理

1 Docker网络通讯

- 在通常情况下，Docker使用网桥（Bridge）和NAT的通信模式。



许大仙

- 简而言之，Docker需要解决的就是容器和容器之间的通讯、容器访问外部网络、外部网络访问容器的问题。
- 容器和容器之间的通讯：
 - Docker是通过docker0网卡来通信的，docker0网卡有点类似于交换器。

```
[root@localhost ~]# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    inet6 fe80::42:b7ff:fe89:3114 prefixlen 64 scopeid 0x20<link>
    ether 02:42:b7:89:31:14 txqueuelen 0 (Ethernet)
    RX packets 1420 bytes 58662 (57.2 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1643 bytes 8749207 (8.3 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.179.100 netmask 255.255.255.0 broadcast 192.168.179.255
    inet6 fe80::1096:22b3:aaa9:f9b prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:c4:9b:b2 txqueuelen 1000 (Ethernet)
    RX packets 801368 bytes 868216667 (827.9 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 305818 bytes 38276790 (36.5 MiB)
    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
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 21796 bytes 1389602 (1.3 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 21796 bytes 1389602 (1.3 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth1a8ba61: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::2460:fcff:fe13:b2e prefixlen 64 scopeid 0x20<link>
    ether 26:60:fc:13:0b:2e txqueuelen 0 (Ethernet)
    RX packets 1420 bytes 78542 (76.7 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1649 bytes 8749683 (8.3 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.122.1 netmask 255.255.255.0 broadcast 192.168.122.255
    ether 52:54:00:e0:4f:0d txqueuelen 1000 (Ethernet)
    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
```

许大仙

- 每启动一个Docker，会在容器内部生成一个虚拟网卡。

```
[root@localhost ~]# docker exec -it 8a45841f9ed7 /bin/bash
root@8a45841f9ed7:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.2 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:ac:11:00:02 txqueuelen 0 (Ethernet)
    RX packets 1649 bytes 8749683 (8.3 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1420 bytes 78542 (76.7 KiB)
    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@8a45841f9ed7:/#
```

虚拟网卡

许大仙

- 在主机，会产生一个与之对应的vethx，即namespace（命名空间，命名空间之间是隔断的，每产生一个Docker容器，会在主机产生一个与之对应的vethx）。

```
[root@localhost ~]# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    inet6 fe80::42:b7ff:fe89:3114 prefixlen 64 scopeid 0x20<link>
    ether 02:42:b7:89:31:14 txqueuelen 0 (Ethernet)
    RX packets 1420 bytes 58662 (57.2 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1643 bytes 8749207 (8.3 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.179.100 netmask 255.255.255.0 broadcast 192.168.179.255
    inet6 fe80::1096:22b3:eea9:f9b prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:c4:9b:b2 txqueuelen 1000 (Ethernet)
    RX packets 801368 bytes 868216667 (827.9 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 305818 bytes 38276790 (36.5 MiB)
    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
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 21796 bytes 1389602 (1.3 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 21796 bytes 1389602 (1.3 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth1a8ba0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::2460:fcff:fe13:b2e prefixlen 64 scopeid 0x20<link>
    ether 26:60:fc:13:0b:2e txqueuelen 0 (Ethernet)
    RX packets 1420 bytes 78542 (76.7 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1649 bytes 8749683 (8.3 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.122.1 netmask 255.255.255.0 broadcast 192.168.122.255
    ether 52:54:00:e0:4f:0d txqueuelen 1000 (Ethernet)
    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@localhost ~]#
```

在主机，会产生一个与之对应的vethx

许大仙

- 容器访问外部网络：就是借助了SNAT（源地址转换）。SNAT：源地址转换是内网地址向外访问时，发起访问的内网ip地址转换为指定的ip地址（可指定具体的服务以及相应的端口或端口范围），这可以使内网中使用保留ip地址的**主机**访问外部网络，即内网的多部主机可以通过一个有效的公网ip地址访问外部网络。A公司拥有多个**公网IP**（60.191.82.105-107），A公司希望内部用户（IP为192.168.1.50）使用某个特定的IP（60.191.82.107）访问互联网，则需在出口路由设备上需要配置源地址转换。

```
iptables -t nat -A POSTROUTING -s 172.17.0.0/16 -o docker0 -j MASQUERADE
```

- 外部网络访问容器：就是借助了DNAT（向internet发布内网服务器）。DNAT：内网web服务器，或是ftp服务器，为了用户在公网也可以访问，有不想买公网ip地址，采用DNAT方案。

```
docker run -d -p 80:80 apache
```

2 Docker网络模式修改

2.1 Docker进程网络修改（不常用）

- `-b, --bridge=""`: 指定Docker使用的网桥设备，默认情况下Docker会自动创建和使用docker0网桥设备，通过此参数可以使用已经存在的网桥设备。
- `--bip`: 指定docker0的IP和掩码，使用标准的CIDR形式，如：10.10.10.10/24。
- `--dns`: 配置容器的DNS，在启动Docker进程是添加，所有容器全部生效。

2.2 Docker容器网络修改

- `--dns`: 用于指定启动的容器的DNS。
- `--net`: 用于指定容器的网络通讯方式，有如下四种值：
 - `bridge`: Docker的默认方式，网桥模式。
 - `none`: 容器没有网络栈。
 - `container`: 使用其他容器的网络栈，Docker容器会加入其他容器的network namespace。
 - `host`: 表示容器使用Host的网络，没有自己独立的网络栈。容器可以完全访问HOST的网络，不安全。

2.3 -p/P选项的使用格式

- `-p 容器端口`: 将指定的容器端口映射到主机所有地址的一个动态端口。
- `-p 主机端口:容器端口`: 映射到指定的主机端口。
- `-p IP地址:容器端口`: 映射到指定的主机的IP的动态端口。
- `-p IP地址::主机端口:容器端口`: 映射到指定的主机IP的主机端口。
- `-P`: 暴露所需要的所有端口。

`docker port 容器id`: 可以查看当前容器的映射关系。

3 网络隔离

- 查看当前可用的网络类型:

```
docker network ls
```

```
[root@localhost ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
eee3e51839cd        bridge             bridge              local
3bb0e95da6fc        host               host                local
a0ee9d0b3dc5        none               null                local
[root@localhost ~]#
```

- 创建网络空间:

```
docker network create -d 类型 网络空间名称
```

类型分为overlay和bridge

```
[root@localhost ~]# docker network create -d bridge lamp
84f0a48dfed73821ed473ec06f5b41e8dbe46f5b5425184b0f395e431c02a61f
[root@localhost ~]# docker network create -d bridge lnmp
8a9ec81850cc149ae96f736b73150bcf8736121ca44d1d801ee0ebb466a50763
[root@localhost ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
eee3e51839cd        bridge             bridge              local
3bb0e95da6fc        host               host                local
84f0a48dfed7        lamp               bridge              local
8a9ec81850cc        lnmp               bridge              local
a0ee9d0b3dc5        none               null                local
[root@localhost ~]#
```

```

[root@localhost ~]# ifconfig
br-84f0a48dfed7: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.18.0.1 netmask 255.255.0.0 broadcast 172.18.255.255
    ether 02:42:0f:3a:16:87 txqueuelen 0 (Ethernet)
    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

br-8a9ec81850ec: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.19.0.1 netmask 255.255.0.0 broadcast 172.19.255.255
    ether 02:42:aa:59:ff:94 txqueuelen 0 (Ethernet)
    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

docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:83:50:a8:fd txqueuelen 0 (Ethernet)
    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

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.179.100 netmask 255.255.255.0 broadcast 192.168.179.255
    inet6 fe80::1096:22b3:eea9:f9b prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:c4:9b:b2 txqueuelen 1000 (Ethernet)
    RX packets 734 bytes 65904 (64.3 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 438 bytes 63188 (61.7 KiB)
    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
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    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

virbr0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.122.1 netmask 255.255.255.0 broadcast 192.168.122.255
    ether 52:54:00:e0:4f:0d txqueuelen 1000 (Ethernet)
    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@localhost ~]#

```

新生成的网络空间
对应的网卡

许大仙

- 启动2个MySQL服务器，使用刚才新建的网络名称：

```

docker run -id -p 3306:3306 --name lampmysql5.7 --network lamp -v
/var/lampmysql5.7/conf:/etc/mysql/conf.d -v /var/lampmysql5.7/logs:/logs -v
/var/lampmysql5.7/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 mysql:5.7 --
lower_case_table_names=1

```

```

docker run -id -p 3307:3306 --name lnmpmysql5.7 --network lnmp -v
/var/lnmpmysql5.7/conf:/etc/mysql/conf.d -v /var/lnmpmysql5.7/logs:/logs -v
/var/lnmpmysql5.7/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 mysql:5.7 --
lower_case_table_names=1

```

```
[root@localhost ~]# docker run -id -p 3306:3306 --name lampmysql5.7 --network lamp -v /var/lampmysql5.7/conf:/etc/mysql/conf.d -v /var/lampmysql5.7/logs:/logs -v /var/lampmysql5.7/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 mysql:5.7 --lower_case_table_names=1
189cc80e64ab6c6a36e37a40470133eeb2b150c46f93204e7cb0a35433f4878
[root@localhost ~]# docker run -id -p 3307:3306 --name lnmppmysql5.7 --network lnmpp -v /var/lnmppmysql5.7/conf:/etc/mysql/conf.d -v /var/lnmppmysql5.7/logs:/logs -v /var/lnmppmysql5.7/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 mysql:5.7 --lower_case_table_names=1
0f773becd59585348a0eb3c8bd5d7482b0fedfba2a56ec7df9c17ed6a1b12a9
[root@localhost ~]#
```

```
[root@localhost ~]# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
0f773becd595  mysql:5.7     "docker-entrypoint.s..." 4 minutes ago  Up 4 minutes  33060/tcp, 0.0.0.0:3307->3306/tcp  lnmppmysql5.7
189cc80e64ab  mysql:5.7     "docker-entrypoint.s..." 4 minutes ago  Up 4 minutes  0.0.0.0:3306->3306/tcp, 33060/tcp  lampmysql5.7

[root@localhost ~]# docker exec -it 0f773becd595 ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.19.0.2 netmask 255.255.0.0 broadcast 172.19.255.255
    ether 02:42:c3:c1:00:02 txqueuelen 0 (Ethernet)
    RX packets 4700 bytes 894505 (0.5 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 2805 bytes 153344 (149.7 KiB)
    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 8 bytes 745 (745.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 745 (745.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@localhost ~]# ifconfig
br-84f0a48dfed7: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.19.0.1 netmask 255.255.0.0 broadcast 172.19.255.255
    inet6 fe80::42:fff:fe3a:1687 prefixlen 64 scopeid 0x20<link>
    ether 02:42:c3:c1:00:02 txqueuelen 0 (Ethernet)
    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

br-8a9ec8185f0c: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.19.0.2 netmask 255.255.0.0 broadcast 172.19.255.255
    inet6 fe80::42:aaff:fe59:ff94 prefixlen 64 scopeid 0x20<link>
    ether 02:42:aa:59:ff:94 txqueuelen 0 (Ethernet)
    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

docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:83:59:a8:fd txqueuelen 0 (Ethernet)
    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

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.179.100 netmask 255.255.255.0 broadcast 192.168.179.255
    inet6 fe80::1066:22b3:aaf:f9b prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:c4:9b:b2 txqueuelen 1000 (Ethernet)
    RX packets 8242 bytes 9185612 (8.7 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 4097 bytes 383790 (374.7 KiB)
    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
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
```

这个MySQL服务器在172.19.0.1网络上，不在docker0那个网段内

许次仙