

Docker概述

Docker为什么出现

传统:开发jar,运维来部署!

现在:开发打包部署上线,一套流程做完!



隔离:Docker核心思想!打包装箱!每个箱子是相互隔离的。

Docker通过隔离机制,可以将服务器利用到极致!

本质: 所欲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!

Docker的历史

2010年, 几个搞IT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有关的容器技术!

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!

开源

2013年Docker开源了!

Docker越来越多的人发现了Docker 的优点!火了,Docker每个月都会更新哪一个版本!

2014年4月9日,Docker1.0发布!

Docker为什么这么火?相对虚拟机十分轻巧!

在容器技术出来之前,我们都是使用的虚拟机技术!

虚拟机 在Windows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! 缺点是笨重!

虚拟机也是属于虚拟化技术, Docker容器技术, 也是一中虚拟化技术!

1 **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

2

3 **docker**, 隔离, 镜像(最核心的环境4m+jdk+mysql)十分小巧, 运行镜像就可以了! 小巧! 几个M KB 秒级启动

到现在所有开发人员都必须要会Docker!

Docker是基于Go语言开发的! 开源项目

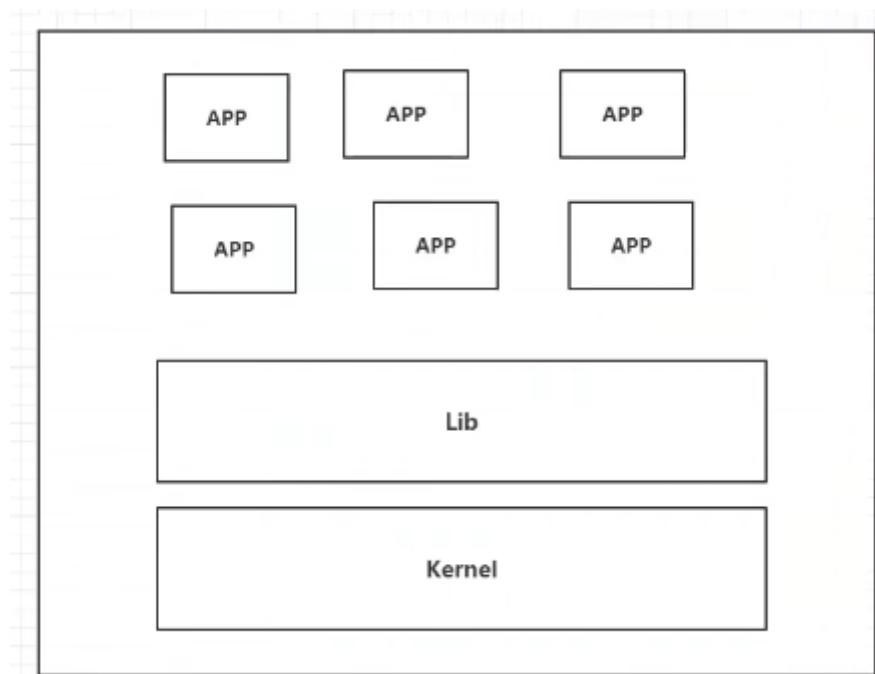
官网: <https://www.docker.com/>

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

仓库地址:<https://hub.docker.com/>

Docker能干嘛

之前的虚拟机技术!

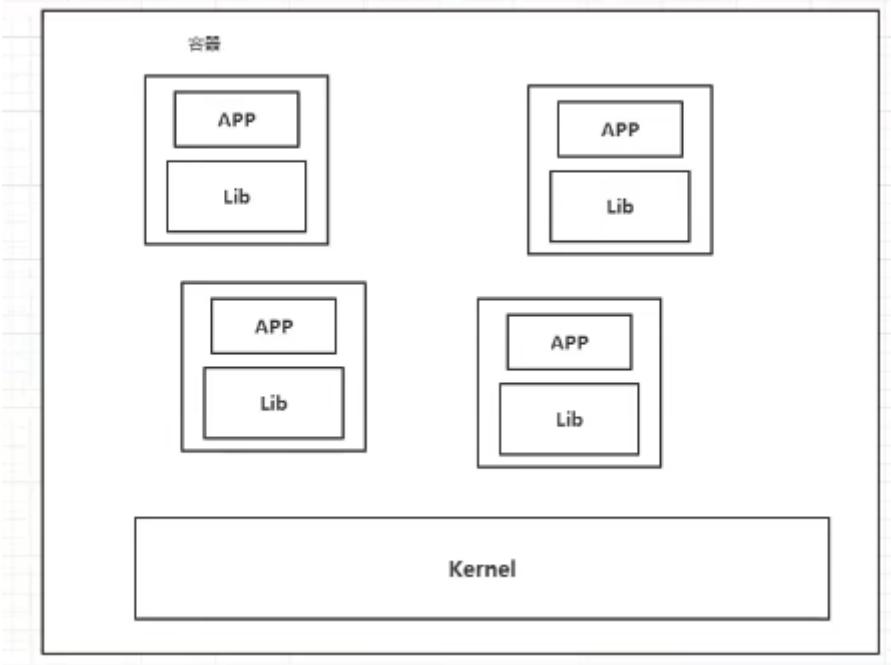


虚拟机的缺点:

- 1、资源占用十分多
- 2、冗余步骤多
- 3、启动很慢

容器化技术

虚拟化技术不是模拟的一个完整的操作系统



比较Docker和虚拟机的技术不同

- 传统虚拟机，虚拟出一条硬件，运行一个完整的系统，然后在这个系统上安装运行软件
- 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，所有就轻便了
- 每个容器是相互隔离的，每个容器都有属于自己的文件系统，互不影响

DevOps(开发、运维)

应用更快速的交付和部署

传统：一堆帮助文档，安装程序

Docker：打包镜像发布测试，一件运行

更便捷的升级和扩缩容

使用Docker之后，我们部署应用就和搭积木一样！

项目打宝为一个镜像，扩展 服务器A！ 服务器B

更简单的系统运维

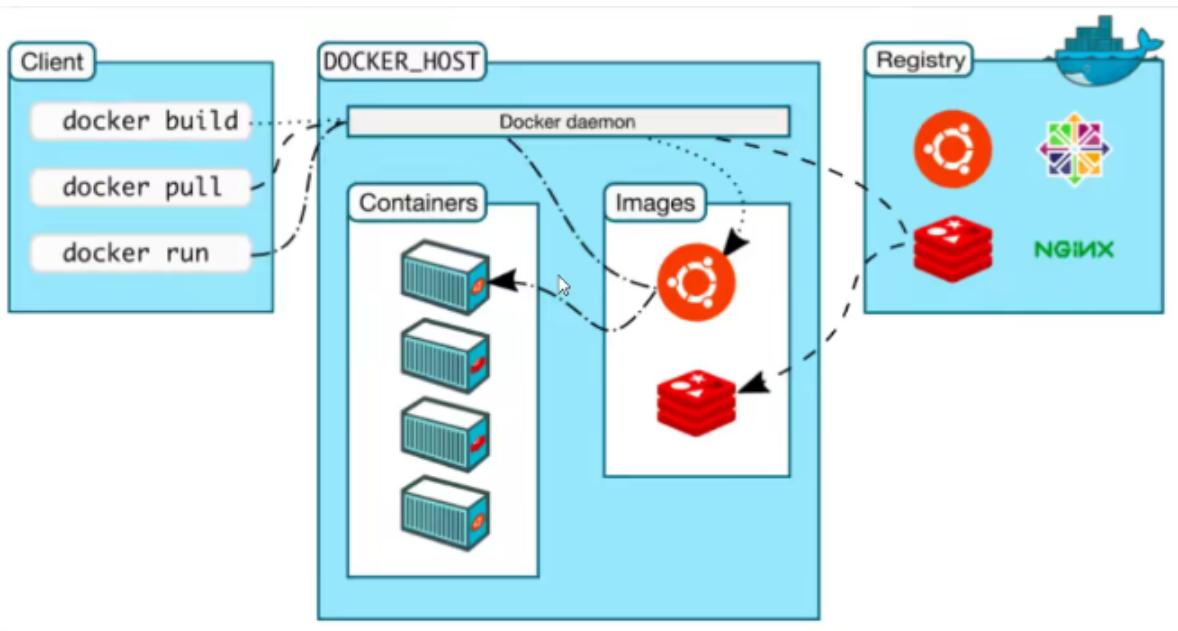
在容器化之后，我们的开发，测试环境都是高度一致。

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。

Docker安装

Docker的基本组成



镜像 (images) :

Docker镜像就好比是一个模板，看他通过这个模板来在创建容器服务，tomcat镜像==>run ==> tomcat01容器（提供服务），通过这个镜像可以创建多个容器（最终服务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，通关镜像来创建的。

启动，停告知删除，基本命令

目前就可以把这个容器理解为就是一个建议的liunx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！

Docker Hub (默认是国外的)

阿里云...都有容器服务（配置镜像加速）

安装Docker

环境准备

- 1、需要会一点点的liunx的基础
- 2、CentOS 7
- 3、使用远程连接软件Xsheel

环境查看

```

1 #查看系统内核
2 uname -r
3 #查看系统版本
4 cat /etc/os-release

```

安装

帮助文档：文档地址：<https://docs.docker.com/>

```
1 # 1、卸载旧的版本
2 yum remove docker \
3             docker-client \
4             docker-client-latest \
5             docker-common \
6             docker-latest \
7             docker-latest-logrotate \
8             docker-logrotate \
9             docker-engine
10
11 # 2、需要的安装包
12 yum install -y yum-utils
13
14
15 # 报错了： 错误： rpmbuild: BDB0113 Thread/process 136661/140660029847360 failed:
16 BDB1507 Thread died in Berkeley DB library
17 # 解决地址: https://www.cnblogs.com/lqlinux/p/12963725.html
18 # 解决方式：重新构建rpm数据库
19 [root@localhost ~]# cd /var/lib/rpm
20 [root@localhost rpm]# ls
21 Basenames      __db.001  __db.003  Group          Name           Packages
22 Requirename    Sigmd5
23 Conflictname   __db.002  Dirnames  Installtid  Obsoletename  Providename
24 Sha1header     Triggername
25 [root@localhost rpm]# rm -rf __db*
26 [root@localhost rpm]# rpm --rebuilddb
27
28 # 3、设置镜像的仓库
29 yum-config-manager \
30   --add-repo \
31   https://download.docker.com/linux/centos/docker-ce.repo #官网是国外的
32
33 yum-config-manager \
34   --add-repo \
35   http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo #推荐使用阿里
36 云的
37
38 # 更新yum软件包索引
39 yum makecache fast
40
41 # 4、安装docker相关的内容 docker-ce 社区版 ee企业版
42 yum install docker-ce docker-ce-cli containerd.io
43
44 # 5、启动docker
45 systemctl start docker
46
47 # 6、判断docker是否安装成功
48 docker version
49
50 # 7、hello-world
51 docker run hello-world
52
53 # 8、查看一下下载的这个hello-world镜像
```

了解：卸载docker

```

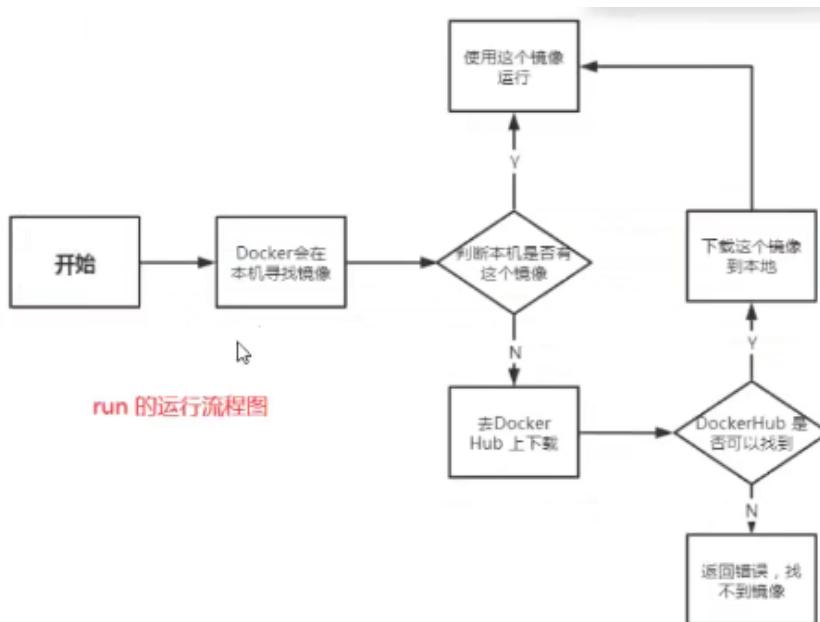
1 # 1、卸载依赖
2 yum remove docker-ce docker-ce-cli containerd.io
3
4 # 2、删除资源
5 rm -rf /var/lib/docker
6
7 # /var/lib/docker    docker的默认工作路径

```

阿里云镜像加速

- 1、登录阿里云找到容器服务
- 2、找到镜像加速地址
- 3、配置使用

回顾HelloWord流程

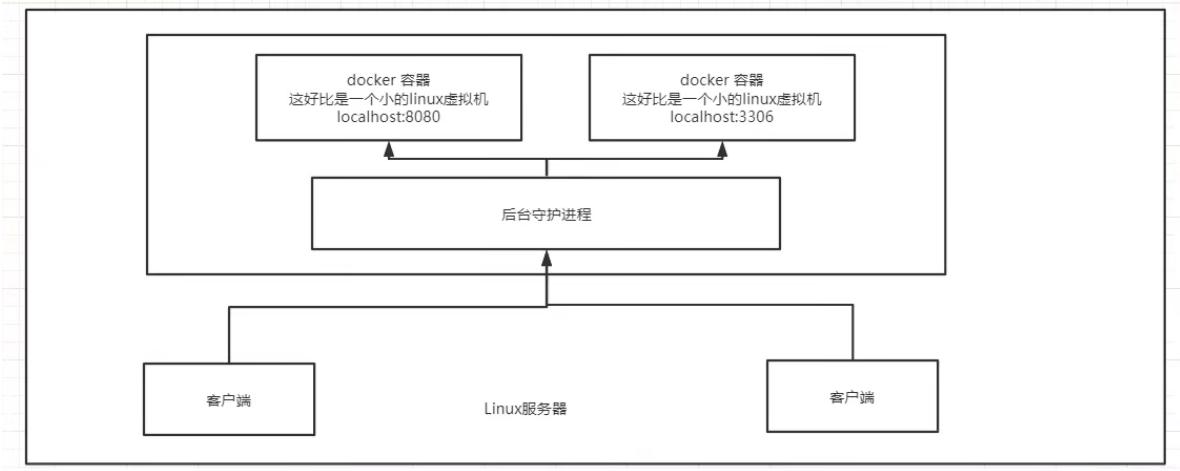


底层原理

Docker是怎么工作的？

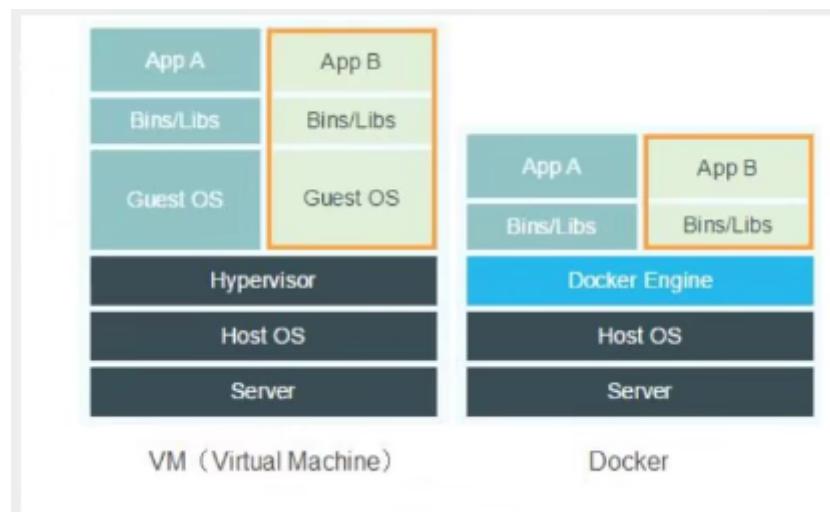
Docker是一个Client-Server结构的系统，Docker的守护进程运行在主机上。通过Socket从客户端访问！

DockerServer接收到Docker-Client的指令，就会执行这个命令！



Docker为什么比VM快?

- 1、Docker有着比虚拟机更少的抽象层。
- 2、Docker利用的是宿主机的内核，vm需要是GuestOS。



所以说，新建一个容器的时候，Docker不需要像虚拟机一样出现加载一个操作系统，避免引导。虚拟机加载GuestOS，分钟级别的，而Docker是利用宿主机的操作系统，省略了这个复杂的过程，秒级的！

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

之后学习完毕所有的命令，在回过头来看这段理论，就会很清晰！

Docker的常用命令

帮助命令

```
1 docker version      #显示docker版本信息  
2 docker info        #显示docker的系统信息，包括镜像和容器的数量  
3 docker 命令 --help    #帮助命令
```

官方文档：<https://docs.docker.com/reference/>

镜像命令

```
1 docker images    # 查看所有本地主机上的镜像  
2  
3 # 解释  
4 REPOSITORY 镜像的仓库源  
5 TAG        镜像的标签  
6 IMAGE ID   镜像的id  
7 CREATED    镜像的创建时间  
8 SIZE       镜像的大小  
9  
10 # 可选项  
11 -a #列出所有镜像  
12 -q #值显示镜像的ID  
13  
14
```

docker search #搜索镜像

```
1 #可选项，通过收藏来过滤  
2 --filter=STARS=3000 #搜索出来的镜像就是STARS大于3000  
3  
4 docker search mysql --filter=STARS=3000 #搜索mysql
```

docker pull 下载镜像

```
1 # 下载镜像docker pull 镜像名 [:tag]  
2 docker pull mysql    #如果不写tag，默认就是latest  
3 #docker pull的原理：分层下载，docker image的核心 联合文件系统  
4 #Digest 签名  
5 #docker.io 真实地址  
6  
7 # 等价于它  
8 docker pull mysql  
9 docker pull docker.io/library/mysql:latest  
10  
11 #指定版本  
12 docker pull mysql:5.7  
13
```

docker rmi删除镜像

```
1 docker rmi -f 镜像ID #根据ID删除指定镜像  
2 docker rmi -f 镜像ID 镜像ID 镜像ID    #删除多个镜像  
3 docker rmi -f $(docker images -aq) #删除所有镜像
```

容器命令

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

```
1 | docker pull centos
```

新建容器并启动

```
1 | docker run [可选参数] image
2 |
3 | # 参数说明
4 | --name="Name"    # 容器名字 tomcat01 tomcat02 , 用来区分容器
5 | -d   # 后台方式运行
6 | -it  # 使用交互方式运行, 进入容器查看内容
7 | -p   # 指定容器的端口 -p 8080:8080
8 |     -p ip: 主机端口容器端口
9 |     -p 主机端口: 容器端口 (常用)
10 |     -p 容器端口
11 | -P   # 随机指定端口
12 |
13 | #测试,启动并进入容器
14 | dockeer run -it centos /bin/bash
```

列出所有的运行的容器

```
1 | # docker ps 命令
2 |      # 列出所有的运行的容器
3 | -a   # 列出所有的运行的容器+带出历史运行过的容器
4 | -n=? # 系那是最创建的容器
5 | -q   # 只显示容器的编号
```

退出容器

```
1 | exit    # 直接容器停止并退出到主机
2 | Ctrl+p+q # 容器不停止退出
```

删除容器

```
1 | 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 -f
2 | docker rm -f $(docker ps -aq) # 删除所有的容器
3 | docker ps -a -q|xargs docker rm # 删除所有的容器
```

启动和停止容器的操作

```
1 | docker start 容器ID # 启动容器
2 | docker restart 容器ID # 重启容器
3 | docker stop 容器ID #停止当前正在运行的容器
4 | docker kill 容器ID #强制停止当前容器
```

常用的其他命令

后台启动容器

```
1 | # 命令 docker run -d 镜像名:
2 | docker run -d centos
3 | # 问题docker ps 发现centos停止了
4 |
5 | # 常见的坑, docker容器使用后台运行, 就必须要有一个前台进程, docker发现没有应用, 就会自动停止
6 | # nainx, 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

```
1 docker logs -tf --tail 10 容器ID # 容器, 没有日志
2
3 # 自己编写一个脚本
4 docker run -d centos /bin/sh -c "while true;do echo kuangshen;sleep 1;done"
5
6 docker ps # 查看运行的容器
7
8 #显示日志
9 -rf # 显示日志 f是带上时间戳
10 --tail number # 要显示日志条数
11 docker logs -tf --tail 10 容器ID
```

查看容器中进程信息ps

```
1 docker top 容器ID
```

查看镜像的元数据

```
1 # docker inspect 容器ID #查看镜像的元数据
```

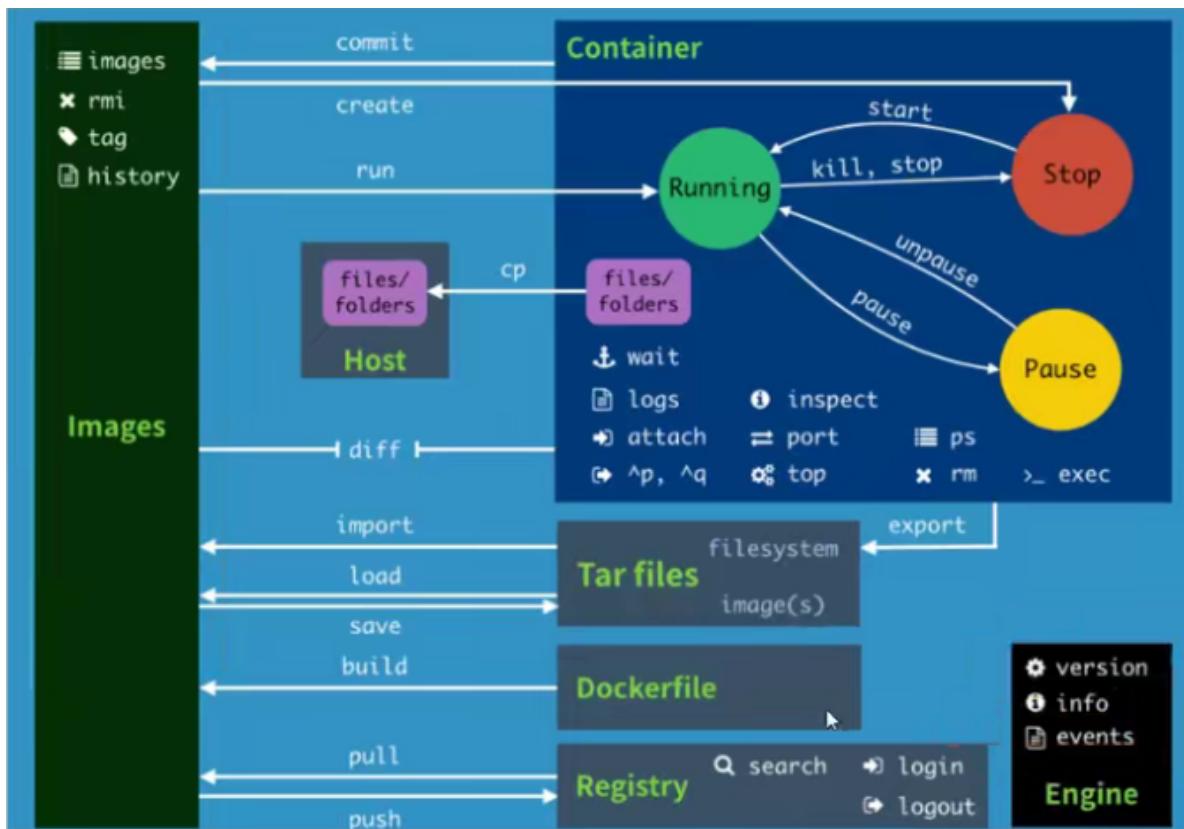
进入当前正在运行的容器

```
1 # 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
2
3 # 命令
4 docker exec -it 容器id bashshell
5
6 # 测试
7 docker exec -it 容器id /bin/bash
8
9 # 方式二: 正在执行当前的代码...
10 docker attach 容器id
11
12 # docker exec # 进入容器后开启一个新的终端, 可以在里面操作 (常用)
13 # docker attach #进入容器正在执行的终端, 不会启动新的进程
```

从容器内拷贝文件到主机上

```
1 docker cp 容器id:复制路径文件 目标路径
2
3 # 将这个文件拷贝出来到主机上
4 docker cp 容器id:/home/kuangshen.java /home
5
6 #拷贝是一种手动过程, 未来我们使用 -v 卷的技术, 可以实现, 自动同步
```

小节



作业练习

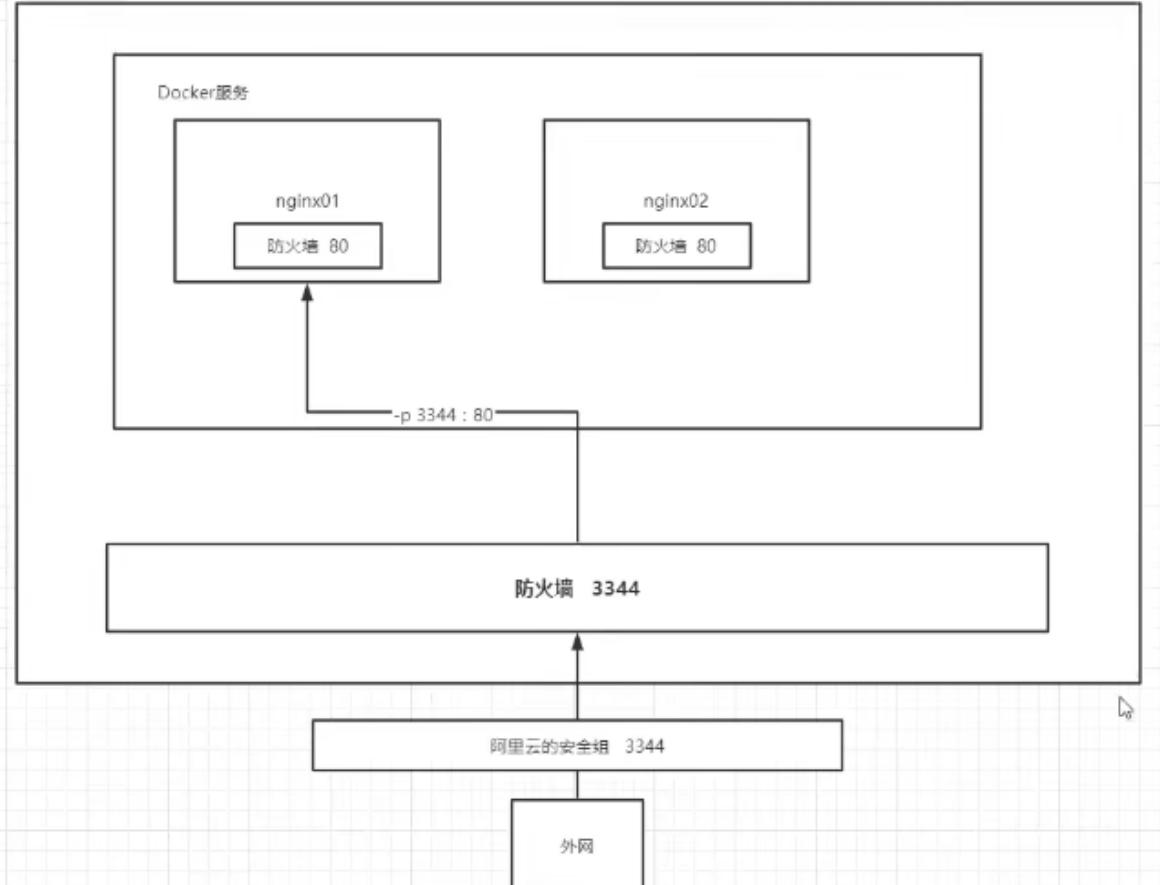
Docker安装Nainx

```

1 # 1、搜索镜像 search 建议大家去docker搜索，可以看到帮助文档
2 # 2、下载镜像 pull
3 # 3、运行测试
4 #   docker images 查看当前镜像
5
6 #   docker run -d --name nainx01 -p 3344:80 nginx
7 #       -d 后台运行
8 #       --name 给容器名
9 #       -p 宿主机端口，容器内部端口
10
11 #   docker ps    查看运行中的镜像
12
13 #   docker exec -it nainx01 /bin/bash    进入容器
14 #   whereis nainx    查看nainx的目录
15
16 #   docker stop NAMES    #停止服务
17
18
19 curl localhost:3344 #liunx访问网页

```

端口暴露的概念



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，我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？-v 数据卷！

作业：docker 来装一个tomcat

```

1 # 官网的使用
2 docker run -it --rm tomccat:9.0 #一搬用来测试，用完就删除
3
4 # 我们之前的启动都是后台，停止了容器之后，容器还是可以查到
5 docker run -it tomcat:8 #doerhub上搜索版本号
6
7 # 下载在启动
8 docker pull tomcat:9.0
9
10 # 启动运行
11 docker images #查看镜像包
12
13 # 启动容器
14 docker run -d -p 3355:8080 tomcat01 tomcat
15
16 # 进入容器
17 docker exec -it tomcat01 /bin/bash
18
19 # 发现问题：1、Linux命令少了 2、没有webapps，阿里云镜像的原因，默认是最小镜像，所有不必要的都剔除掉
20 # 保证最小可运行环境！
21
22 # 方式一：把webapps.dist 复制到webapps目录下
23 cp -r webapps.dist/* webapss
24
25 # 查看所有文件

```

26 ls -al
27

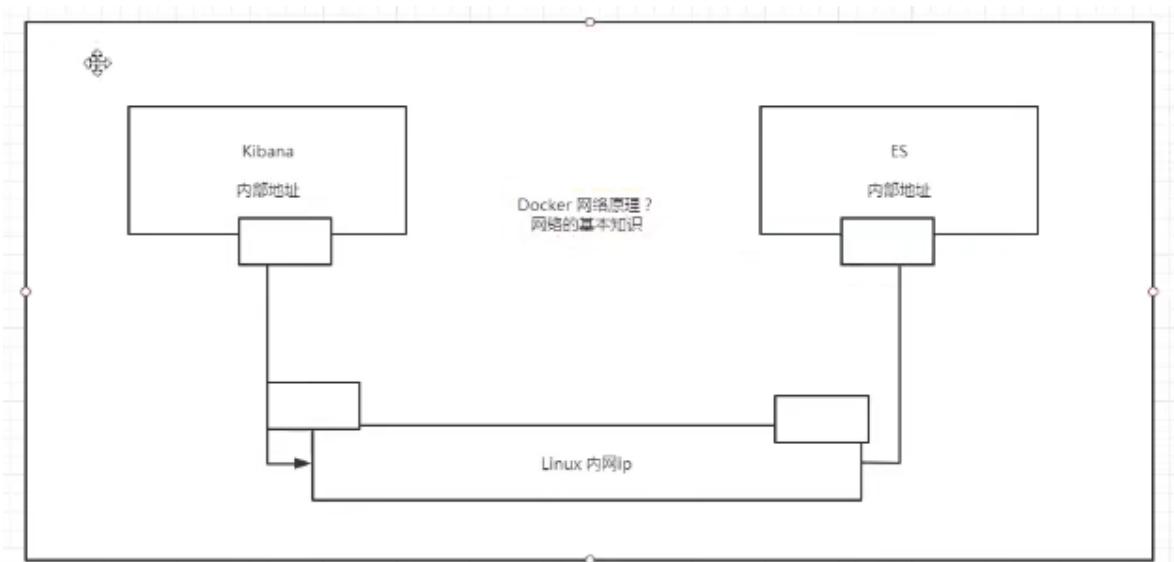
思考问题：我们以后要部署项目，如果每次都要进入容器是不是十分麻烦？我要是可以在容器外部提供一个映射路径，webapps，我们在外部放置项目，就自动同步到内部就好了！

docker容器 tomcat+网站 docker+mysql

作业：部署 es+kibana

```
1 # es 暴露的端口很多!
2 # es 十分耗内存
3 # es 的数据一半需要放置到安全目录！挂载
4
5 # dockerhub 上搜索 elasticsearch
6 #   --net somenetwork ? 网络配置
7 #   tag 版本号
8 docker run -d --name elasticsearch --net somenetwork -p 9200:9200 -p
9   9300:9300 -e "discovery.type-single-node" elasticsearch:tag
10
11 # 去掉--net somenetwork 还没学 启动elasticsearch
12 docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e
13   "discovery.type-single-node" elasticsearch:tag
14
15 # 启动了linux就很卡 直接卡住了
16
17 # 查看 cpu的状态
18 docker stats
19
20 # 增加内训的限制，修改配置文件 -e 环境配置修改
21 docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 -e
22   "discovery.type-single-node" -e ES_JAVA_OPTS="Xms64m -Xmx521m"
23   elasticsearch:tag
```

作业：使用kibana连接ES? 网络如何才能连接过去!



可视化

- portainer (先用这个)

```
1 | docker run -d -p 8080:9000 --restart=always -v  
     /var/run/docker.sock:/var/run/docker.sock --privileged=true  
     portainer/portainer
```

- Rancher (CI/CD 再用)

什么是portainer?

Docker图形化页面管理工具！提供一个后台面板供我们操作！

```
1 | docker run -d -p 8080:9000 --restart=always -v  
     /var/run/docker.sock:/var/run/docker.sock --privileged=true  
     portainer/portainer
```

Docker镜像讲解

镜像是什么

进项是一种轻量级，可执行的独立软件包，用来打包软件运行环境和基于运行环境开发的软件，它包含运行某个软件所需的所有内容，包括代码、运行时、库、环境变量和配置文件。

所有的应用，直接打宝docker镜像，就可以直接跑起来！

如何得到镜像：

- 从远程仓库下载
- 朋友拷贝
- 自己制作一个镜像DockerFile

Docker镜像加载原理

UnionFS(联合文件系统)

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，它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。

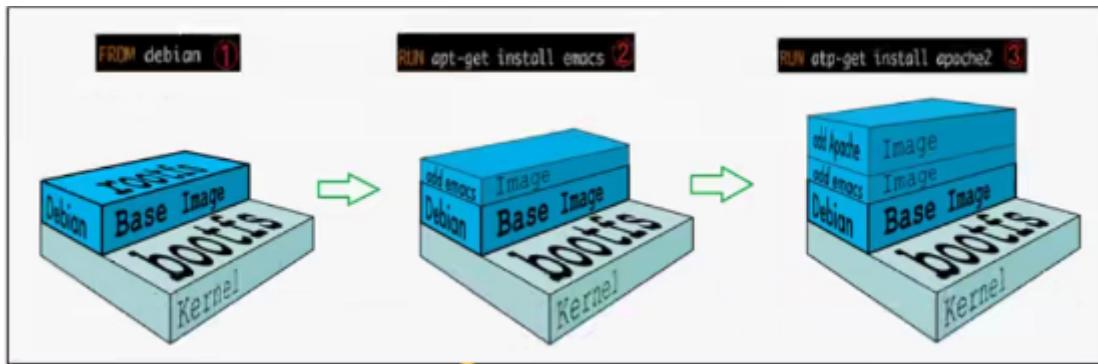
特性：一次同时加载多个文件系统，但从外面看起来，只能看到一个文件系统，联合加载会把各层文件系统叠加起来，这样最终的文件系统会包含所有底层的文件和目录

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.

bootfs(boot file system)主要包含bootloader和kernel, bootloader主要是引导加载kernel, Linux刚启动时会加载bootfs文件系统，在Docker镜像的最底层是bootfs。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。



平时我们安装金虚拟机的CentOs都是好几个G，为什么Docker这里才200M？

对于一个精简的OS，rootfs可以很小，只需要包含最基本的命令，工具和程序就可以了，应为底层直接用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发型版，bootfs基本是一致的，rootfs会有差别，因此不同的发型版可以共用bootfs。

分层理解

分层的镜像

我们可以下载一个镜像，注意观察狭隘的日志输出，可以看到是一层一层的在下载！

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。

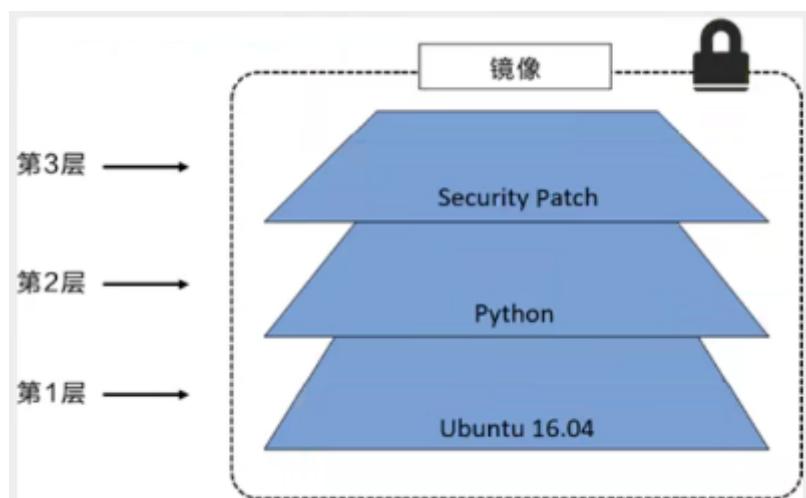
查看镜像分层的方式可以通过docker image inspect命令！

理解

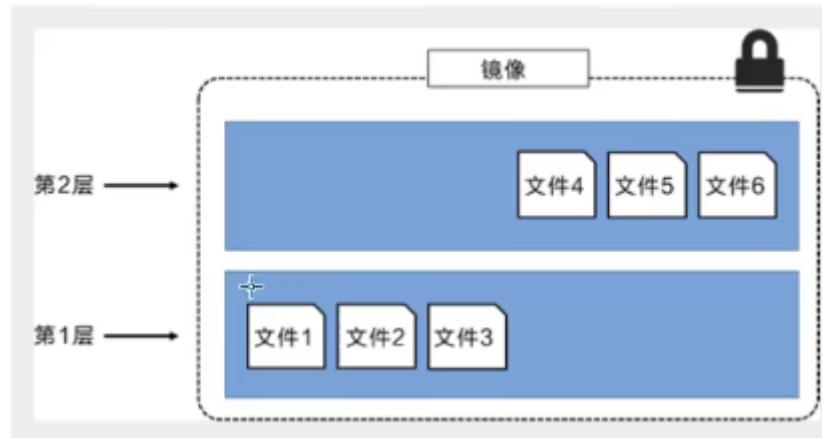
所有的Docker镜像都起始于一个基础镜像层，当进行修改或增加新的内容时，就会在当前镜像层之上，创建新的镜像层。

举一个简单的例子，假如基于Ubuntu Linux 16.04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。

该镜像当前已经包含3个镜像层，如下图所示(这只是一个用于演示的很简单的例子)。

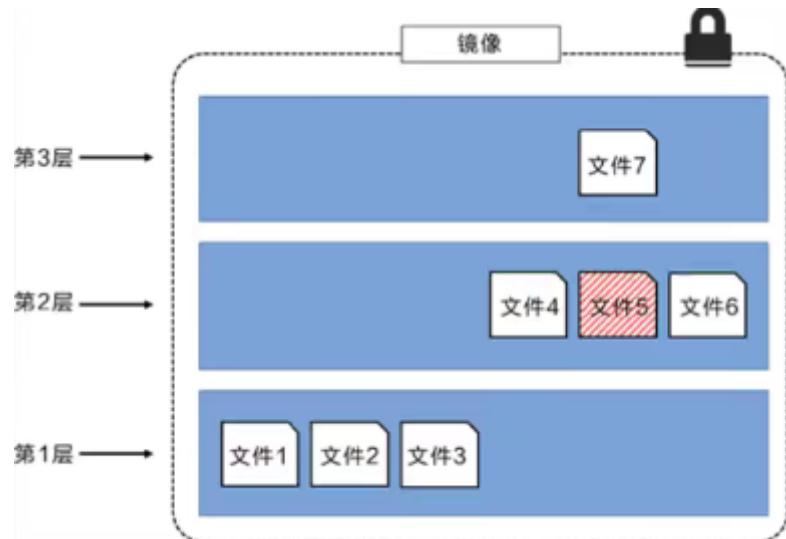


在添加额外的镜像层的同时，镜像始终保持是当前所有镜像的组合，理解这一点非常重要。下图中举了一个简单的例子，每个镜像层包含3个文件，而镜像包含了来自两个镜像层的6个文件。



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的一个更新版本。



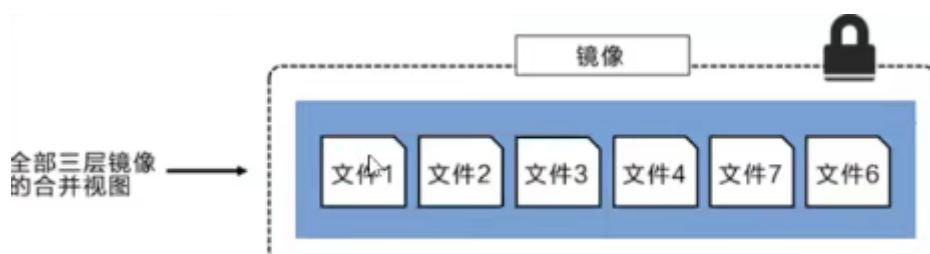
这种情况下，上层镜像层中的文件覆盖了底层镜像层中的文件。这样就使得文件的更新版本作为一个新镜像层添加到镜像当中。

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。所有镜像层堆叠并合并，对外提供统一的视图。



特点

Docker镜像都是只读的，当容器启动时，一个新的科博层被加载到镜像的顶部！

这一层就是我们通常说的容器层，容器之下的都叫镜像层！

如何提交一个自己的镜像

comment镜像

```
1 docker commit #提交容器成为一个新的副本  
2  
3 # 命令和git类似  
4 docker commit -m="提交的描述信息" -a="作者" 容器id 目标镜像名: [TAG]版本号
```

实战测试

```
1 # 启动默认tomcat  
2 docker run -it -p8080:8080 tomcat  
3     #-p 暴露端口  
4     #-it 交互模式启动  
5  
6 # 发现这个默认的tomcat是没有webapps应用，镜像的原因，官方的镜像默认webapps下面是没有文件的  
7  
8 # 查看容器  
9 docker ps  
10  
11 # 进入容器  
12 dockerexec -it 容器id /bin/bash  
13  
14 # 拷贝webapps.dist  
15 cp -r weapps.dist/* weapps  
16     # -r 递归拷贝  
17  
18  
19 # 提交镜像  
20 docker commit -a="Star" -m="add webapps app" 容器id tomcat02:1.0
```

学习方式：理解概念，但是一定要实践，最后实践和理论结合，一次搞定这个知识！

```
1 # 如果你想保存当前容器的状态，就可以通过commit来提交，获得一个镜像  
2 # 就好比学VM中的快照！
```

容器数据卷

什么是容器数据卷

Docker历年回顾

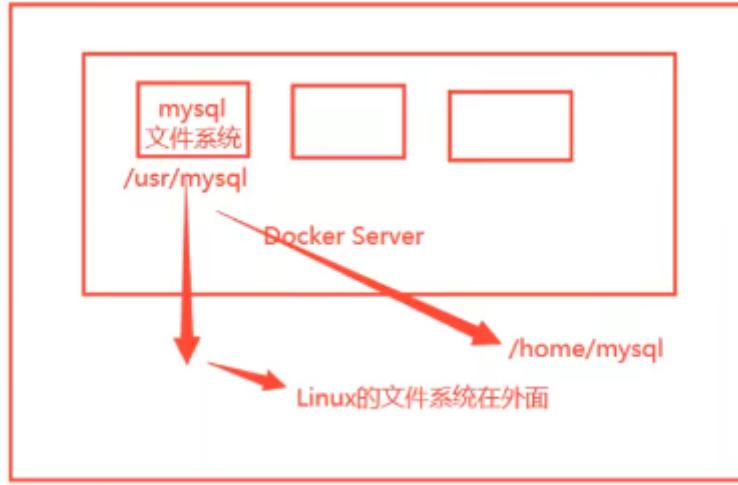
将应用和环境打包成一个镜像！

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，容器删了=删库跑路！需求：MySQL数据可以存储在本地！

容器之间可以有个数据共享技术！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：容器的持久化和同步操作！容器间也是可以数据共享的！

使用数据卷

方式一：直接使用命令来挂载 -v v = volume

```

1 docker run -it -v 主机目录:容器内目录
2
3 # 测试
4 docker run -it -v /home/ceshi:/home centos /bin/bash
5
6 # 启动起来之后，可以通过docker inspect 容器id 查看容器详细信息
7 docker inspect 容器ID
8
9 # 查看所有容器
10 docker ps -a
11
12 # 开启容器
13 docker start 容器ID
14
15 # 连接容器
16 docker attach 容器ID
17

```

```

        "Name": "overdayz",
      },
      "Mounts": [
        {
          "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": "5be0fd0f209b"
      }
    }
  ]
}

```

好处：我们以后修改只需要在本地修改即可，容器会自动同步！

实战：安装MySQL

思考：MySQL的数据持久化问题！

```
1 # 搜索mysql
2 docker search mysql
3
4 # 拉取mysql
5 docker pull mysql:5.7
6
7 # 安装启动MySQL 需要配置密码的，这是注意点！
8 # 运行容器，需要做数据挂载 /home/mysql真实路径 -v可以挂载多个目录
9 # 官方测试：docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=123456 -d
10 mysql:tag
11 # 启动我们的
12 -d 后台运行
13 -p 端口映射
14 -v 数据卷挂载
15 -e 环境配置
16 --name 容器名字
17 docker run -d -p 3310:3306 -v /home/mysql/conf:/etc/mysql/conf.d -v
18 /home/mysql/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=123456 --name mysql01
19 mysql:5.7
20
21 # 启动成功之后，在本地使用sql工具来测试一下
22 # sql工具连接到服务器的3310，3310和容器内的3306映射，这个时候就可以连接上了！
23
24 # 在本地测试创建一个数据库，查看ix我们映射的路径是否ok
25
26 # 删除容器
27 docker rm -f mysql01
```

结论：假设容器删除，发现挂载到本地的数据卷依旧没有丢失，这就是实现了容器数据持久化功能！

具名和匿名挂载

```
1 # 匿名挂载，-v 只指定容器内的，没有指定容器外的
2 -v 容器内路径！
3 docker run -d -p --name nainx01 -v /etc/nginx nginx
4
5 # 查看所有的volume的情况
6 docker volume ls
7 local          3f388efefe8fe8fe8fe8fefafafa8f8ef8
8
9 # 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，没有写容器外的路径！
10
11 # 具名挂载
12 docker run -d -P --name nginx02 -v juming-nginx:/etc/nginx nginx
13
14 # 通过 -v 卷名：容器内路径
15
16 #查看一下这个卷
17 docker volume inspect juming-nginx
```

```
[root@kuangshen home]# docker volume inspect juming-nginx
[{"Created": "2020-05-15T19:37:31+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/volume/xxx/.data`

我们通过具名挂载可以方便的找到我们的一个卷，大多数情况在使用 `具名挂载`

```
1 # 怎么区分匿名挂载和具名挂载，还是指定路径挂载
2 -v 容器内的路径 # 匿名路径
3 -v 卷名:容器内的录# 具名路径
4 -v /宿主机路径:容器内路径 # 指定路径挂载
```

拓展：

```
1 # 通过 -v 容器内路径:ro rw 改变读写权限
2 ro readonly #可读
3 rw readwrite #可读可写
4
5 # 一旦这个设置了容器权限，容器对我们挂载出来的内容就有限定了！
6 docker run -d -P --name nginx2 -v juming-nginx:/etc/nginx:ro nginx
7 docker run -d -P --name nginx2 -v juming-nginx:/etc/nginx:rw nginx
8
9 # ro 只要看到ro就说明这个路径就只能通过宿主机来操作，容器内部是无法操作的。
10 # rw 容器可读可写
```

初识DockerFile

DockerFile就是来构建docker镜像的构建文件！命令脚本！先体验一下！

通过脚本可以生成镜像，镜像是一层一层的，脚本是一个个命令，

方式二：

```
1 # 创建一个dockerfile文件，名字可以随机，建议dockerfile
2 # 文件中的内容 指令（大写）参数
3 # 编写脚本
4 vim dockerfile
5
6 # 脚本内容
7 FROM centos
8 # 匿名挂载
9 VOLUME ["volume01","volume02"]
10 CMD echo "----end----"
11 CMD /bin/bash
12
13 # 这里每个命令，就是镜像的一层
```

```

14
15 # 构建一个镜像
16 -f 构建的地址
17 . 构建到当前的目录下
18 docker build -f /home/docker-test-volume/dockerfile1 -t kuangshen/centos:1.0
.

```

```

[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 e73346bdf465 32 hours ago 448MB
nginx latest 602e11c06b6 3 weeks ago 127MB
centos latest 470671670cac 3 months ago 237MB
[root@kuangshen docker-test-volume]#

```

```

1 # 启动自己写的容器
2 docker run -it 镜像ID

```

```

[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

这个卷和外部一定有一个同步的目录！

```

FROM centos

VOLUME ["volume01", "volume02"] 匿名挂载

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

```

```

1 # 查看匿名卷挂载的路径信息
2 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": ""
      }
    ]
  }
}

```

这种方式在未来使用非常多，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，就要手动挂载 -v 卷名:容器内路径！

数据卷容器

两个MySQL同步数据，数据共享！



```

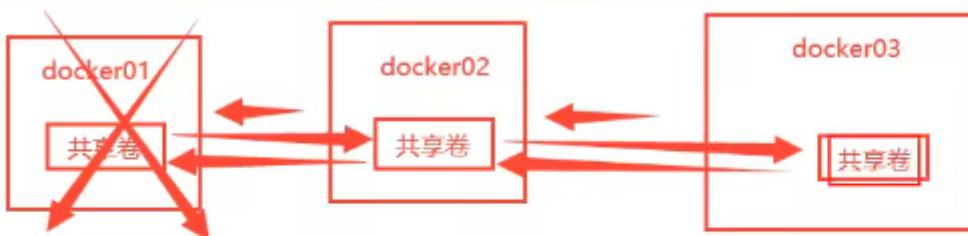
1 # 启动3个容器，通过我们刚才自己写的镜像启动
2 docker run -it --name kuangshen/centos:1.0
3
4 # 不关闭退出
5 ctrl+P+Q
6
7 # 绑定共享数据
8 docker run -it --name docker02 --volumes-from docker01 kuangshen/centos:1.0
9
10 # 进入容器
11 docker attach 容器ID
12
13 # 只要通过 --volumes-from 我们就可以容器间实现数据共享了

```

```
[root@kuangshen /]# docker run -it --name docker01 kuangshen/centos:1.0
[root@45768e0d0196 /]# ls
bin etc lib lost+found mnt proc run srv tmp var volume02
dev home lib64 media opt root sbin sys usr volume01
[root@45768e0d0196 /]# ls -l
total 56
lrwxrwxrwx 1 root root 7 May 11 2019 bin -> usr/bin
drwxr-xr-x 5 root root 360 May 15 12:18 dev
drwxr-xr-x 1 root root 4096 May 15 12:18 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 115 root root 0 May 15 12:18 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
```

数据卷

```
1 # 测试，可以删除docker01，查看一下docker02和docker03是否还可以访问这个文件
2 # 测试发现依旧可以访问
```



拷贝的概念

多个MySQL实现数据共享

```
1 # 启动两个MySQL实现数据共享
2 -d 后台运行
3 -p 端口映射
4 -v 数据卷
5 -e 环境配置
6 # 启动mysql 01
7 docker run -d -p 3310:3306 -v /etc/mysql/conf.d -v /var/lib/mysql -e
  MYSQL_ROOT_PASSWORD=123456 --name mysql01 mysql5.7
8
9 # 启动mysql 02 并绑定mysql 01 卷
10 docker run -d -p 3311:3306 -e MYSQL_ROOT_PASSWORD=123456 --volume-from --
    name mysql01 mysql5.7
11
12 # 这个时候，可以实现两个容器数据同步
```

结论：

容器之间配置信息传递，数据卷容器的生命周期一直持续到没有容器使用为止。

但是一旦持久化到了本地，这个时候本地的数据是不会删除的！

DockerFile

DockerFile介绍

dockerfile 是用来构建docker镜像的文件！命令参数脚本！

构建步骤：

- 1、编写一个 dockerfile 文件
- 2、docker build 构建一个镜像
- 3、docker run 运行镜像
- 4、docker push 发布镜像（dockerHub、阿里云镜像仓库）

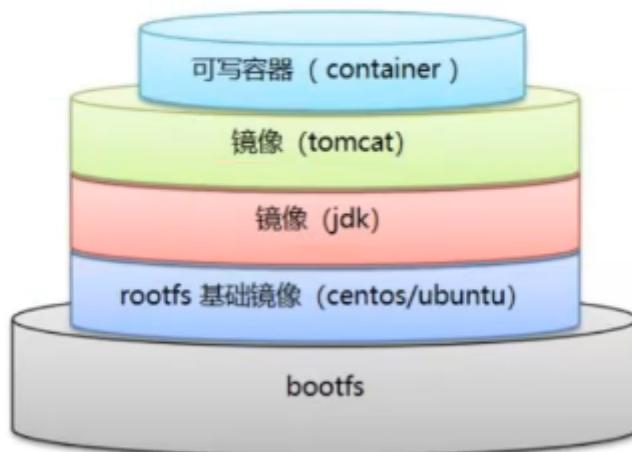
很多官方镜像都是基础包，很多功能没有，我们通常会自己搭建自己的镜像！

官方既然可以制作镜像，我们也可以！

DockerFile的制作过程

基础知识

- 1、每个保留关键字（指令）都是必须是大写字母
- 2、执行从上到下顺序执行
- 3、#表示注释
- 4、每一个指令都会创建提交一个新的镜像层，并提交！



dockerfile是面向开发的，我们以后要发布项目，做镜像，就需要编写dockerfile文件，这个文件十分简单！

Docker镜像逐渐成为企业交付的标准，必须要掌握！

DockerFile：构建文件，定义了一切的步骤，源代码

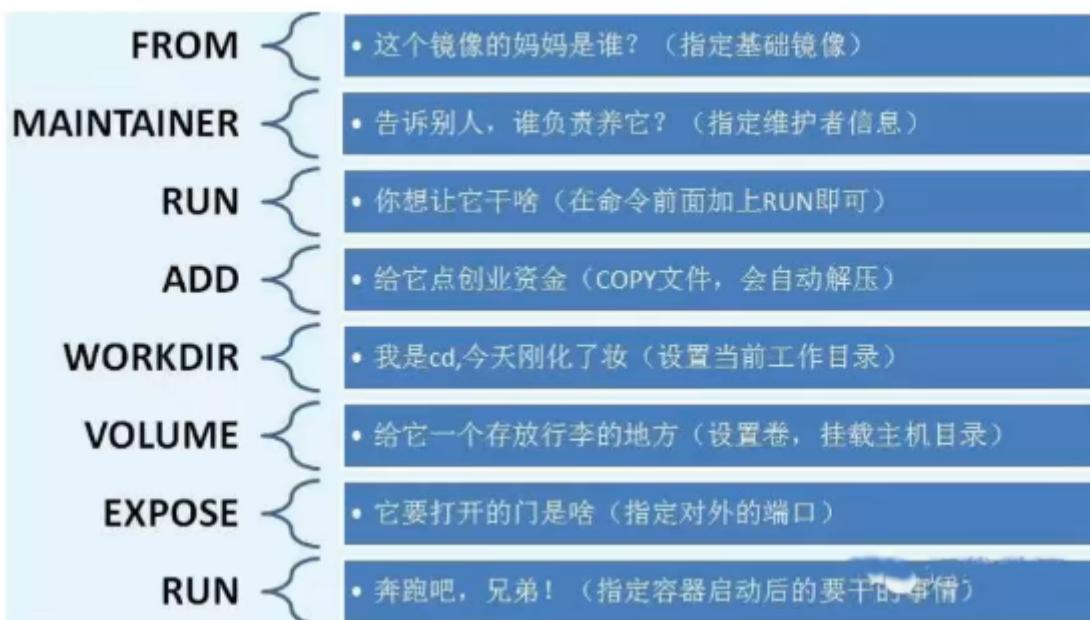
DockerImage：通过DockerFile构建生成的镜像，最终发布和运行的产品！原来是jar war，现在是直接通过Docker运行！

Docker容器：容器就是镜像运行起来提供服务的！

DockerFile指令

以前的话我们就是使用别人的，现在我们知道这些指令，我们来练习一个镜像！

```
1 FROM      # 基础镜像，一切从这里开始构建 需要centos
2 MAINTAINER # 镜像是谁写的，姓名+邮箱
3 RUN        # 镜像构建的时候需要运行的命令
4 ADD        # 步骤：tomcat镜像，需要一个tomcat压缩包！添加内容
5 WORKDIR   # 镜像的工作目录
6 VOLUME    # 挂载的目录
7 EXPOSE    # 暴露端口配置
8 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
9 ENTRYPOINT # 指定这个容器启动的时候需要运行命令，可以追加命令
10 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。
11 COPY      # 类似ADD ，将我们文件拷贝到镜像中
12 ENV       # 构建的时候设置环境变量！
```



实战测试

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

创建一个自己的centos

```
1 # 1、编写DockerFile的文件，编写脚本
2 vim mydockerfile-centos
3
4 # 指定一个基础镜像
5 FROM centos
```

```
6 MAINTAINER QianXunJian<2589857361@qq.com>
7
8 ENV MYPATH /usr/local
9 WORKDIR $MYPATH
10
11 RUN yum -y install vim
12 RUN yum -y install net-tools
13
14 EXPOSE 80
15
16 CMD echo $MYPATH
17 CMD echo "---end---"
18 CMD /bin/bash
19
20 # 2、通过这个文件构件镜像
21 # 命令 docker build -f dockerfile文件路径 -t 镜像名:[tag] .
22 docker build -f mydockerfile-centos -t mycentos0.1 .
23
24 # 3、测试运行
25 # 开启容器， -it 在后台运行
26 docker run -it mycentos:0.1
```

我们可以列出本地进行的变更历史

```
1 # 我们平时拿到镜像，可以研究一下它是怎么做的
2 docker history 镜像ID
```

CMD和ENTRYPOINT的区别

```
1 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
2 ENTRYPOINT # 指定这个容器启动的时候需要运行命令，可以追加命令
```

测试cmd

```
1 # 编写 dockerfile 文件
2 vim dockerfile-cmd-test
3
4 # 编写脚本
5 FROM centos
6 CMD ["ls", "-a"]
7
8 # 构建镜像
9 docker build -f dockerfile-cmd-test -t cmdtest .
10
11 # run运行，发我们的ls -a 命令生效
12 docker run 镜像ID
13
14 # 想追加一个命令 -l; ls -al
15 docker run 镜像ID -l
16 # cmd的情况下， -l 替换了CMD ["ls", "-a"]命令， -l 不是命令所以报错！
```

测试 ENTRYPOINT

```
1 # 编写dockerfile文件
2 vim dockerfile-cmd-entrypoint
3
```

```

4 # 2、编写脚本
5 FROM centos
6 ENTRYPOINT ["ls","-a"]
7
8 # 3、构建
9 docker build -f dockerfile-cmd-entrypoint -t entrypoint-test .
10
11 # 我们的追加命令，是直接凭借在我们的 ENTRYPOINT 命令的后面！
12 docker run 镜像ID -1
13
14

```

DockerFile中很多命令都十分相似，我们需要了解他们的的区别，我们最好的学习就是对比他们然后测试效果！

实战：Tomcat镜像

1、准备镜像文件 tomcat 压缩包， jdk的压缩包！

```

[root@kuangshen tomcat]# ll
total 165972
-rw-r--r-- 1 root root 10929702 May 12 12:31 apache-tomcat-9.0.22.tar.gz
-rw-r--r-- 1 root root 159019376 May 12 12:39 jdk-8ull-linux-x64.tar.gz

```

2、编写Dockefile文件，官方命名：Dockerfile， build会自动寻找这个文件，不需要 -f 指定了！

```

1 # 1、编写 Dockerfilewe 文件
2 vim Dockerfile
3
4 # 2、开始编写脚本
5 FROM centos
6
7 MAINTAINER qianxunjian<2589857361@qq.com>
8
9 COPY readme.txt /usr/local/readme.txt
10
11 ADD jdk-8ull-linux-x64.gz /usr/local/
12 ADD apache-tomcat-9.0.22.tar.gz /usr/local/
13
14 RUN yum -y install vim
15
16 ENV MYPATH /usr/local
17 WORKDIR $MYPATH
18
19 ENV JAVA_HOME /usr/local/jdk1.8.0_11
20 ENV CLASSPATH $JAVA_HOME/lib/rt.jar:$JAVA_HOME/lib/tools.jar
21 ENV CATALINA_HOME /usr/local/apache-tomcat-9.0.22
22 ENV CATALINA_BASH /usr/local/apache-tomcat-9.0.22
23 ENV PATH $PATH:$JAVA_HOME/bin:$CATALINA_HOME/lib:$CATALINA_HOME/bin
24
25 EXPOSE 8080
26
27 CMD /usr/local/apache-tomcat-9.0.22/bin/startup.sh && tail -F
28 /url/local/apache-tomcat-9.0.22/bin/logs/catalina.out
29
30 # 3、构建镜像，由于官方命名，所以不用 -f 指定名字
31 docker build -t diytomcat .
32

```

```
32 # 4、启动
33 -d # 后台运行
34 -p # 暴露端口
35 -v # 挂载 本地目录:容器内目录
36 docker run -d -p 9090:8080 --name qianxunjiantomcat -v
    /home/kuangshen/build/tomcat/test:/usr/local/apache-tomcat-
    9.0.22/webapps/test -v
    /home/kuangshen/build/tomcat/tomcatLogs:/usr/local/apache-tomcat-9.0.22/logs
diytomcat
37
38 # 5、访问测试
39
40 # 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！）
41
42 # 项目部署成功，可以直接访问OK!
```

```
1 # 停止容器
2 docker stop 容器id
```

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惊喜那个来发布运行！

发布自己的镜像

[狂神P32 发布镜像到阿里云容器服务](#)

发布在阿里云镜像服务上

- 1、登录阿里云
- 2、找到容器镜像服务
- 3、创建命名空间
- 4、创建容器镜像
- 5、浏览阿里云

1 登录阿里云Docker Registry

```
$ sudo docker login --username=18225148644 registry.cn-beijing.aliyuncs.com
```

用于登录的用户名为阿里云账号全名，密码为开通服务时设置的密码。

您可以在访问凭证页面修改凭证密码。

2 从Registry中拉取镜像

```
$ sudo docker pull registry.cn-beijing.aliyuncs.com/bilibili-kuangshen/kuangshen-test:[镜像版本号]
```

3 将镜像推送到Registry

```
$ sudo docker login --username=18225148644 registry.cn-beijing.aliyuncs.com
$ sudo docker tag [ImageId] registry.cn-beijing.aliyuncs.com/bilibili-kuangshen/kuangshen-test:[镜像版本号]
$ sudo docker push registry.cn-beijing.aliyuncs.com/bilibili-kuangshen/kuangshen-test:[镜像版本号]
```

请根据实际镜像信息替换示例中的[ImageId]和[镜像版本号]参数。

4. 选择合适的镜像仓库地址

从ECS推送镜像时，可以选择使用镜像仓库内网地址，推送速度将得到提升并且将不会损耗您的公网流量。

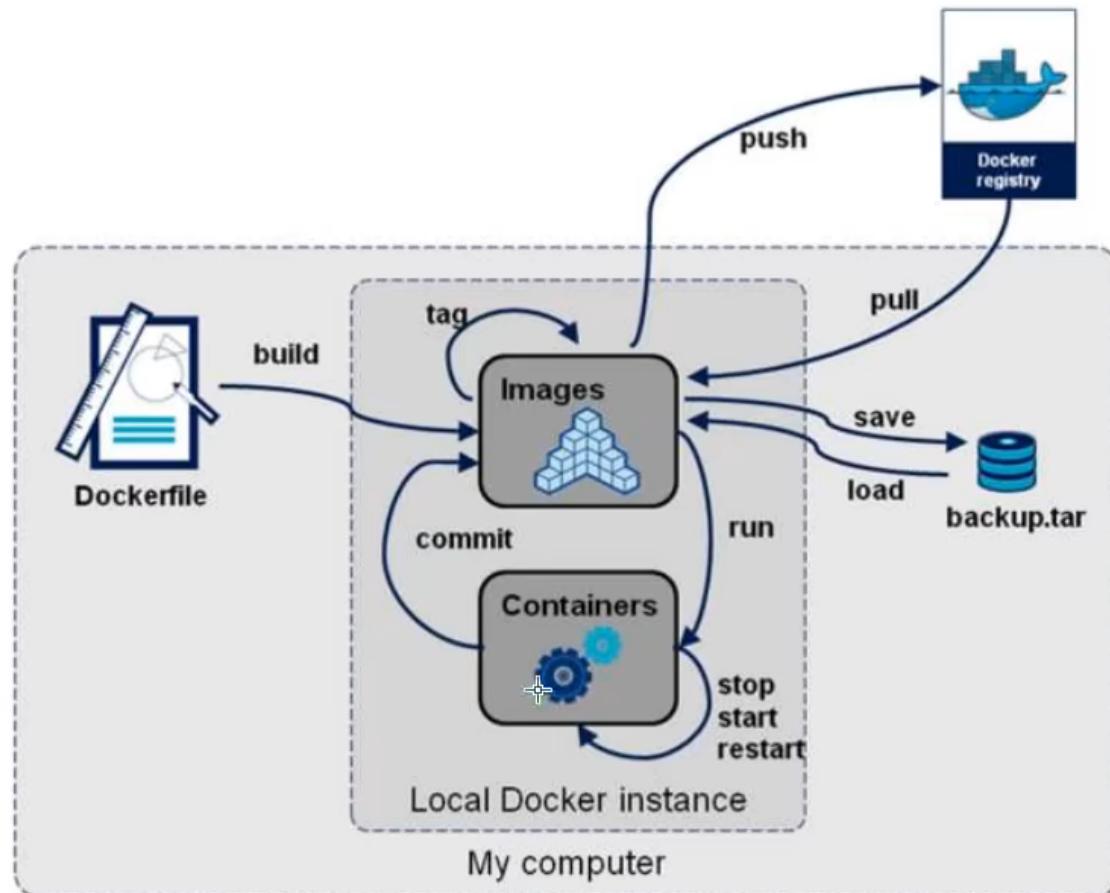
如果您使用的机器位于VPC网络，请使用 registry-vpc.cn-beijing.aliyuncs.com 作为Registry的域名登录，并作为镜像命名空间前缀。

5 示例

使用“docker tag”命令重命名镜像，并将它通过专有网络地址推送至Registry。

```
$ sudo docker images
REPOSITORY          TAG      IMAGE ID      CREATED       VIRTUAL SIZE
registry.aliyuncs.com/acs/agent      0.7-dfb6816   37bb9c63c8b2  7 days ago   37.89 MB
```

小节



Docker网络

理解Docker0

清空所有环境

```
1 # 删除所有容器
2 docker rm -f $(docker ps -aq)
3
4 # 删除所有镜像
5 docker rmi -f $(docker images -aq)
```

ip addr 查看到有三个网络

```
[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
```

问题：docker 是如何处理容器网络访问的？



```
1 docker run -d -p --name tomcat01 tomcat
2
3 # 查看容器的内部网络地址ip addr, 发现容器启动的时候会得到一个eth0 ip地址, docker分配的
4 docker exec -it tomcat01 ip addr
5
6 # 思考: linux 能不能ping通容器内部!
7 # 答: linux 可以 ping 通 docker 容器内部
8
```

原理

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

```
[root@kuangshen /]# 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 310744401sec preferred_lft 310744401sec
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP 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
262: veth96781fd@if261: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether da:8c:90:81:b5:f1 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

2、在启动一个容器,发现又多了一对网卡！

```

1 # 启动容器 run
2 docker run -d -P -name tomcat02 tomcat
3
4 # 进入容器 -it后台运行
5 docker exec -it tomcat02 ip addr
6
7 # 我们发现这个容器带来网卡，都是一对对的
8 # evth-pair 就是一队的虚拟设备接口，他们都是成对出现的，一段连着下一，一段彼此相连
9 # 正因为这个特性，evth-pair 充当一个桥梁，连接各种虚拟网络设备的
10 # OpenStack, Docker容器之间的连接，ovs的连接，都是使用 evth-pair 技术

```

```

[root@kuangshen ~]# docker run -d -P --name tomcat02 tomcat
312857784cd4b502208f1292de0b197315be0a43f31b840039c1008ea7543cd7
[root@kuangshen ~]# 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:00:00:00 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global dynamic eth0
        valid_lft 310744325sec preferred_lft 310744325sec
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP 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
262: vethc96781f@if261: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether da:8c:90:81:85:f1 brd ff:ff:ff:ff:ff:ff link-netnsid 0
264: veth2d06adc@if263: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether b0:26:dc:e5:ee:c0 brd ff:ff:ff:ff:ff:ff link-netnsid 1

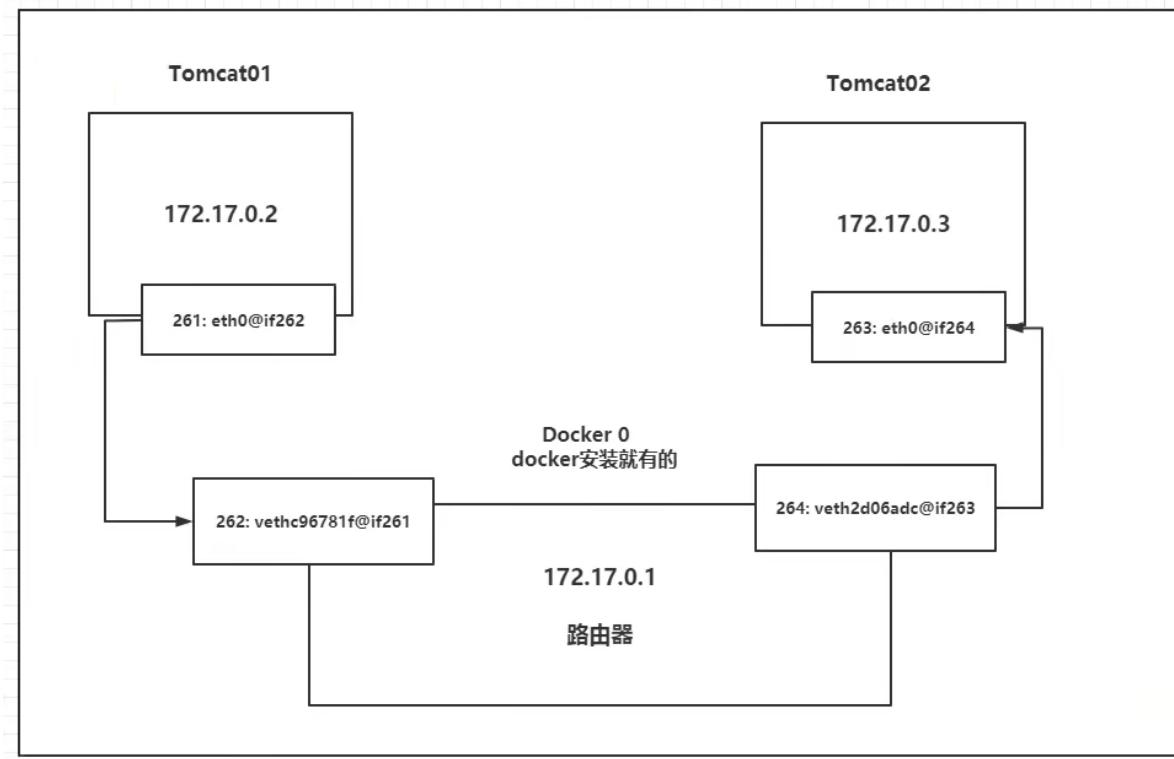
```

3、我们来想一下tomcat01和tomcat02是否可以 ping 通？

```

1 docker exec -it tomcat02 ping 172.18.0.2
2
3 # 结论：容器和容器之间是可以相互ping通的！

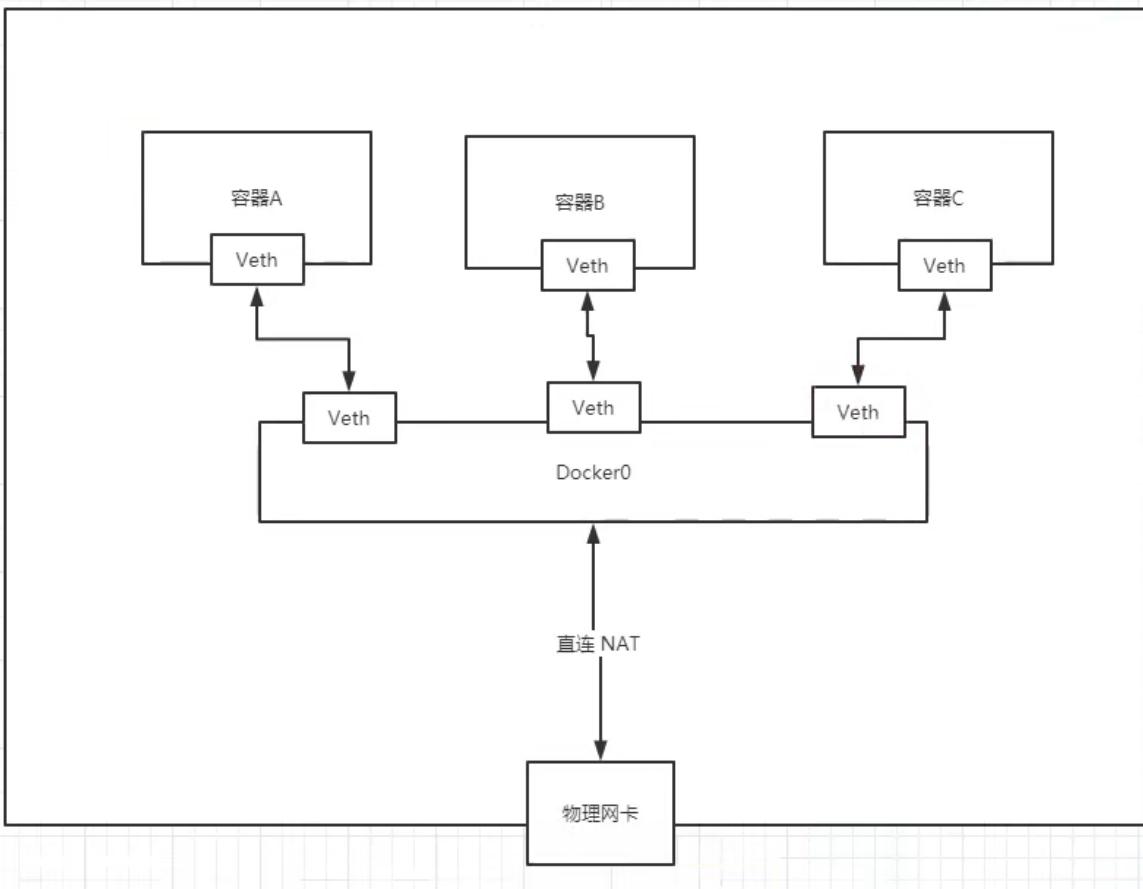
```



结论：tomcat01 和 tomcat02 是公用的一个路由器，docker0。

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

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



Docker 中的所有的网络接口都是虚拟的。虚拟的转发效率高！（内网传输）

只要容器删除，对应网桥一对没了！

--link

高可用 思考一个场景，我们编写一个微服务，database url = ip;项目不重启，数据库ip换掉了，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？

```
1 docker exec -it tomcat02 ping tomcat01
2 # ping 不通
3
4 # --link 命令链住
5 docker run -d -P --name tomcat03 --link tomcat02 tomcat
6
7 docker exec -it tomcat03 ping tomcat02
8 # ping 通了
9
10 # 反向可以ping通吗？
11 docker exec -it tomcat02 ping tomcat03
12 # 反向 ping 不通！
13
14 # 查看网络的详细信息
15 docker network inspect [NETWORK ID]
16
```

探究: inspect!

```
1 hosts  
2  
3 # 查看tomcat03 的网络信息  
4 docker exec -it tomcat03 cat /etc/hosts
```

```
[root@kuangshen /]# docker exec -it tomcat03 cat /etc/hosts  
127.0.0.1 localhost  
::1 localhost ip6-localhost ip6-loopback  
fe00::0 ip6-localnet  
ff00::0 ip6-mcastprefix  
ff02::1 ip6-allnodes  
ff02::2 ip6-allrouters  
172.18.0.3 tomcat02 312857784cd4  
172.18.0.4 5ca72d80ebb0
```

--link 就是在我们的hosts配置中增加了一个172.18.0.3 tomcat 02 容器id
所以我们在tomcat03 中才可以通过容器名称或者容器ID去 ping 通 tomcat02

--link 现在Docker已经不推荐使用了

自定义网络! 不适用docker0!

docker0问题: 他不支持容器名连接访问

扩展: springcloud feign = 服务名

自定义网络

```
1 # 查看所有的Docker网络  
2 docker network ls  
3  
4 # 移除网络  
5 docker network rm [NAME]
```

网络模式

bridge: 桥接 docker (默认, 自己创建也使用bridge模式)

none: 不配置网络

host: 和宿主机共享网络

container: 容器网络连通! (用的少! 局限性很大)

测试

```

1 # 我们直接启动的命令，默认就有--net bridge，而这个就是我们的coker0
2 docker run -d -P --name tomcat01 tomcat 等于
3 docker run -d -P --name tomcat01 --net bridge tomcat
4
5 # docker0特点：默认，域名不能访问， --link可以打通连接！
6
7 # 我们可以自定义一个网络！
8 docker network create --driver bridge --subnet 192.168.0.0/16 --gateway
192.168.0.1 mynet
9
10 # 查看dokcer网络
11 docker network ls

```

```

[root@kuangshen /]# docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
5a008c015cac   bridge    bridge      local
db44649a9bff   comosetest_default bridge      local
ae2b6209c2ab   host      host       local
eb21272b3a35   mynet     bridge      local

```

```

1 # 查看网络信息
2 docker network inspect [NAME]

```

```

[root@kuangshen /]# docker network inspect mynet
[
  {
    "Name": "mynet",
    "Id": "eb21272b3a35ceaba11b4aa5bbff131c3fb09c4790f0852ed4540707438db052",
    "Created": "2020-05-15T23:19:21.771943838+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": {}
  }
]

```

```

1 # 使用自己的网络
2 --net [网络NAME]
3 docker run -d -P --name tomcat-net-01 --net mynet tomcat
4 docker run -d -P --name tomcat-net-02 --net mynet tomcat
5
6 # 自定义的网络，对比docker0网络，自定义可以直接ping名字
7 docker exec -it tomcat-net-01 ping tomcat-net-02

```

结论：我们自定义的网络docker都已经帮我们维护好了我们对应的关系，推荐我们平时这样使用网络！

好处：搭建集群

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

网络连通

```
1 # 容器连接到一个网络上
2 docker network connect mynet tomcat01
3
4 # 查看mynet的网络情况
5 docker network inspect mynet
6
7 # 连通之后就是将 tomcat01 放到了 mynet 网络下
8
9 # 官方解释：一个容器两个ip地址！
10 # 阿里云服务：公网ip 私网ip
11
12 # 测试：
13 # 01连通了mynet
14 # 02没有联通
```

```
[root@kuangshen /]# docker exec -it tomcat01 ping tomcat-net-01
ping: tomcat-net-01: Name or service not known
[root@kuangshen /]# docker network --help
```

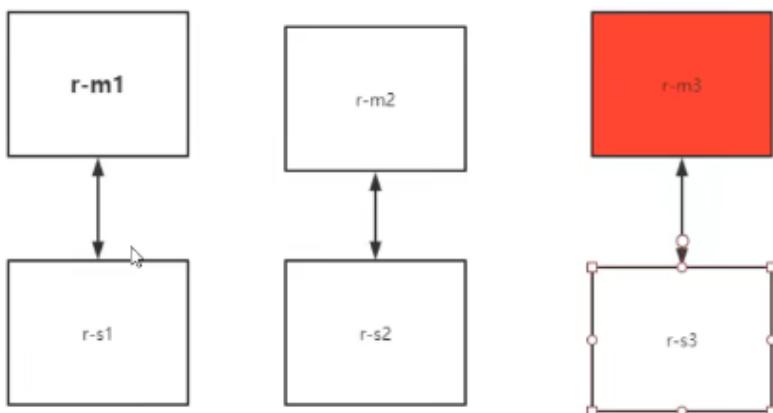
```
Usage: docker network COMMAND

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
```

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

实战：Redis集群

分片 + 高可用 + 负载均衡



```
1 # 移除容器
2 docker rm -f $(docker ps -aq)
3
4 # 创建一个网卡
5 docker network create redis --subnet 173.38.0.0/16
6
7 # 查看网络信息
8 docker network ls
9 docker network inspect redis
10
11 # 通过脚本创建六个redis配置
12 for port in $(seq 1 6):\n13 do \
14 mkdir -p /mydata/redis/node-$port/conf
15 touch /mydata/redis/node-$port/conf/redis.conf
16 cat << EOF >>/mydata/redis/node-$port/conf/redis.conf
17 port 6379
18 bind 0.0.0.0
19 cluster-enabled yes
20 cluster-config-file nodes.conf
21 cluster-node-timeout 5000
22 cluster-announce-ip 172.38.0.1${port}
23 cluster-announce-port 6379
24 cluster-announce-bus-port 16379
25 appendonly yes
26 EOF
27 done
28
29 # 启动--编程脚本
30 docker run -p 637${port}:6379 -p 1637${port}:16379 --name redis-$port \
31 -v /mydata/redis/node-$port/data:/data \
32 -v /mydata/redis/node-$port/conf/redis.conf:/etc/redis/redis.conf \
33 -d --net reddit --ip 172.38.0.1${port} redis:5.0.9-alpine3.11 redis-server \
34 /etc/redis/redis.conf;
```

```
35 # 启动--手动启动, 改
36 docker run -p 6371:6379 -p 16371:16379 --name redis-1 \
37 -v /mydata/redis/node-1/data:/data \
38 -v /mydata/redis/node-1/conf/redis.conf:/etc/redis/redis.conf \
39 -d --net redidis --ip 172.38.0.11 redis:5.0.9-alpine3.11 redis-server
40 /etc/redis/redis.conf
41 # 创建集群
42 redis-cli --cluster create 172.38.0.11:6379 172.38.0.12:6379
43 172.38.0.13:6379 172.38.0.14:6379 172.38.0.15:6379 172.38.0.16:6379 --
44 cluster-relicas 1
45 # 询问是否这么配置, 输入yes
46
47 # 进入容器, redis中是没有bash命令的, 用的sh命令
48 docker exec -it reidis-1 /bin/sh
49
50 # 测试
51 redis-cli -c
52
53 # 集群的信息
54 cluster info
55
56 # 集群的节点
57 cluster nodes
58
59 # 新增一个值
60 set a b
61
62 # 停止一个redis, 测试是否可以从机顶上来
63 docker stop redis-3
64
65 # 测试获取a的值, 发现他是从从机上获取的, 实现了高可用
66 get a
```

docker搭建redis集群完成!

我们使用了docker之后，素有急速都会慢慢的变得简单起来！

SpringBoot微服务打包Docker镜像

视频教程：[狂神视频](#)

- 1、构建springboot项目
- 2、打包应用
- 3、编写dockerfile

```
Dockerfile x
1 >> FROM java:8
2
3   COPY *.jar /app.jar
4
5   CMD ["--server.port=8080"]
6
7   EXPOSE 8080
8
9   ENTRYPOINT ["java","-jar","/app.jar"]
```

4、构建镜像

5、发布运行

idea安装插件 搜索docker

##

企业实战Docker Compose

简介

DockerFile build run 手动操作，单个容器！

微服务，100个微服务！依赖关系。

Docker Compose 来轻松高效管理容器！定义运行多个容器。

官方介绍

定义、运行多个容器。

YAML file 配置文件。

single command 命令有哪些？

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see [the list of features](#).

所有的环境都可以使用 Compose

Compose works in all environments: production, staging, development, testing, as well as CI workflows. You can learn more about each case in [Common Use Cases](#).

三步骤：

Using Compose is basically a three-step process:

1. Define your app's environment with a `Dockerfile` so it can be reproduced anywhere.
 - DockerFile 保证我们的项目在任何地方可以运行。
2. Define the services that make up your app in `docker-compose.yml` so they can be run together in an isolated environment.
 - service 什么是服务。
 - docker-compose.yml 这个文件怎么写！
3. Run `docker compose up` and the [Docker compose command](#) starts and runs your entire app. You can alternatively run `docker-compose up` using the docker-compose binary.
 - 启动项目

作用：批量容器编排。

狂神的理解

Compose 是 Docker 官方开源的项目。需要安装！

`Dockerfile` 让程序在任何的地方运行。web服务、redis、mysql...多个容器。

Compose

```
1 version: "3.9" # optional since v1.27.0
2 services:
3   web:
4     build: .
5     ports:
6       - "5000:5000"
7     volumes:
8       - ./code
9       - logvolume01:/var/log
10    links:
11      - redis
12    redis:
13      image: redis
14    volumes:
15      logvolume01: {}
```

`docker-compose up` 100个服务

Compose : 重要的概念。

- 服务service，容器。应用。（web、redis、mysql...）
- 项目project。一组关联的容器。博客（web、mysql）wp

安装

1、下载

```

1 sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.2/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
2
3 官方下载非常慢，我们使用第三方的下载地址，可能会快点
4 curl -L https://get.daocloud.io/docker/compose/releases /
download/1.25.5/docker-compose-"uname -s '- 'uname -m' >
/usr/local/bin/docker-compose

```

```

[root@kuangshen home]# curl -L https://get.daocloud.io/docker/compose/releases/download/1.25.5/docker-compose-`uname -s`-`u
name -m` > /usr/local/bin/docker-compose
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload Total Spent   Spent    Left Speed
100  423  100  423    0     0  403      0:00:01  0:00:01  ---:---  403
100 16.7M  100 16.7M   0     0  8188k      0:00:02  0:00:02  ---:--- 21.0M
[root@kuangshen home]# ^C
[root@kuangshen home]# cd /usr/local/bin
[root@kuangshen bin]# ll
total 54928
-rwxr-xr-x 1 root root 17586312 Jul 24 20:39 docker-compose
-rwxr-xr-x 1 root root 4739112 May 13 21:59 redis-benchmark
-rwxr-xr-x 1 root root 9617624 May 13 21:59 redis-check-aof
-rwxr-xr-x 1 root root 9617624 May 13 21:59 redis-check-rdb
-rwxr-xr-x 1 root root 5049928 May 13 21:59 redis-cli
lrwxrwxrwx 1 root root       12 May 13 21:59 redis-sentinel -> redis-server
-rwxr-xr-x 1 root root 9617624 May 13 21:59 redis-server
[root@kuangshen bin]#

```

2、授权

```

1 sudo chmod +x /usr/local/bin/docker-compose
2
3 # 测试安装是否成功
4 docker-compose version

```

体验

地址: <https://docs.docker.com/compose/gettingstarted/>

1、python 应用。计数器 redis

```

1 mkdir composetest
2 cd composetest
3
4 # 创建一个文件: app.py
5 # 编辑: app.py
6 import time
7
8 import redis
9 from flask import Flask
10
11 app = Flask(__name__)
12 cache = redis.Redis(host='redis', port=6379)
13
14 def get_hit_count():
15     retries = 5
16     while True:
17         try:
18             return cache.incr('hits')
19         except redis.exceptions.ConnectionError as exc:
20             if retries == 0:
21                 raise exc
22             retries -= 1
23             time.sleep(0.5)
24

```

```
25 @app.route('/')
26 def hello():
27     count = get_hit_count()
28     return 'Hello World! I have been seen {} times.\n'.format(count)
29
30 # 编辑自动创建 requirements.txt
31 vim requirements.txt
32
33 # 编辑器内
34 flask
35 redis
```

第二步：创建一个dockerfile，应用打包为镜像

```
1 vim Dockerfile
2
3 # syntax=docker/dockerfile:1
4 FROM python:3.7-alpine
5 WORKDIR /code
6 ENV FLASK_APP=app.py
7 ENV FLASK_RUN_HOST=0.0.0.0
8 RUN apk add --no-cache gcc musl-dev linux-headers
9 COPY requirements.txt requirements.txt
10 RUN pip install -r requirements.txt
11 EXPOSE 5000
12 COPY .
13 CMD ["python", "run"]
```

第三步：定义服务在 Compose file，（定义整个服务，需要的环境。web、redis）

```
1 vim docker-compose.yml
2
3 version: "3.9"
4 services:
5   web:
6     build: .
7     ports:
8       - "5000:5000"
9   redis:
10    image: "redis:alpine"
```

第四步：用Compose 构建和运行你的app，启动compose 项目（docker-compose up）

```
1 docker-compose up
```

流程：

- 1、创建网络
- 2、执行 docker-compose.yml
- 3、启动服务

```
Status: Downloaded newer image for redis
Creating composetest_web_1 ... done
Creating composetest_redis_1 ... done
```

默认规则：

```

root@kuangshen ~]# clear
[root@kuangshen ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
1ac14aab1a54      redis:alpine       "docker-entrypoint.s..."   3 minutes ago     Up 15 seconds
cp                composetest_redis_1
[root@kuangshen ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
4851571e3436      composetest_web    "python app.py"      31 seconds ago    Up 30 seconds
0:5000->5000/tcp  composetest_web_1
1ac14aab1a54      redis:alpine       "docker-entrypoint.s..."   6 minutes ago     Up 30 seconds
cp                composetest_redis_1
[root@kuangshen ~]# curl localhost:5000
Hello World! I have been seen 1 times.
[root@kuangshen ~]# curl localhost:5000
Hello World! I have been seen 2 times.
[root@kuangshen ~]# curl localhost:5000
Hello World! I have been seen 3 times.
[root@kuangshen ~]# curl localhost:5000
Hello World! I have been seen 4 times.
[root@kuangshen ~]# curl localhost:5000
Hello World! I have been seen 5 times.
[root@kuangshen ~]# curl localhost:5000
Hello World! I have been seen 6 times.
[root@kuangshen ~]#

```

服务启动成功!

服务正常

docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
composetest_web	latest	dcd9421baf78	8 minutes ago	81.3MB
<none>	<none>	2e50cd447752	15 minutes ago	73.1MB
redis	alpine	c7b388ce3d39	2 days ago	32.1MB
python	3.6-alpine	d6b500d78779	3 weeks ago	70.1MB
python	3.7-alpine	6ca3e0b1ab69	3 weeks ago	73.1MB
hello-world	latest	bf756fb1ae65	6 months ago	13.3kB

默认的服务器名 文件名 _服务名 _1

多个服务器。集群。A、B _num 副本数量

服务Redis 服务 => 4个副本。

集群状态。服务都不可能只有一个运行实例。弹性、10 HA 高并发。

kubectl service 负载均衡

3、网络规则

```

1 # 查看网络
2 docker network ls
3
4 # 查看网络箱详细信息
5 docker network inspect NAME

```

Docker network is not a docker command. See 'docker --help'			
NETWORK ID	NAME	DRIVER	SCOPE
7f2a3e7b5da0	bridge	bridge	local
caff1f59b938	composetest_default	bridge	local
r2965e19a00f	host	host	local
ffffebb0a079b	none	null	local

10个服务 ==> 项目 (项目中的内容都在通过网络下。域名访问)

```

    "Network": "",
  },
  "ConfigOnly": false,
  "Containers": {
    "1ac14aab1a545e735cf515f1aa5b0ac9aa8e2e66429e8e322c62ed87b0a370b1": {
      "Name": "composetest_redis_1",
      "EndpointID": "8df4a09d9f71ede71895c6a56daa6bfea071d35570df834a8ede386836365ce9",
      "MacAddress": "02:42:ac:1b:00:02",
      "IPv4Address": "172.27.0.2/16",
      "IPv6Address": ""
    },
    "4851571e3436171794527e16793986cb848418186051cbba8daa78e973ce872b": {
      "Name": "composetest_web_1",
      "EndpointID": "c3d027525e9947f83ed202de32a0bde7deb8278a99596fc7ba935438158c97fd",
      "MacAddress": "02:42:ac:1b:00:03",
      "IPv4Address": "172.27.0.3/16",
      "IPv6Address": ""
    }
  },
  "Options": {}
}

```

如果在同一个网络下，我们可以直接通过域名访问。

HA!

停止: docker-compose down / ctrl + c

```

web_1    | 172.27.0.1 - - [24/Jul/2020 13:04:52] "GET / HTTP/1.1"
web_1    | 172.27.0.1 - - [24/Jul/2020 13:04:53] "GET / HTTP/1.1"
web_1    | 172.27.0.1 - - [24/Jul/2020 13:04:54] "GET / HTTP/1.1"
^CGracefully stopping... (press Ctrl+C again to force)
Stopping composetest_web_1 ... done
Stopping composetest_redis_1 ... done
[root@kuangshen composetest]#

```

docker-compose

一起拿都是单个 docke run 启动容器。

docker-compose : 通过docker-compose 编写yaml配置文件、可以通过compose 一键启动所有服务，停止！

Docker小结:

- 1、Docker 镜像。run => 容器
- 2、Docker File 构建镜像（服务打包）
- 3、docker-compose 启动项目（编排、多个微服务/环境）
- 4、Docker 网络

yaml规则

docker-compose.yaml 核心！

<https://docs.docker.com/compose/compose-file/>

```

1 # 有且只有三层!
2
3 version: '' # 版本
4 service:     # 服务
5   服务1: web
6     # 服务配置 docker容器的配置
7     images
8     build

```

```
9      network
10     ...
11     服务2: redis
12     ...
13     服务3: redis
14     ...
15 # 其他配置 网络/卷、全局规则
16 volumes:
17 networks:
18 configs:
```

多看多写。compose.yaml 配置!

1、官方文档

<https://docs.docker.com/compose/compose-file/compose-file-v3/>

2、开源项目 compose.yaml

redis、mysql、mq

开源项目（启动）

博客

下载程序、安装数据库、配置...

compose 应用。=>一键启动

<https://docs.docker.com/samples/wordpress/>

1、下载项目（docker-compose.yaml）

2、如果需要文件 Dockerfile

3、文件准备齐全（直接一件启动）

```
1 # 后台启动
2 docker-compose up -d
3
4 # 停止
5 docker-compose down
```

掌握：docker基础，原理，网络，服务，集群，错误排查，日志。

linux、docker、k8s 12k-20k

实战：

1、编写项目微服务，确保项目本地可以运行

2、编写接口地址：采用服务名替换服务器地址 redis、mysql

3、编写dockerfile 构建镜像

```
1 FROM java:8
2
3 COPY *.jar /app.jar
4
5 CMD ["--server.port=8080"]
6
7 EXPOSE 8080
8
9 ENTRYPOINT ["java","-jar","/app.jar"]
```

4、编写docker-compose.yaml 编排项目

```
1 version: '3.8'
2 service:
3     chihiroapp
4         build: .
5         image:chihiroapp
6         depends_on:
7             - redis
8         ports:
9             - "8080:8080"
10    redis
11        image: "redis:alpine"
```

5、上传到服务器， docker-compose up

假设项目要重新部署打包

```
1 docker-compose up --build # 重新构建
```

小结：

未来项目只要有 docker-compose 文件。按照这个规则，启动编排容器。

公司： docker-compose 直接启动

网上开源项目： docker-compose up 一键启动

总结：

工程、服务、容器

项目compose：三层

- 工程 Project
- 服务 Service
- 容器 Container 运行实例！ docker k8s 容器

Docker Swarm

集群方式的部署

购买服务器

4台阿里云服务、2核4g

到此，服务器购买完毕！1主，3从！

4台机器安装Docker

发送键到所有会话



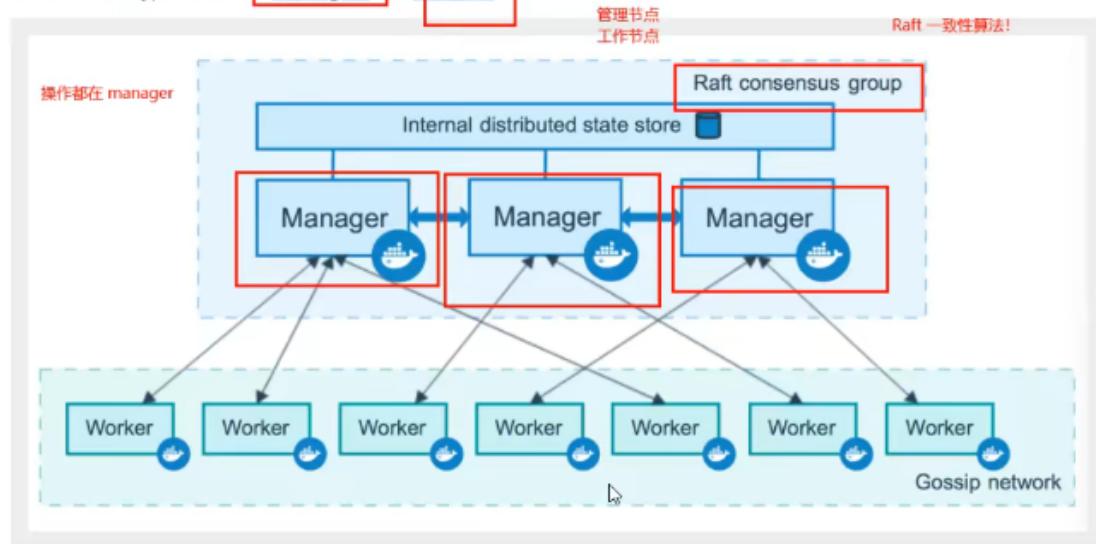
```
1 # 安装看前面 docker 单机教程  
2 # 技巧：使用xshell直接同步操作，省时间
```

官网地址：<https://docs.docker.com/engine/swarm/>

工作模式

mode.

There are two types of nodes: **managers** and **workers**.



If you haven't already, read through the [swarm mode overview](#) and [key concepts](#).

搭建集群

```
1 # 查看网络
2 docker network ls
```

```
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
6ddf7e46cff4    bridge    bridge      local
cd4a3e83d894    host      host       local
820d7c2ce5a3    none      null       local
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]#
```

```
1 # 查看 swarm 命令
2 docker swarm --help
3
4 # 初始化 swarm 节点
5 docker swarm init --help
```

```
820d/czcesas      none      null      local
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker swarm --help
Usage: docker swarm COMMAND

Manage Swarm

Commands:
  ca      Display and rotate the root CA
  init   Initialize a swarm
  join   Join a swarm as a node and/or manager
  join-token Manage join tokens
  leave  Leave the swarm
  unlock Unlock swarm
  unlock-key Manage the unlock key
  update Update the swarm

Run 'docker swarm COMMAND --help' for more information on a command.
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]#
```

```
1 # 初始化工作（主节点）节点
2 docker swarm init --advertise-addr 主节点内网地址
```

```
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker swarm init --advertise-addr 172.24.82.149
Swarm initialized: current node (t3jgzjpkhks7jj788rjtxuu3y) is now a manager.

To add a worker to this swarm, run the following command:

  docker swarm join --token SWMTKN-1-4alqo02iwnaoqbt9vuagbucuslkk9p45lu3apf7ed7ybm50a3x-dkmn5q0ocys8lc91222xerwx 172.24.82.149:2377

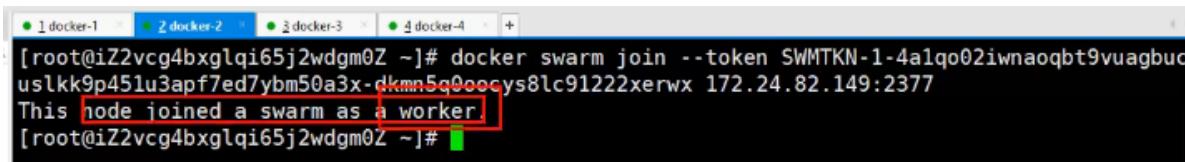
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

[root@iZ2vcg4bxglqi65j2wdgm3Z ~]#
```

初始化节点 `docker swarm init`

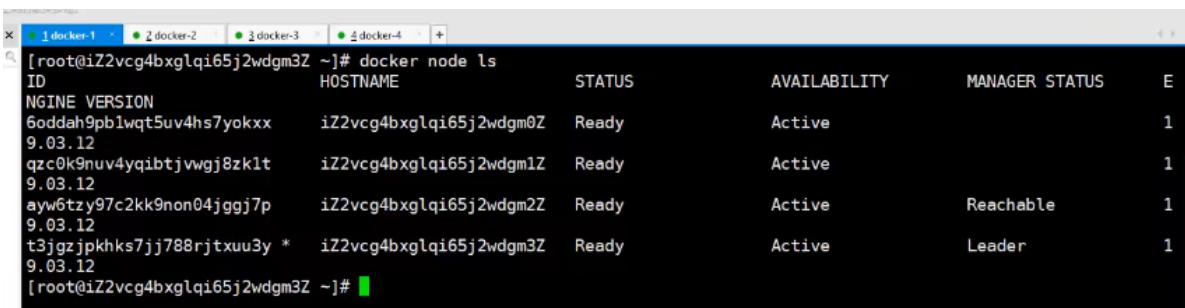
`docker swarm join` 加入一个节点!

```
1 # 获取令牌
2 # 创建一个管理者的 token 令牌 manager
3 docker swarm join-token manager
4 # 创建一个工作者的 token 令牌 worker
5 docker swarm join-token worker
```



```
[root@iZ2vcg4bxglqi65j2wdgm0Z ~]# docker swarm join --token SWMTKN-1-4alqo02iwnaoqbt9vuagbucuslkk9p45lu3apf7ed7ybm50a3x-dkmn5q0ocys8lc91222xerwx 172.24.82.149:2377
This node joined a swarm as a worker.
[root@iZ2vcg4bxglqi65j2wdgm0Z ~]#
```

```
1 # 查看节点信息
2 docker node ls
```



ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS	E
6oddah9pb1wqt5uv4hs7yokxx 9.03.12	iZ2vcg4bxglqi65j2wdgm0Z	Ready	Active		1
qzc0k9nuv4yq1btjvwgj8zk1t 9.03.12	iZ2vcg4bxglqi65j2wdgm1Z	Ready	Active		1
ayw6tz97c2kk9non04jggj7p 9.03.12	iZ2vcg4bxglqi65j2wdgm2Z	Ready	Active	Reachable	1
t3jgzjpkhks7jj788rjtxuu3y 9.03.12	iZ2vcg4bxglqi65j2wdgm3Z	Ready	Active	Leader	1

步骤:

1、生成主节点init

2、加入 (管理者、 worker)

目前是: 双主双从!

Raft协议

双主双从：假设一个节点挂了！其他节点是否可用！

Raft协议：保证大多数节点存活才可用。只要>1，集群至少大于3台！

实验：

1、将 docker1 机器停止。模拟宕机！双主，另外一个主节点也不能使用了！`systemctl stop docker` 命令模拟宕机。

```
[root@iZ2vcg4bxglqi65j2wdgm2Z ~]# docker node ls
Error response from daemon: rpc error: code = Unknown desc = The swarm does not have a leader. It's possible
that too few managers are online. Make sure more than half of the managers are online.
[root@iZ2vcg4bxglqi65j2wdgm2Z ~]#
```

2、可用将其他节点离开

```
1 # 离开集群  
2 docker swarm leave
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STAT
6oddah9pb1wqt5uv4hs7yokxx	iZ2vcg4bxglqi65j2wdgm0Z	Ready	Active	
19.03.12				
qzc0k9nuv4yqibtjvwgj8zk1t	iZ2vcg4bxglqi65j2wdgm1Z	Down	Active	
19.03.12				
ayw6tzy97c2kk9non04jggj7p *	iZ2vcg4bxglqi65j2wdgm2Z	Ready	Active	Leader
19.03.12				
t3jgzjpkhks7jj788rjtxuu3y	iZ2vcg4bxglqi65j2wdgm3Z	Ready	Active	Reachable
19.03.12				

3、在创建一个管理者，要在管理者的机子中生成

```
1 # 创建一个管理者的 token 令牌 manager  
2 docker swarm join-token manager  
3  
4 # work就是工作的，没办法用管理者的命令！
```

```
[root@iZ2vcg4bxglqi65j2wdgm1Z ~]# docker swarm join --token SWMTKN-1-4a1qo02iwnaoqbt9vuagbucuslkk9p  
451u3apf7ed7ybmm50a3x-6ekudx8sjj0fnkhp9zk5xjch3 172.24.82.149:2377  
This node joined a swarm as a manager.  
[root@iZ2vcg4bxglqi65j2wdgm1Z ~]#
```

小节：集群，可用！3个主节点。>1台管理节点存活！

Reft协议：保证大多数节点存活，才可以使用。

服务

弹性、扩缩容！

以后告别docker run！

docker-compose up! 启动一个项目。单机

集群：swarm **docker service**

容器 => 服务！

容器 => 服务！=>副本！

redis服务 => 10个副本（同时开启10个redis容器）

体验：创建服务、动态扩展服务、动态跟新服务、日志。

```
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker service --help
Usage: docker service COMMAND

Manage services

Commands:
  create          Create a new service
  inspect         Display detailed information on one or more services
  logs            Fetch the logs of a service or task
  ls              List services
  ps              List the tasks of one or more services
  rm              Remove one or more services
  rollback        Revert changes to a service's configuration
  scale           Scale one or multiple replicated services
  update          Update a service

Run 'docker service COMMAND --help' for more information on a command.
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]#
```

灰度发布：金丝雀发布！

企业中很少停止网站发布：401！

```
1 # 容器启动！不具有扩缩容
2 docker run
3
4 # 服务！具有扩容，滚动更新！
5 docker service
6
7 # 创建一个服务
8 docker service create -p 8888:80 --name my-nginx nginx
9
10 # 查看一个服务 REPLICAS 显示副本数
11 docker service ps 服务名称
```

```
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker service ps my-nginx
ID           NAME      IMAGE          NODE          DESIRED STATE   CURRENT STA
TE           ERROR     PORTS          iZ2vcg4bxglqi65j2wdgm1Z  Running
0l2hvlvg5bre  my-nginx.1  nginx:latest
ut a minute ago
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker service ls
ID           NAME      MODE          REPLICAS    IMAGE          PORTS
34wxyynfryorr  my-nginx  replicated  1/1        nginx:latest  *:8888->80/tcp
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]#
```

```
1 # 查看服务详细信息
2 docker service inspect NAME
3
4 # 动态扩缩容
5 docker service update --replicas 3 NAME
```

服务在集群中任意的节点都可以访问。服务可以有多个副本动态扩缩容实现高可用！

```
1 # 动态扩缩容 和 update效果是一样的
2 docker service scale my-nginx=5
```

```

my-nginx          replicated    1/1
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker service scale my-nginx=5
my-nginx scaled to 5
overall progress: 5 out of 5 tasks
1/5: running [=====>]
2/5: running [=====>]
3/5: running [=====>]
4/5: running [=====>]
5/5: running [=====>]
verify: Service converged
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker service scale my-nginx=2
my-nginx scaled to 2
overall progress: 2 out of 2 tasks
1/2: running [=====>]
2/2: running [=====>]
verify: Service converged

```

```

1 # 移除服务
2 docker service rm my-nginx

```

docker swarm 其实并不难

只要会搭建集群、会启动服务、动态管理容器就可以了！

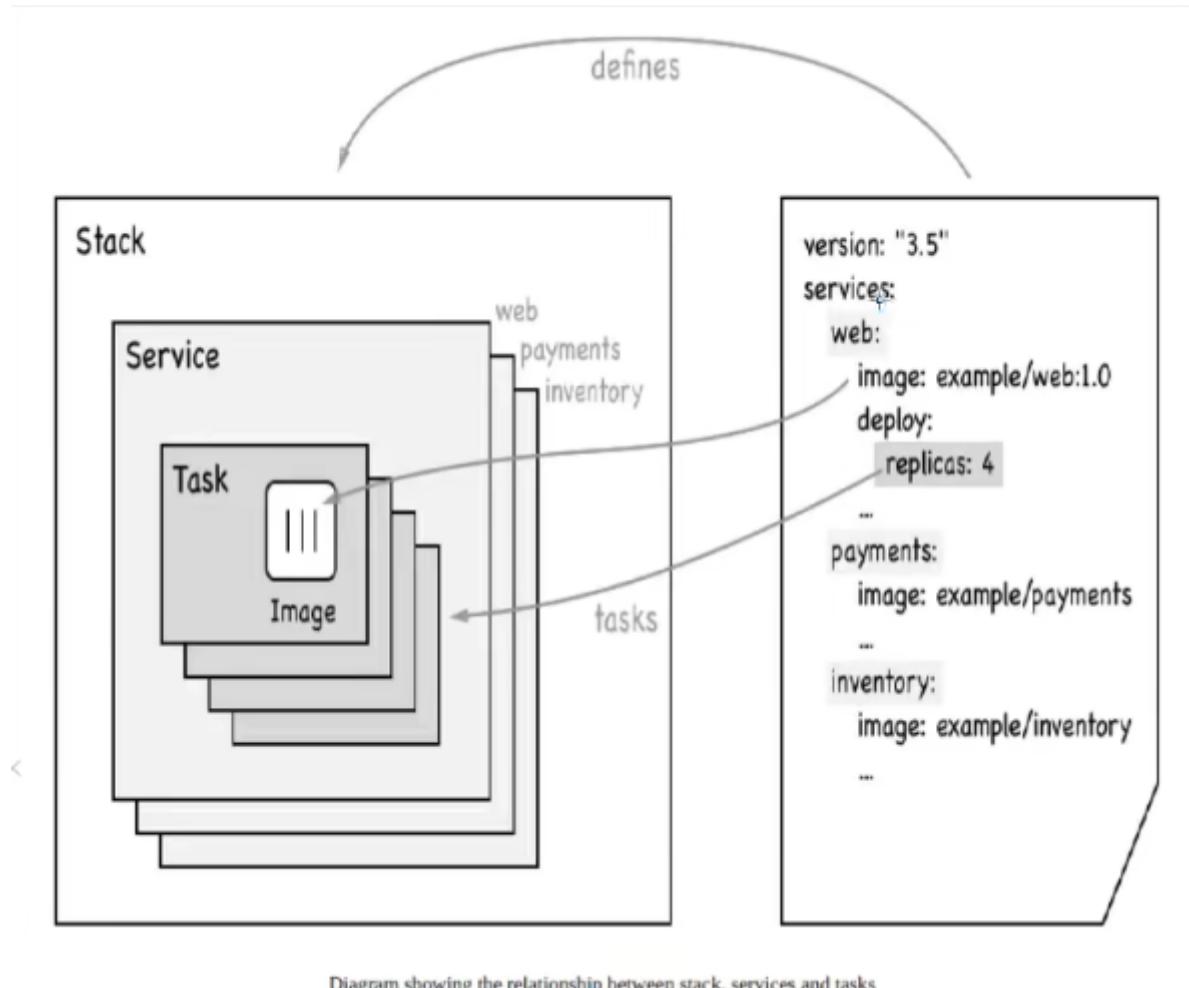
概念总结

swarm: 集群的管理和编排。 docker 可以初始化一个swarm集群，其他节点可以加入。 (管理、工作者)

Node: 就是一个docker节点。 多个节点就组成了一个网络集群。 (管理、工作者)

service: 任务，可以在管理节点或者工作节点运行。 核心！ 用户访问！

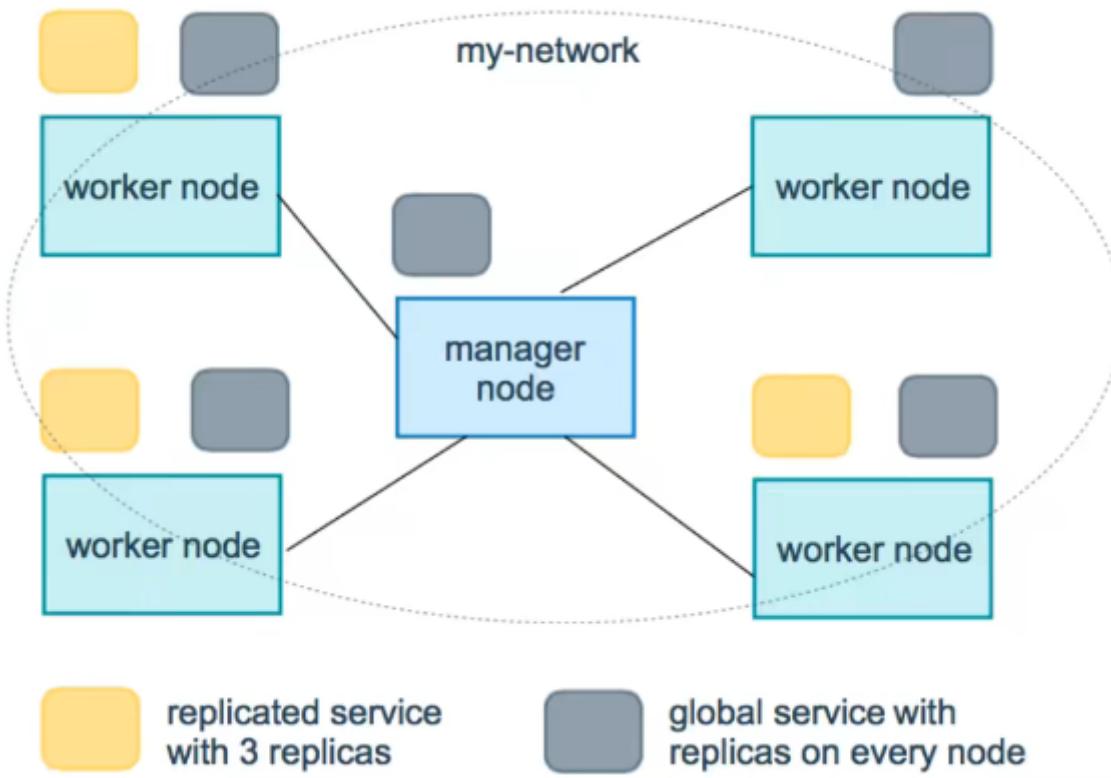
Task: 容器内的命令，细节任务



逻辑是不变的

命令->管理->api->调度->工作节点（创建Task容器维护创建！）

服务副本与全局服务



调整service 以什么方式运行

```
1 --mode string
2 service mode (replicated or global)(default "replicated")
3
4 docker service create --mode replicated --name mytom tomcat:7 默认的
5
6 docker service create --mode global --name haha alpine ping baidu.com
7
8 #场景? 日志收集
9 每一个接地带有自己的日志收集器, 过滤。把所有日志最终再传给日志中心
10 服务监控。状态性能。
```

Docker Stack

docker-compose 单机部署项目！

Docker Stack 部署，集群部署！

```
1 # 单机  
2 docker-compose up -d wordpress.yaml  
3  
4 # 集群  
5 docker stack deploy wordpress.yaml  
6  
7 # docker-compose 文件  
8
```

Docker Secret

安装！配置密码，证书！

```
services      List the services in the stack  
  
Run 'docker stack COMMAND --help' for more information on a command.  
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker secret  
  
Usage:  docker secret COMMAND  
  
Manage Docker secrets  
  
Commands:  
  create      Create a secret from a file or STDIN as content  
  inspect     Display detailed information on one or more secrets  
  ls          List secrets  
  rm          Remove one or more secrets  
  
Run 'docker secret COMMAND --help' for more information on a command.  
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]#
```

Docker config

配置

```
[root@iZ2vcg4bxglqi65j2wdgm3Z ~]# docker config  
  
Usage:  docker config COMMAND  
  
Manage Docker configs  
  
Commands:  
  create      Create a config from a file or STDIN  
  inspect     Display detailed information on one or more configs  
  ls          List configs  
  rm          Remove one or more configs  
  
Run 'docker config COMMAND --help' for more information on a command.
```

Docker compose Swarm！

学习方式：

网上找案例跑起来试试！查看命令帮助文档 --help，官网！

k8s docker go

扩展到K8S

云原生时代 大趋势！

Go语言！必须掌握！Java、Go！

Docker 是 Go 开发的

k8s 也是 Go 开发的

Go 是并发语言！

B 语言 C 语言的创始人。 Unix 创始人

云应用

电商网站

在线教育平台...

下载 => 配置！