目录

[1. Docker简介 2](#_Toc18159)

[2. Docker的架构 2](#_Toc28104)

[3. Centos7安装docker 2](#_Toc560)

[3.1 卸载老版本的docker 3](#_Toc17228)

[3.2 安装仓库 3](#_Toc31088)

[3.2.1 执行以下命令，安装docker所需的包 3](#_Toc19785)

[3.2.2 执行以下命令，安装stable仓库 3](#_Toc1758)

[3.2.3 执行以下命令，启用edge及test仓库(可选) 3](#_Toc13040)

[4. 安装docker ce 4](#_Toc31676)

[4.1 执行以下命令，更新yum的包索引 4](#_Toc29696)

[4.2 执行以下命令，安装最新版本的docker ce 4](#_Toc9773)

[4.3 启动docker 4](#_Toc17900)

[4.4 验证安装是否正确 5](#_Toc28233)

[4.5 升级docker 5](#_Toc7960)

[5. 配置镜像加速器 5](#_Toc3304)

[6．Docker常用命令 6](#_Toc2291)

[6.1 docker镜像常用命令 6](#_Toc12749)

[6.1.1 搜索镜像 6](#_Toc22529)

[6.1.2 下载镜像 6](#_Toc23569)

[6.1.3 列出镜像 6](#_Toc8461)

[6.1.4 删除本地镜像 7](#_Toc32512)

[6.1.5 保存镜像 7](#_Toc15648)

[6.1.6 加载镜像 7](#_Toc7880)

[6.1.7 构建镜像 8](#_Toc31211)

[6.2 docker容器常用命令 8](#_Toc11171)

[6.2.1 新建并启动容器 8](#_Toc17093)

[6.2.2 列出容器 9](#_Toc22381)

[6.2.3 停止容器 9](#_Toc11895)

[6.2.4 强制停止容器 9](#_Toc16577)

[6.2.5 启动已停止的容器 9](#_Toc94)

[6.2.6 重启容器 9](#_Toc26633)

[6.2.7 进入容器 9](#_Toc24496)

[6.2.8 删除容器 10](#_Toc17973)

[6.2.9 导出容器 10](#_Toc24043)

[6.2.10 导入容器 10](#_Toc23018)

[7. 将微服务运行在docker上 10](#_Toc28085)

[7.1 使用Dockerfile构建docker镜像 10](#_Toc26476)

[7.1.1 将项目打成可执行jar(以config-client为例) 10](#_Toc13746)

[7.1.2 将项目上传到linux 11](#_Toc18574)

[7.1.3 构建镜像 11](#_Toc18207)

[7.1.4 启动镜像进行测试 12](#_Toc21868)

[7.1.5 将config-server运行在docker上 12](#_Toc15924)

[8. 管理docker镜像 14](#_Toc29635)

[8.1 使用docker hub管理镜像 14](#_Toc17044)

[8.1.2 注册与登录(https://hub.docker.com ) 14](#_Toc2581)

[8.1.2 创建仓库 14](#_Toc20575)

[8.1.3 推送镜像 15](#_Toc15627)

[8.2 使用私有仓库管理镜像 16](#_Toc13978)

[8.2.1 搭建私有仓库 16](#_Toc20647)

[8.2.2 将镜像推送到私有仓库 17](#_Toc24069)

[9. 使用maven插件构建docker镜像 17](#_Toc5189)

[9.1 添加maven的docker插件(以docker-server为例) 17](#_Toc17320)

[9.2 执行以下命令，构建Docker镜像 17](#_Toc14213)

[9.3 启动镜像 18](#_Toc11405)

[9.4 访问测试 18](#_Toc6026)

[10. 插件读取Dockerfile进行构建 19](#_Toc1155)

[10.1 以config-client为例，main/docker新建Dockerfile 19](#_Toc20281)

[10.2 修改pom.xml 19](#_Toc10217)

[10.3 构建Docker镜像 20](#_Toc27222)

[10.4 推送镜像(使用maven插件) 21](#_Toc6344)

[10.4.1 修改maven的全局配置文件settings.xml 21](#_Toc23223)

[10.4.2 修改pom.xml 21](#_Toc25032)

[10.4.3 推送镜像 21](#_Toc2024)

[10.5 启动并测试 21](#_Toc15812)

# Docker简介

Docker是一个基于GO语言实现的开源的容器引擎，它有助于更快的交互应用。Docker可将应用程序和基础设施层隔离，并且能将基础设施当做程序一样进行管理。使用docker，可更快的打包、测试及部署应用程序，并可缩短从编写到部署运行代码的周期，达到“一次封装，到处运行”的效果

官网： [www.docker.com](http://www.docker.com) / [www.docker-cn.com](http://www.docker-cn.com)

# Docker的架构

|  |
| --- |
|  |

# Centos6.8安装docker

Docker运行在centos-6.5或者更高版本的centos上，要求系统为64位，系统内核版本为2.6.32-641或者更高版本

查看对应的内核信息：uname -r

查看centos的版本信息：lsb\_release -a 或者 cat /etc/redhat-release

## 3.1 安装epel库，docker安装需要依赖于epel库

yum install -y epel-release

## 3.2 安装docker-io

yum install -y docker-io

## 3.3 安装后配置文件地址/etc/config/docker

## 3.4 启动docker

## 3.5 查看docker版本 docker version

# Centos7安装docker

Docker运行在centos-6.5或者更高版本的centos上，要求系统为64位，系统内核版本为2.6.32-641或者更高版本

查看对应的内核信息：uname -r

查看centos的版本信息：lsb\_release -a 或者 cat /etc/redhat-release

以下命令如果使用root登录的话，命令前面不需要加sudo

## 卸载老版本的docker

sudo yum remove docker

若需删除docker存储的文件，例如镜像、容器等信息，需执行如下命令

sudo rm -rf /var/lib/docker

sudo rm -rf /var/lib/docker-engine

sudo rm -rf /var/lib/dockershim

## 安装仓库

### 4.2.1 执行以下命令，安装docker所需的包

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



其中，yum-utils提供了yum-config-manager工具；device-mapper-persistent-data及lvm2则是devicemapper存储驱动所需的包

### 4.2.2 执行以下命令，安装stable仓库

sudo yum-config-manager \ --add-repo \

<https://download.docker.com/linux/centos/docker-ce.repo>

### 4.2.3 执行以下命令，启用edge及test仓库(可选)

sudo yum-config-manager --enable docker-ce-edge #启用edge仓库

sudo yum-config-manager --enable docker-ce-test #启用test仓库

若再次禁用，可加上—disable标签

sudo yum-config-manager --disable docker-ce-edge #禁用edge仓库

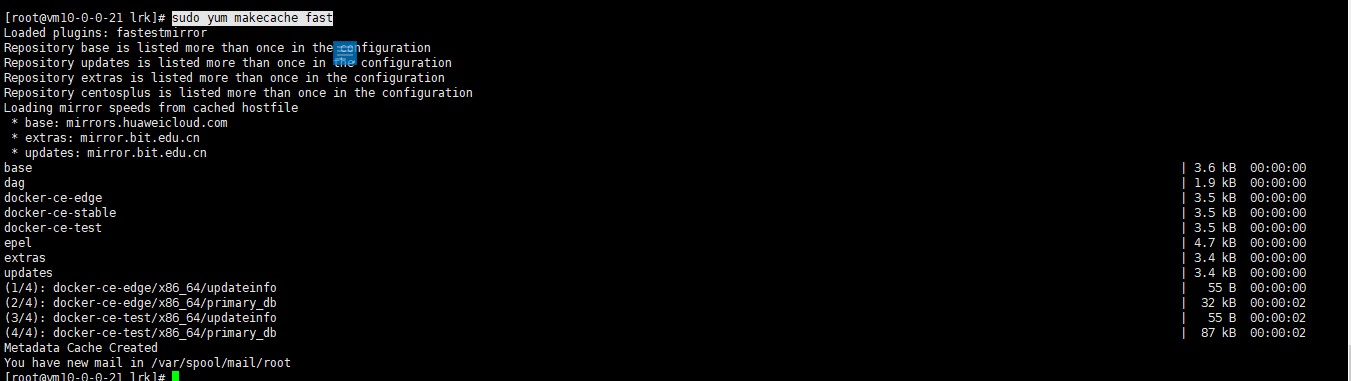
sudo yum-config-manager --disable docker-ce-test #禁用test仓库

备注：edge/test仓库其实也包含在了docker,repo文件中，但默认是禁用的

# 安装docker ce

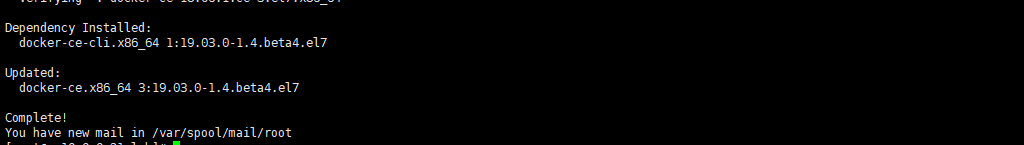
## 5.1 执行以下命令，更新yum的包索引

sudo yum makecache fast



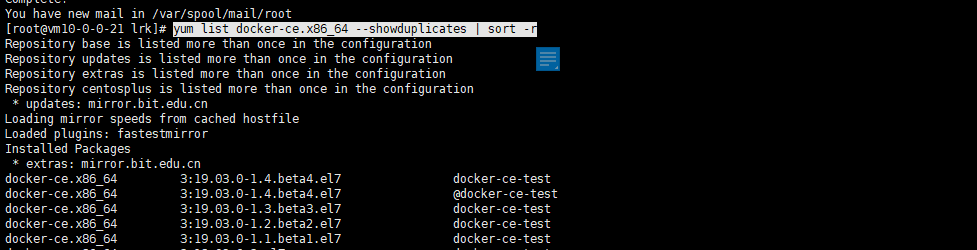
## 5.2 执行以下命令，安装最新版本的docker ce

sudo yum install docker-ce



查看docker当前可用版本

yum list docker-ce.x86\_64 --showduplicates | sort -r



安装指定版本的docker

sudo yum install docker-ce-<version>

## 5.3 启动docker

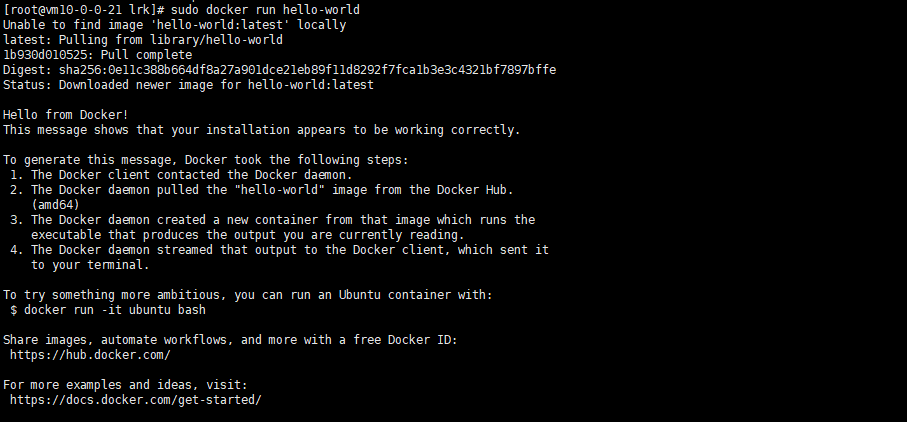
sudo systemctl start docker



## 5.4 验证安装是否正确

sudo docker run hello-world

然后docker将下载测试镜像，并使用该镜像启动一个容器。如果能够看到如下的输出，说明安装成功



run命令会先去本地查找对应的镜像，如果本地没有，则去docker hub(我们这里改成了aliyun)上面找对应的镜像，找到的话就拉取下来并执行，找不到就返回错误not exist

## 5.5 升级docker

执行以下命令

sudo yum makecache fast

然后按照安装docker的步骤，即可完成升级

# 配置镜像加速器

国内访问docker hub的速度很不稳定，有时甚至出现连接不上的情况。

常用的docker加速器有阿里云加速器、DaoCloud加速器等。

以aliyun为例：

访问 <https://dev.aliyun.com/search.html>

注册aliyun账号，账号：liurkali 密码：lrk291988731 绑定邮箱：[17600272806@163.com](mailto:17600272806@163.com)

访问： <https://cr.console.aliyun.com/undefined/instances/mirrors>

|  |
| --- |
|  |

## centos6.8上面配置镜像加速器

1. 复制镜像加速器的地址，修改docker配置文件other\_args参数

vim /etc/sysconfig/docker

|  |
| --- |
|  |

1. 重启docker：service docker restart

3. 检查加速器是否生效：ps -ef | grep docker

|  |
| --- |
|  |

如果在后面可以看到配置的专属镜像加速器地址，说明配置成功

## centos7上面配置镜像加速器

1. 复制镜像加速器的地址，修改docker配置文件other\_args参数

vim /etc/docker/daemon.json

上面的配置文件可能不存在，需要自己手动创建

1. 在daemon.json配置文件里面添加如下内容

|  |
| --- |
| {  "registry-mirrors": ["https://avsl9hr1.mirror.aliyuncs.com"]  } |

1. 重新加载配置文件

systemctl daemon-reload

1. 重启docker

systemctl restart docker

1. 配置是否生效，同centos6.8

ps -ef | grep docker

# Docker常用命令

Docker的基本组成：镜像、容器、仓库

## docker帮助命令

1. 查看版本号：docker version
2. 查看对应的信息：docker info
3. docker帮助命令：docker --help
4. docker stats：查看docker的cpu和内存

## 7.2 docker镜像常用命令

### 7.2.1 概念

镜像是一种轻量级、可执行的独立软件包，用来打包软件运行环境和基于运行环境开发的软件，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

Docker镜像(image)就是一个只读的模板，镜像可以用来创建docker容器，一个镜像可以创建多个容器

#### 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。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

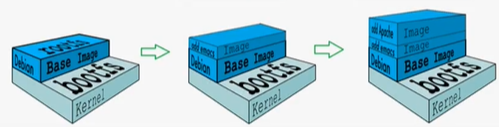
特性：一次同时加载多个文件系统，但从外面看起来，只能看到一个文件系统，联合加载会把各层文件系统叠加起来，这样最终的文件系统会包含所有底层的文件和目录

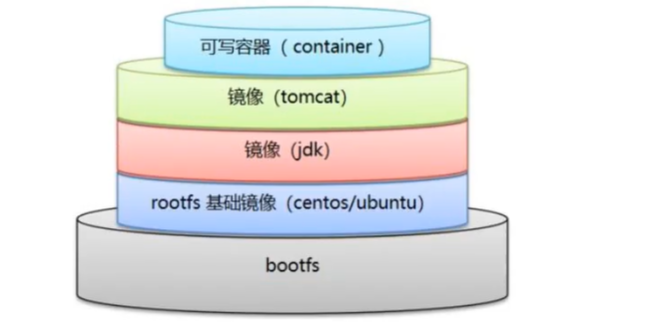
#### 7.2.2.2 docker镜像加载原理

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称为UnionFS。

bootfs(boot file system)主要包含bootloader和kernel。Bootloader主要是引导加载kernel，linux系统刚启动时会加载bootfs文件系统，在docker镜像的最底层是bootfs。这一层与linux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





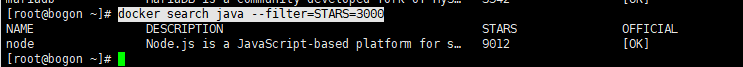
### 7.2.2 搜索镜像

可使用docker search命令搜索存放在docker hub中的镜像

docker search java

搜索收藏数大于3000的：docker search java --filter=STARS=3000



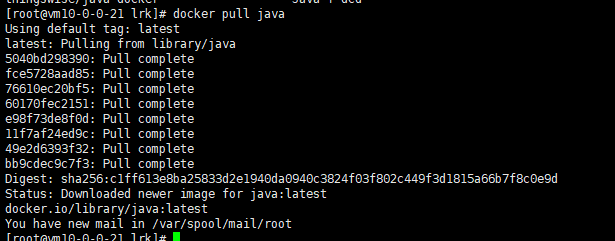


### 7.2.3 下载镜像

使用docker pull 命令即可从docker registry上下载镜像

docker pull java 等价于 docker pull java docker.io/library/java:latest

如果下载慢，就配置加速器，参考5

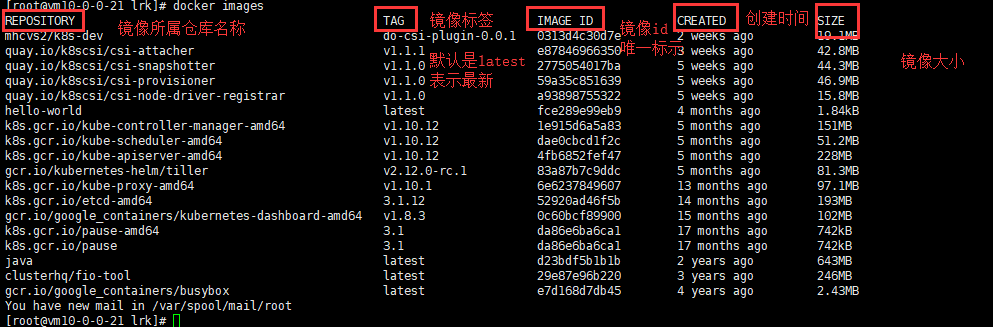


指定想要下载的镜像标签及docker registry地址

docker pull reg.itmuch.com/java:7

### 7.2.4 列出当前主机镜像

docker images



同一个镜像所属的仓库名称可以有多个TAG(镜像标签)，代表着不同的版本。我们可以使用repository:tag来定义不同的镜像。

如果不指定一个镜像的版本标签，默认使用latest镜像

options说明：

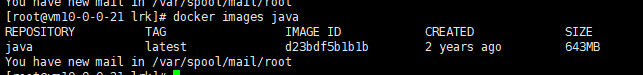
docker images -a：列出本地所有的镜像（含中间印象层）

docker images -q：只显示镜像id

docker images --digests：显示镜像的摘要信息

docker images --no-trunc：显示完整的镜像信息

docker images java



### 7.2.5 删除本地镜像

删除一个镜像

docker rmi (-f) hello-world(镜像名或者imageId都可以)

等价于docker rmi (-f) hello-world:latest(镜像名或者imageId都可以)

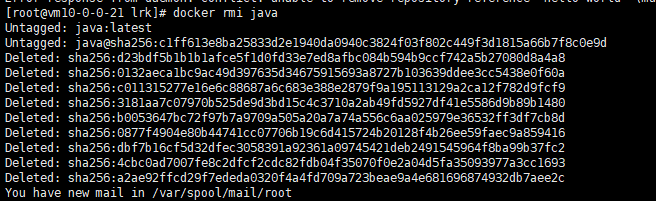
删除多个镜像

docker rmi -f 镜像名1：tag 镜像名2：tag

删除所有镜像

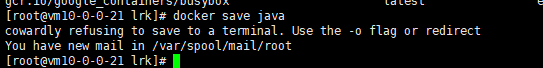
docker rmi -f $(docker images -qa)

-f 强制删除，可以删除正在运行中的镜像



### 7.2.6 保存镜像

docker sava java



### 7.2.7 加载镜像

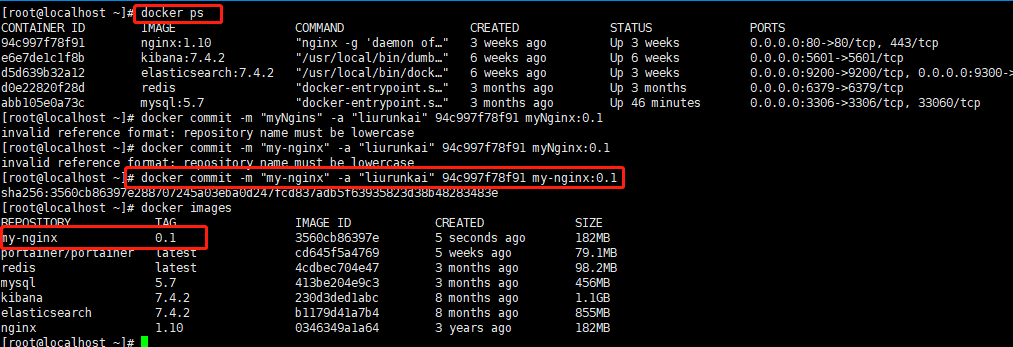
docker load

### 7.2.8 构建镜像

docker build

### 7.2.9 提交镜像

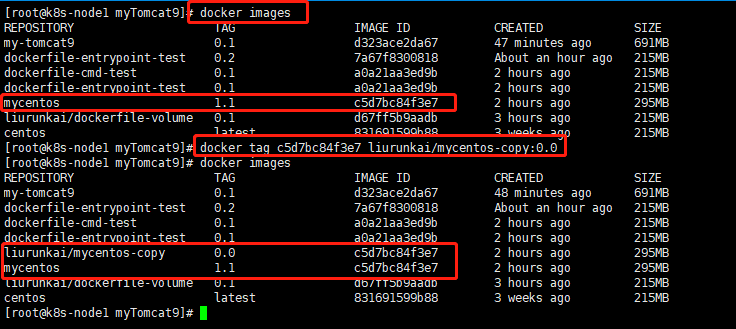
docker commit -m=”提交的描述信息” -a=“作者” 容器id 要创建的目录镜像名：tag



### 7.2.10 给镜像打标签

相当于创建了一个新的镜像名称，但是镜像id不变

docker tag 镜像id 镜像名称:tag



## 7.3 docker容器常用命令

### 7.3.1 概念

Docker利用容器(container)独立运行一个或一组应用，容器是用镜像创建的运行实例。

他可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台

### 7.3.2 新建并启动容器

docker run使用docker可以新建并启动一个容器，使用docker run启动一个容器时，会先检查本地是否存在指定镜像。如果本地不存在该名称的镜像，docker会自动从GitHub上下载镜像并启动一个docker容器docker run [options] image [command] [args...]

Options说明：

--name=”容器新名字”：为容器指定一个名称

-d：后台运行容器，并返回容器id，也即启动守护式容器

-i：以交互模式运行容器，通常与-t(-it)同时使用

-t：为容器重新分配一个伪输入终端，通常与-i(-it)同时使用

-P：随机端口映射

-p：指定端口映射，有4种形式：

ip:hostPort:containerPort

ip::containerPort

hostPort:containerPort

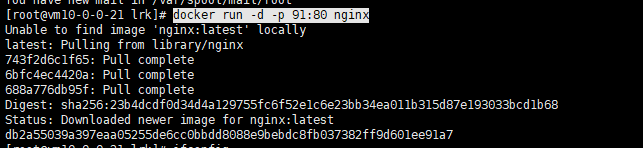
containerPort

-v：指定挂载卷

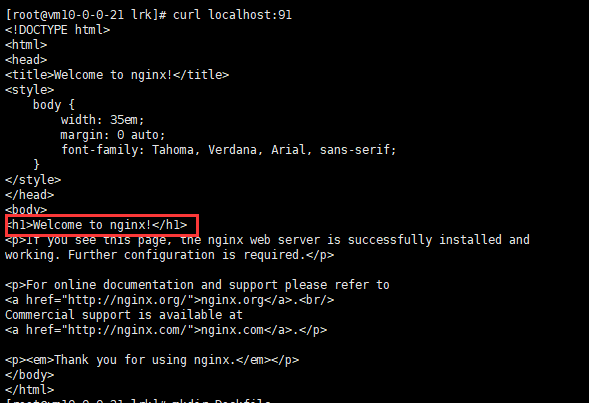
1. docker run java /bin/echo 'Hello World'



1. docker run -d -p 91:80 nginx



访问<http://localhost:91>，可以看到nginx欢迎页面



1. docker run的运行流程
   1. 检测本地是否有镜像，如果有镜像，则执行运行
   2. 如果本地没有镜像，则去仓库下载镜像，如果仓库有镜像，则下载并运行
   3. 如果仓库没有镜像，则直接报错

### 7.3.3 列出容器

列出当前所有正在运行中的容器：docker ps



optios：

-a：列出当前所有正在运行中的和以前运行过的容器：docker ps -a

-l：显示最近创建的容器（最新运行的容器）：docker ps -l

-n：显示最近n个创建的容器（展示最近三次运行的容器）：docker ps -n 3

-q：静默模式，只显示容器编号：docker ps -q

--no-trunc:不截断输出

### 7.3.4 停止容器

docker stop 容器id



### 7.3.5 强制停止容器

docker kill 容器id



### 7.3.6 启动已停止的容器

docker start 容器id



### 7.3.7 重启容器

docker restart 容器id，执行成功之后会返回对应的容器编号



### 7.3.8 进入容器

(1)docker exec -t 容器id /bin/bash：进入容器后开启一个新的终端

(2)docker attach 容器id：进入容器正在执行的终端，不会打开新的终端

在很多场景下，使用docker attach命令并不方便，当多个窗口同时attach到同一个容器时，所有窗口都会同步显示。同理，如果某个窗口发生阻塞，其它窗口也无法执行操作

(3)使用nsenter进入容器

nsenter工具包含在util-linux2.23或更高版本中，为了连接到容器，需要找到容器中的第一个进程PID，可通过以下命令获取

docker inspect --format "{{State.Pid}}" 容器id

获得PID后，就可以使用nsenter命令进入容器了

nsenter –target PID –mount –uts –ipc –net –pid

### 7.3.9 删除已停止的容器

删除指定容器

docker rm 容器id，不能删除正在运行的容器

删除所有容器，-f可以删除正在运行的容器

docker rm -f $(docker ps -a -q)

删除所有容器：通过管道方式

docker ps -a -q|xargs docker rm

### 7.3.10 导出容器

docker export

### 7.3.11 导入容器

docker import

### 7.3.12 退出并停止容器

exit

执行exit之后再执行docker ps命令，可以看到刚刚的容器没了

### 7.3.13 退出但不停止容器

ctrl + p + q

再执行docker ps命令，可以看到刚刚的容器还在

### 7.3.14 查看容器日志

docker logs -f -t --tail 容器id

docker logs -f -t --tail 行数 容器id

### 7.3.15 查看容器内运行的进程

docker top 容器id

### 7.3.16 查看容器内部细节(元数据)

docker inspect 容器id

### 7.3.17 从容器内拷贝文件到主机上

docker cp 容器id:容器内路径 目的主机路径

eg:docker cp 1111:/usr/local/aa.txt /home

## 7.4 仓库

### 7.4.1 概念

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对个仓库，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(public)仓库和私有(private)仓库两种形式

最大的公开仓库是Docker Hub(https://hub.docker.com)

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与、网易云等

## 7.5 docker设置容器开机自启

docker update 容器名称 --restart=always

eg：docker update nginx --restart=always



## 7.6 docker为什么比虚拟机快？

（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guest os

# 容器数据卷

## 概念

主要用来做容器间的数据共享以及容器数据持久化的，如果数据都在容器中，那么容器删除，数据就会丢失。

卷就是目录或文件，存在于一个或多个容器中，由docker挂载到容器，但不属于联合文件系统，因此能够绕过Union File System提供一些用于持续存储或共享数据的特性

卷的设计目的就是数据持久化，完全独立于容器的生存周期，因此docker不会在容器删除时删除其挂载的数据卷

特点：

数据卷可以在容器之间共享或重用数据

卷中的更改可以直接生效

数据卷中的更改不会包含在镜像的更新中

数据卷的生命周期一直持续到没有容器使用它为止

## 容器数据卷（volume）的使用

### 使用命令添加容器数据卷

1. 使用-v命令添加容器数据卷

docker run -it -v 主机目录地址:容器目录地址 镜像名

eg(对里面的内容拥有读写权限)：

docker run -it -v /myDataVolume:/containerDataVolume centos

执行上述命令后会在主机根目录下面生成myDataVolume文件夹



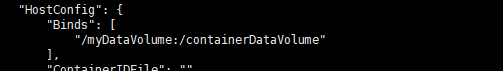
在centos容器里面的根目录下生成containerDataVolume文件夹

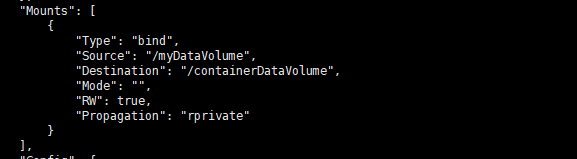


1. 查看容器数据卷是否挂载成功

docker inspect 容器id

返回一个json串信息，里面会有volume属性及binds属性等，可以查看主机的myDataVolume目录和容器containerDataVolume的目录绑定成功，说明容器数据卷挂载成功。并且volumesRW属性为true，拥有读写权限

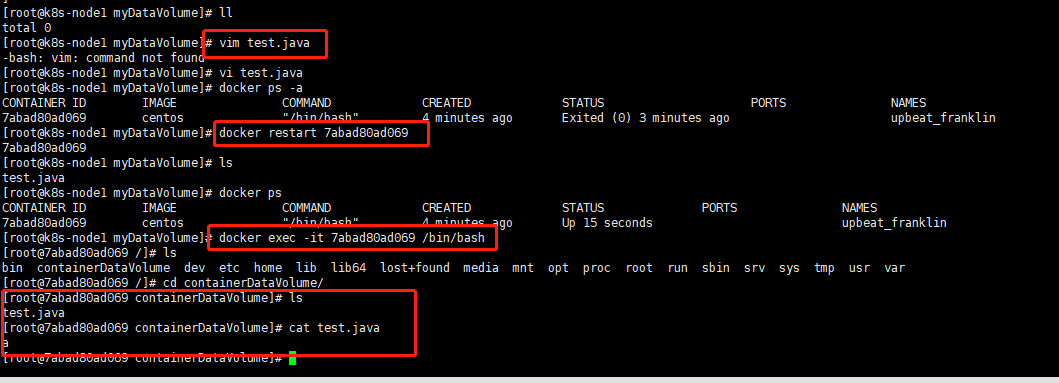




1. 容器和宿主机之间数据共享

当我们在主机上面的myDataVolume目录下面新建一个test.java文件后，我们就可以在容器里面的containerDataVolume目录下面看到对应的test.java文件，同理，当我们修改了容器内test.java文件里面的内容后可以看到虚拟机内test.java里面的文件内容被修改。

当我们将容器停止或者删除后，只要镜像在，虚拟机内的数据不会丢失



1. 容器停止退出(exit)或者删除容器后，主机修改后数据是否同步(是)

当在容器目录下面执行exit命令后，容器关闭并且退出，我们修改主机的myDataVolume的目录下的host.txt文件并且新建host02.txt文件。

当我们在开启容器时docker start 容器id并且进入容器docker attach 容器id后，仍然可以看到新建的host02.txt文件以及被修改过的host.txt文件内容



1. 命令(带权限ro：只读，rw：可读可写)

docker run -it -v 主机目录地址:镜像目录地址 镜像名

eg(容器里面拥有读权限，没有写权限，ro表示readonly)：

docker run -it -v /myDataVolume:/containerDataVolume:ro centos

即当在主机里面的myDataVolume下新建host.txt文件并修改host.txt文件后，容器里面的containerDataVolume目录下可以看到host.txt文件及看到里面修改的内容，但是不能在容器里面的containerDataVolume目录下新建或者修改文件

1. 备注：如果出现错误：cannot open directory.:Permission denied

解决办法：在挂载目录后多加一个--privileged=true参数

docker run -it -v /myDataVolume:/containerDataVolume --privileged=true centos

### 具名挂载和匿名挂载

1. 匿名挂载：不指定虚拟机目录，指定容器目录

docker run -it -v /niMingVolume centos

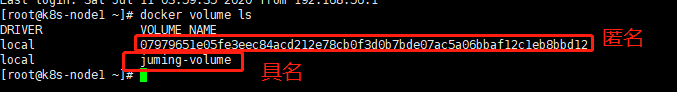
查看所有的volume：docker volume ls



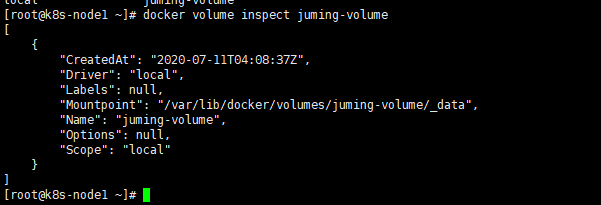
这种就是匿名挂载，-v只指定了容器内的路径，没有写容器外的路径

1. 具名挂载：指定虚拟机名称(注意：不是目录)，指定容器目录

docker run --name centos02 -it -v juming-volume:/juMingVolume centos



查看具名文件的位置：docker volume inspect juming-volume



1. 如何区分是具名挂载、匿名挂载、指定路径挂载？

匿名挂载：-v 容器内路径

具名挂载：-v 卷名:/容器内路径

指定路径挂载：-v /虚拟机路径:/容器内路径

### 使用dockerFile添加容器数据卷

1. 用dockerFile添加容器数据卷
   1. 在主机的根目录下新建mydocker文件夹并进入 mkdir /mydocker
   2. 创建dockerFile(vi volumeTestDockerFile)并添加如下内容：

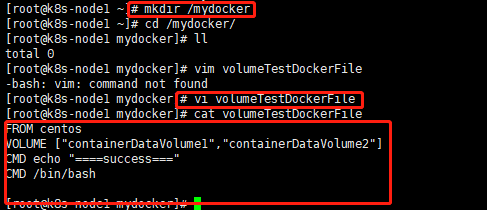
# volume test 容器数据卷测试

FROM centos # 来自于centos这个镜像

VOLUME [”/containerDataVolume1”,”/containerDataVolume2”]#新建两个volume

CMD echo “finished,------success” # 输出finished,------success

CMD /bin/bash #



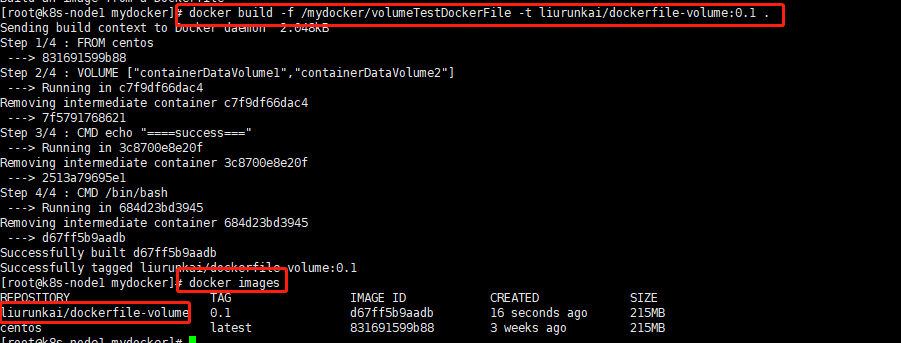
* 1. docker build 命令将上面的volumeTestDockerFile生成一个新的镜像

docker build -f /mydocker/volumeTestDockerFile -t xxx/xxx/xxx(镜像名称) .

其中-f：file指明dockerFile的位置

执行成功后会返回Successfully built + 镜像id

使用docker images命令可以看到们对应的镜像xxx/xxx/xxx



* 1. 运行

docker run -it xxx/xxx/xxx /bin/bash



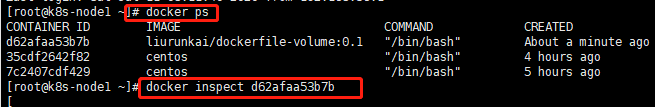
* 1. ll命令可以看到数据容器卷：containerDataVolume1和containerDataVolume2



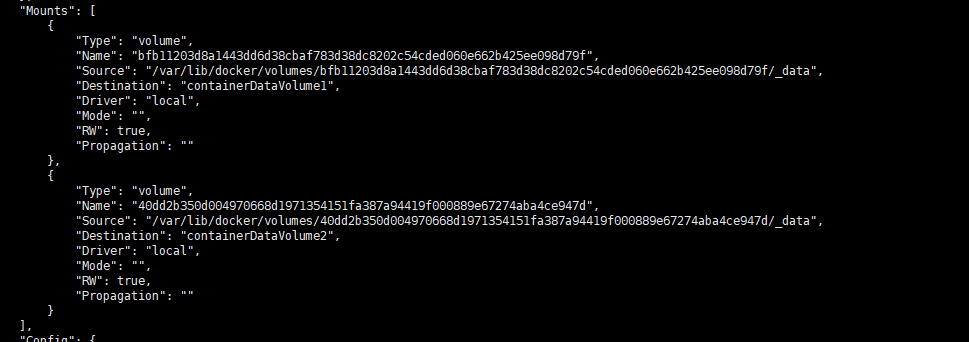
* 1. 查看对应的主机目录，最好打开另一个主机窗口查看

查看正在运行的容器：docker ps

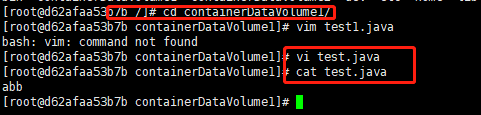
查看容器细节：docker inspect 容器id

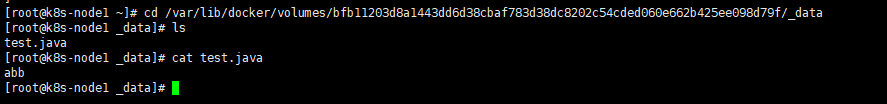


返回的json信息里面volumes可以查看容器内容器数据卷的地址对应的主机内的容器数据卷地址



在容器内containerDataVolume1下创建test.java文件，可以在虚拟机内看到test.java



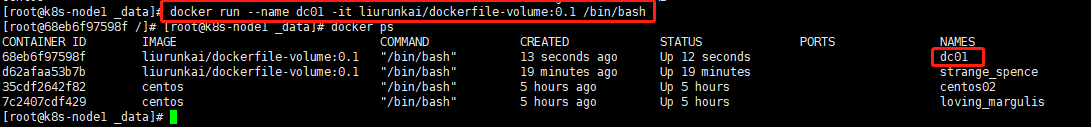


## 容器间传递共享(--volumes-from)

需求：dc02和dc03的数据来自于dc01

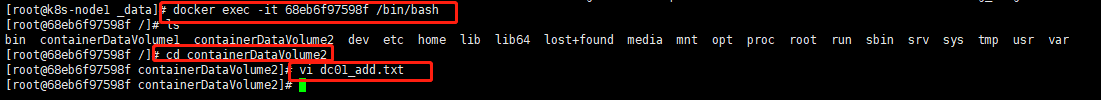
1. 以上一步新建的镜像xxx/xxx/xxx为模板并运行容器dc01、dc02、dc03
2. 启动一个父容器dc01并在containerDataVolume2下面创建文件dc01\_add.txt并修改

docker run -it --name dc01 xxx/xxx/xxx



cd /containerDataVolume2

vi dc01\_add.txt



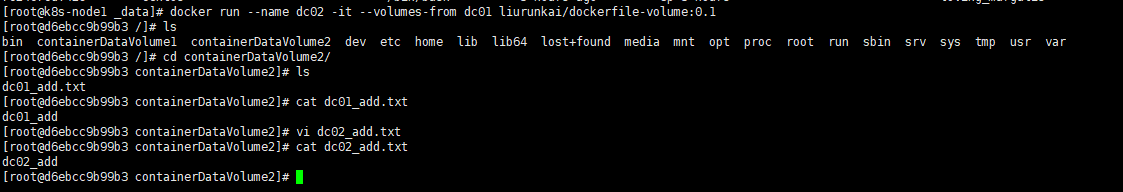
1. dc02继承自dc01

命令：docker run -it --name dc02 --volumes-from dc01 xxx/xxx/xxx

cd /containerDataVolume2

ll 可以看到dc01新建的dc01\_add.txt

同理dc02在containerDataVolume2下面创建文件vi dc02\_add.txt



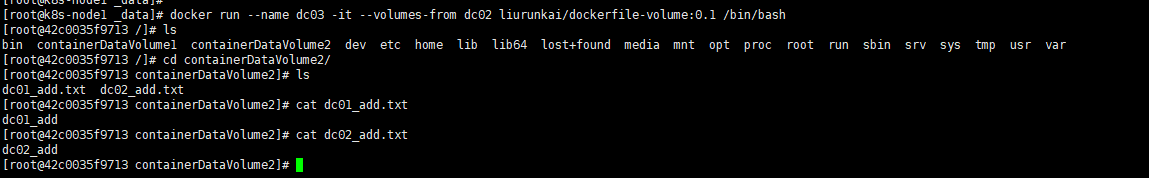
1. dc03继承自dc01

命令：docker run -it --name dc03 --volumes-from dc01 xxx/xxx/xxx

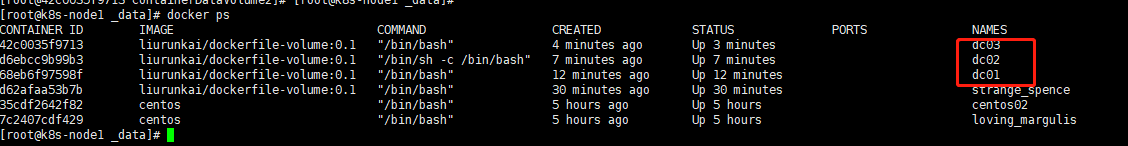
cd /containerDataVolume2

ll 可以看到dc01新建的dc01\_add.txt和dc02新建的dc02\_add.txt

同理dc03在containerDataVolume2下面创建文件vi dc03\_add.txt



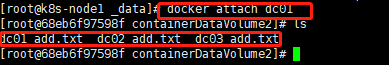
1. docker ps 可以看到现在启动了3个容器dc01、dc02、dc03



1. 进入容器dc01下的containerDataVolume2：docker attach dc01

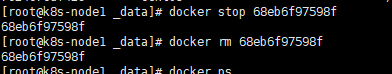
可以看到dc02\_add.txt和dc03\_add.txt文件

说明：容器继承之后，各个容器之间的容器卷即可以达到继承又可以共享数据

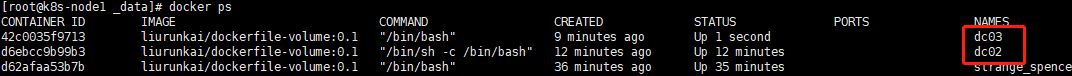


1. 停止并删除dc01容器后，dc02和dc03之间是否可以共享和继承

删除dc01：docker rm -rf dc01

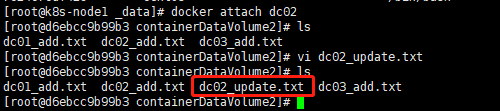


docker ps 可以看到只用dc02和dc03两个容器运行

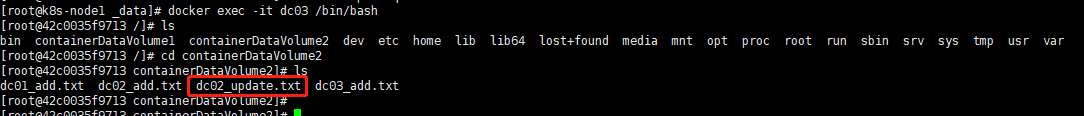


进入dc02的containerDataVolume2目录：docker attach dc02，可以看到 dc01\_add.txt、dc02\_add.txt和dc03\_add.txt文件。

修改dc02的containerDataVolume2目录下的文件：vi dc02\_update.txt，可以执行



1. 进入dc02的containerDataVolume2目录：docker attach dc02，可以看到 dc01\_add.txt、dc02\_add.txt、dc03\_add.txt和dc02\_update.txt文件。



说明：容器之间配置信息的传递，容器数据卷的生命周期一直持续到没有容器使用它为止

使用场景：多个mysql、redis之间数据和配置文件同步

# Dockerfile解析

## 概念

Dockerfile是用来构建Docker镜像的构建文件，是由一系列命令和参数构成的脚本

编写Dockerfile ---> docker build ---> docker run ---> docker push

## DockerFile说明

1. 每条保留字指令都必须为大写字母且后面要跟随至少一个参数
2. 指令按照从上到下顺序执行
3. #表示注释
4. 每条指令都会创建一个新的镜像层，并对镜像进行提交

## 9.3 DockerFile保留字指令(体系结构)

### 9.3.1 FROM指令

使用FROM指令指定基础镜像，当前镜像是基于哪个镜像的。FROM指令必须指定且必须写在其它指令之前。

支持3种格式：

FROM <image>

FROM <image>:<tag>

FROM <image>@<digest>

### 9.3.2 MAINTAINER指令

用于指定维护者的信息(姓名和邮箱地址)，用于为DockerFile署名

格式为：MAINTAINER <name> eg: MAINTAINER [liurk<17600272806@163.com>](mailto:liurk<17600272806@163.com>)

### 9.3.3 RUN指令

容器构建时需要运行的命令

支持2种格式：

RUN <command>

RUN<”executable”,”param1”,”param2”>

### 9.3.4 EXPOSE指令

用于声明在运行时容器提供服务的端口。这只是一个声明，运行时并不会因为该声明就打开相应到端口。该指令作用主要是帮助镜像使用者理解该镜像服务的守护端口；其次是当运行时使用随机映射时，会自动映射EXPOSE的端口

格式为：EXPOSE <port> [<port>...]

eg：

# 声明暴露一个端口示例

EXPOSE port1

# 相应的运行容器使用的命令

docker run -p port1 image / docker run -P image

# 声明暴露多个端口示例

EXPOSE port1 port2 port3

# 相应的运行容器使用的命令

docker run -p port1 -p port2 -p port3 image

# 也可以指定需要映射到宿主机器上的端口号

docker run -p host\_port1:port1 -p host\_port2:port2 -p host\_port3:port3 image

### 9.3.5 WORKDIR指令

指定在创建容器后，终端默认登录进来的工作目录

格式为：WORKDIR /xxx/xxx/xxx

### 9.3.6 ENV指令

用于设置环境变量

格式为：ENV <key> <value>

ENV MY\_PATH /usr/local

WORKDIR $MY\_PATH

### 9.3.7 ADD指令

复制文件+解压文件：将宿主机目录下的文件拷贝进镜像且ADD命令会自动处理url和解压tar压缩包

格式为：ADD <src> <dest> 或者 ADD [“src”,”dest”] 从src目录复制文件到容器的dest。其中src可以是DockerFile所在目录的相对路径、url、压缩包

### 9.3.8 COPY指令

类似ADD,拷贝文件和目录到镜像中

将从构建上下文目录中<源路径>的文件/目录复制到新的一层的镜像内的<目标路径>位置

格式为：

COPY <src> <dest> / COPY [“src”,”dest”]

### 9.3.9 VOLUME指令

容器数据卷，用于数据保存和持久化

格式为：VOLUME [“/data”,”/data2”]

### 9.3.10 CMD指令

指定一个容器启动时要运行的命令。每个DockerFile只有一个CMD命令，如果指定了多个CMD命令，那么只有最后一条会被执行，如果启动容器时指定了运行的命令，则会覆盖CMD指定的命令

格式为：

shell格式：CMD command param1 param2

参数列表格式(为ENTRYPOINT指令提供预设参数)：CMD [“param1”,”param2”]

exec格式：CMD [“executable”,”param1”,”param2”]

### 9.3.11 ENTRYPOINT指令

指定一个容器启动时要运行的命令

ENTRYPOINT和CMD命令的目的一样，都是在指定容器启动程序及参数，如果指定了多个ENTRYPOINE命令,则会追加

### 9.3.12 ONBUILD指令

当构建一个被继承的Dockerfile时运行命令，父镜像在被子继承后父镜像的onbuild被触发

### 9.3.13 ARG指令

设置构建参数，类似于ENV,和ENV不同的是，ARG设置的是构建时的环境变量，在容器运行时是不会存在这些变量的

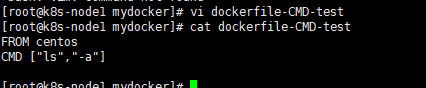
### 9.3.14 USER指令

该指令用于设置启动镜像时的用户或者UID

## 9.4 CMD和ENTRYPOINT命令的区别

（1）CMD：指定一个容器启动时要运行的命令。每个DockerFile只有一个CMD命令，如果指定了多个CMD命令，那么只有最后一条会被执行，如果启动容器时指定了运行的命令，则会覆盖CMD指定的命令

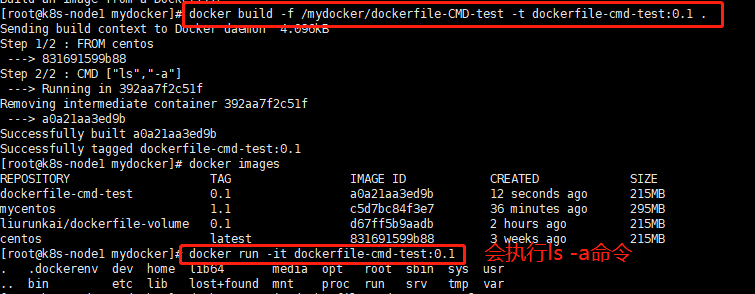
编写dockerfile-CMD-test文件



构建并启动镜像

docker build -f /mydocker/dockerfile-CMD-test -t dockerfile-cmd-test:0.1 .

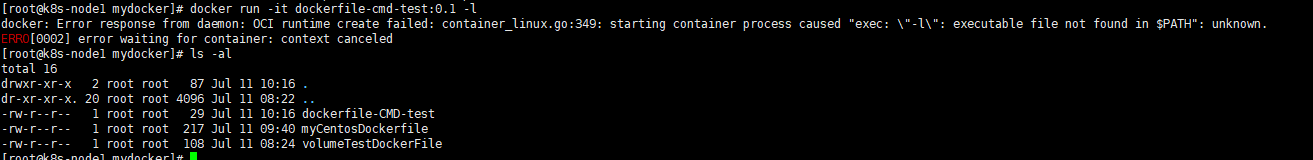
docker run -it dockerfile-cmd-test:0.1



如果追加命令-l：docker run -it dockerfile-cmd-test:0.1 -l 则会报错

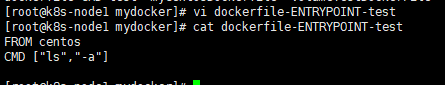


因为-l 不是一条命令，只有ls -al才是命令，CMD命令追加时会覆盖，-l 覆盖了命令ls -a



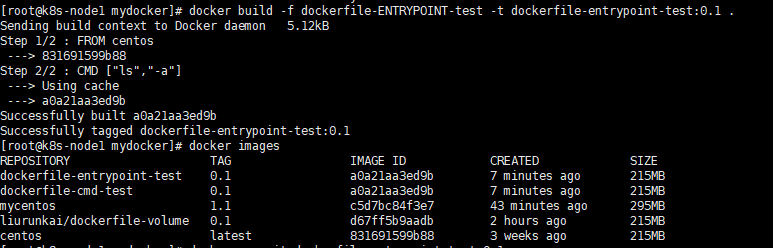
1. ENTRYPOINT:指定一个容器启动时要运行的命令。ENTRYPOINT和CMD命令的目的一样，都是在指定容器启动程序及参数，如果指定了多个ENTRYPOINE命令,则会追加

编写dockerfile-ENTRYPOINT-test文件



构建并启动镜像

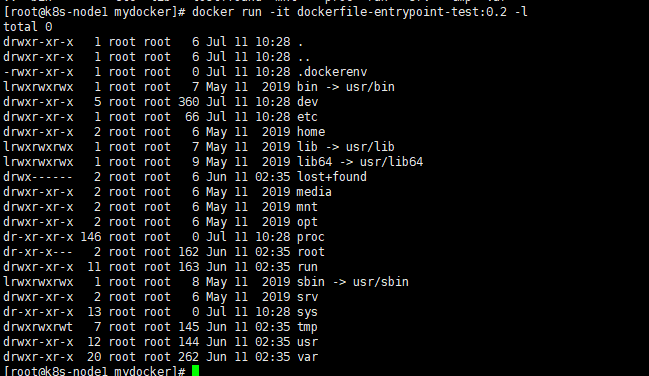
docker build -f dockerfile-ENTRYPOINT-test -t dockerfile-entrypoint-test:0.1 .



docker run -it dockerfile-entrypoint-test:0.1



如果后面追加命令-l，则不会报错



## 9.5 Dockerfile案例

### 9.5.1 自定义镜像myCentos

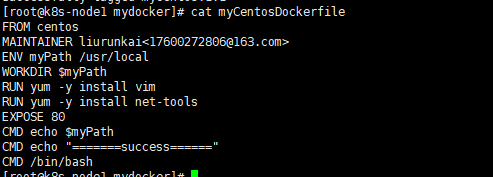
1. 需求

从docker hub上拉取的centos镜像不支持vim和ifconfig命令，且启动容器后默认是根目录。

现在自定义mycentos镜像需要支持vim和ifconfig命令并修改启动容器后的根目录

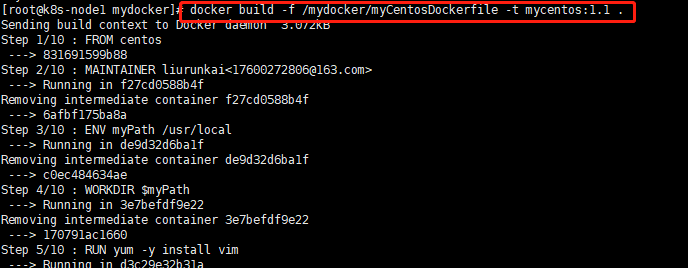
1. 编写Dockerfile: mkdir /myDockerfile vim myCentosDockerfile

|  |
| --- |
| FROM centos # 指定基础镜像为从docker hub上面拉取的centos：docker pull centos  MAINTAINER [liurk<17600272806@163.com>](mailto:liurk<17600272806@163.com>) # 指定维护者的信息  ENV myPath /usr/local # 设置环境变量  WORKDIR $myPath # 指定在创建容器后，终端默认登录进来的工作目录  RUN yum -y install vim # 指定容器构建时需要执行的命令：支持vim命令  RUN yum -y install net-tools#指定容器构建时需要执行的命令：支持ifconfig命令  EXPOSE 80 # 声明暴露的端口  CMD echo $myPath # 指定容器启动时要执行的命令：输出myPath路径  CMD echo “success”  CMD /bin/bash |

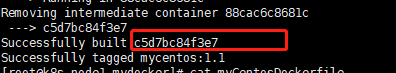


1. build构建镜像

docker build -f /myDockerfile/myCentosDockerfile -t myCentos:1.1 .

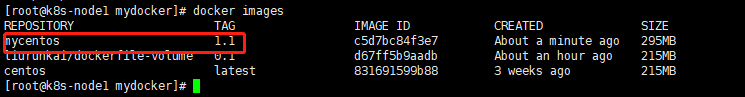


直到构建成功：Successfully built 镜像id



1. 查看对应的镜像

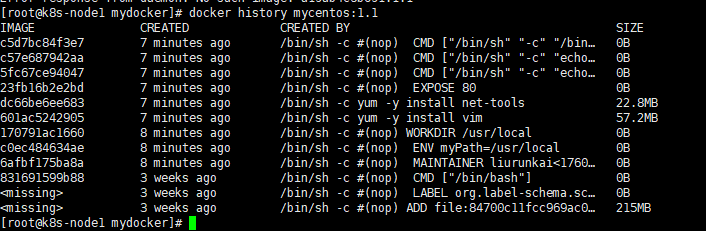
docker images 可以看到我们刚刚构建的镜像myCentos



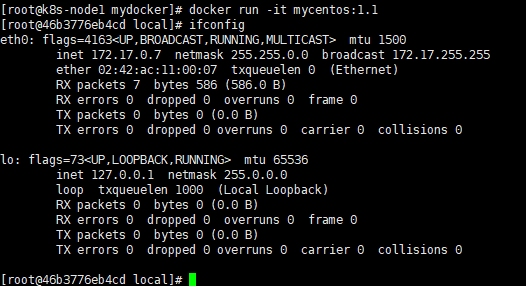
1. 运行镜像

docker run -it myCentos:1.1

1. 查看镜像的构建过程/历史：docker history 镜像id



1. 测试
2. 容器启动后终端默认的根路径/usr/local
3. 测试vim命令和ifconfig命令

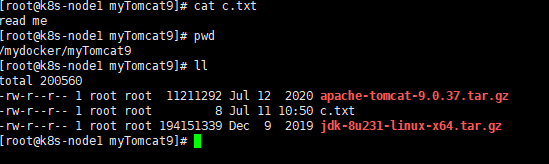


### 9.5.2 自定义镜像myTomcat9

1. 编写Dockerfile：mkdir -p /myDockerfile/tomcat

vi myTomcat9Dockerfile(存放镜像) touch c.txt(演示COPY指令)

拷贝两个压缩包(jdk和tomcat的.tar.gz包)文件(演示ADD指令)



FROM centos # 指定基础镜像为centos，需要从docker hub上面拉取centos镜像

MAINTAINER [liurk<17600272806@163.com>](mailto:liurk<17600272806@163.com>) # 指定维护者的信息

COPY c.txt /usr/local/container.txt #把主机上的c.txt拷贝到容器/usr/local路径下

ADD jdk.tar.gz /usr/local/ # 把jdk.tar.gz拷贝到容器/usr/local路径下

ADD tomcat.tar.gz /usr/local/ # 把tomcat.tar.gz拷贝到容器/usr/local路径下

RUN yum -y install vim # 安装vim编辑器

ENV myPath /usr/local # 设置环境变量

WORKDIR $myPath # 设置登录后的位置

ENV JAVA\_HOME /usr/lcoal/jdk1.8.0\_171 # 配置jdk环境变量

ENV CLASSPATH $JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar # 配置jdk环境变量

ENV CATALINA\_HOM E /usr/local/apache-tomcat-9.0.8 #配置tomcat环境变量

ENV CATALINA\_BASE /usr/local/apache-tomcat-9.0.8 #配置tomcat环境变量

ENV PATH $PATH:$JAVA\_HOME/bin:$CATALINA\_HOME/lib:$CATALINA\_HOME/bin #配置tomcat

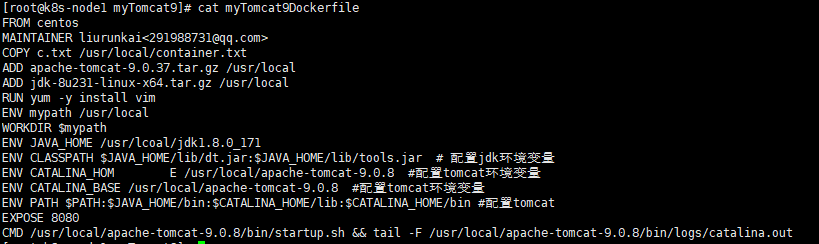
EXPOSE 8080 # 容器运行时监听的端口

# ENTRIPOINT [“/usr/local/apache-tomcat-9.0.8/bin/startup.sh”]

# CMD [“/usr/local/apache-tomcat-9.0.8/bin/startup.sh”,”run”]

CMD /usr/local/apache-tomcat-9.0.8/bin/startup.sh \

&& tail -F /usr/local/apache-tomcat-9.0.8/bin/logs/catalina.out

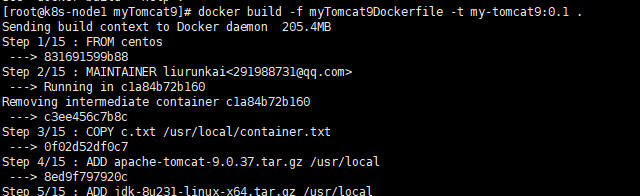


1. 构建myTomcat9Dockerfile

docker build -f myTomcat9Dockerfile -t myTomcat9 .

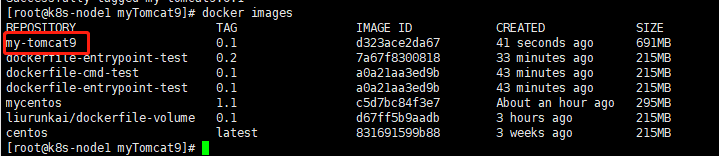
如果在当前目录下执行不需要加-f参数，如果文件名就叫Dockerfile的话，它会自动读取Dockerfile文件，可以简写成：docker build -t myTomcat9 .

知道构建成功：successfully built 镜像id





1. 查看镜像 docker images 可以看到myTomcat9



1. 执行

docker run -d -p 9080:8080 --name myTomcat9 \

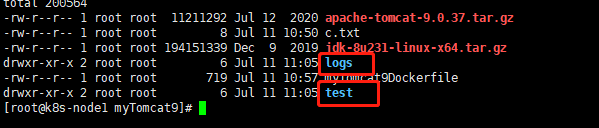
-v /myDockerfile/tomcat/test:/usr/local/apache-tomcat-9.0.8/webapps/test \

-v /myDockerfile/tomcat/tomcat9logs:/usr/local/apache-tomcat-9.0.8/logs \

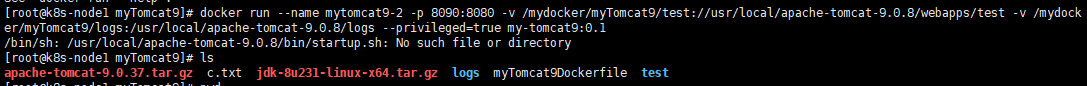
--privileged=true myTomcat9



1. 虚拟机内可以看到容器里面挂载出来的test和logs



1. 进入容器并查看docker exec -it 容器id /bin/bash



1. 验证localhost:9080 可以看到tomcat页面说明启动成功

# 将springboot微服务运行在docker上

## 10.1 使用Dockerfile构建docker镜像

Dockerfile是一个文本文件，其中包含了若干条指令，指令描述了构建镜像的细节，并支持以#开头的注释行

### 将项目打成可执行jar(以log4j2为例)

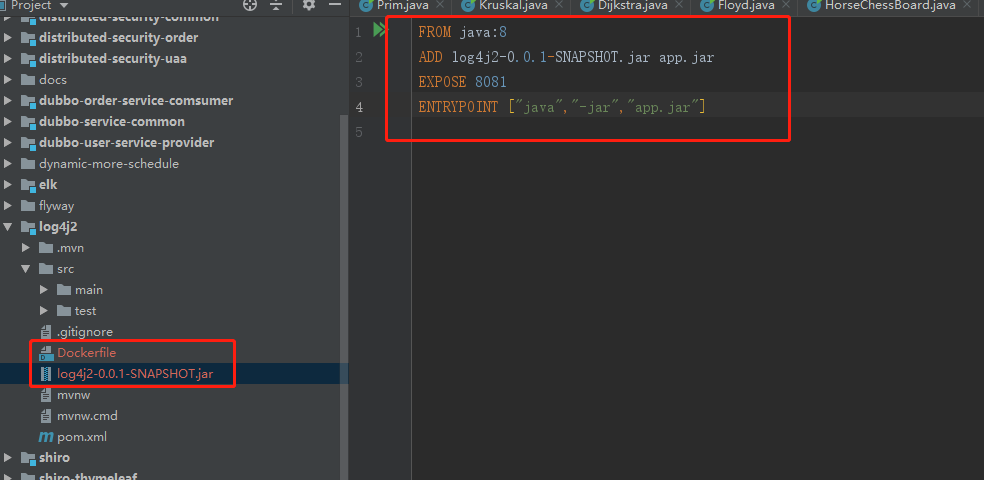
进入到log4j2目录下，执行mvn clean install -Dmaven.test.skip=true

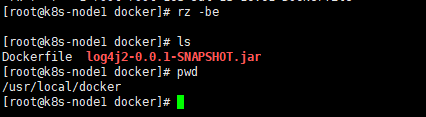
执行成功后，在target目录下可以看到jar

|  |
| --- |
|  |

### 编写Dockerfile并将jar和Dockerfile上传到服务器

|  |
| --- |
| FROM java:8 ADD log4j2-0.0.1-SNAPSHOT.jar app.jar EXPOSE 8081 ENTRYPOINT *[*"java","-jar","app.jar"*]* |





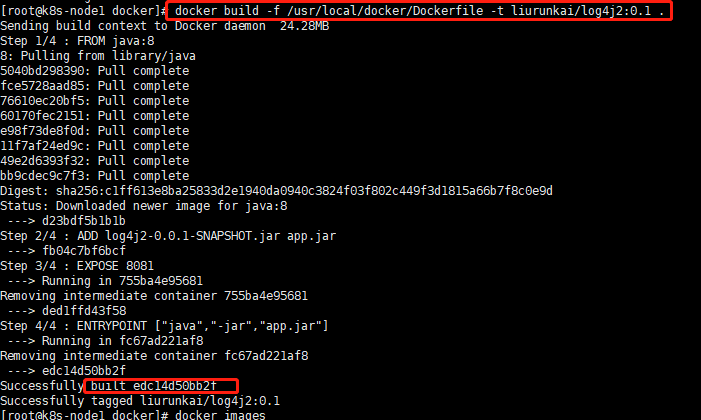
### 10.1.3 构建镜像

使用docker build命令构建镜像

格式：docker build -f Dockerfile地址 -t 仓库名称/镜像名称(:标签)

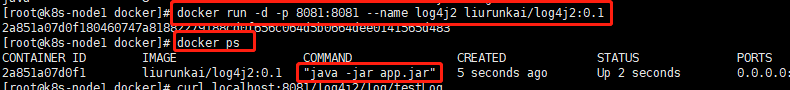
-t 选项指定了镜像的标签

docker build -f /usr/local/docker/Dockerfile -t liurunkai/log4j2:0.1 .

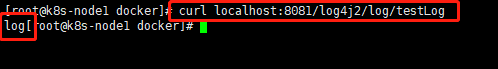


### 10.1.4 启动镜像进行测试

docker]# docker run -d -p 8081:8081 --name log4j2 liurunkai/log4j2:0.1



访问curl localhost:8081/log4j2/log/testLog



### 10.1.5 将config-server运行在docker上

（1）打包并上传

|  |
| --- |
|  |
|  |

（2）构建镜像

|  |
| --- |
|  |
| # 基于哪个镜像  FROM java:8  # 将本地文件挂载到当前容器  VOLUME /tmp  # 复制文件到容器，也可以直接写成ADD config-server-0.0.1-SNAPSHOT.jar /app.jar  ADD config-server-0.0.1-SNAPSHOT.jar app.jar  RUN bash -c 'touch /app.jar'  # 声明需要暴露的端口  EXPOSE 8762  # 配置容器启动后执行的命令  ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"] |
|  |

1. 启动镜像

|  |
| --- |
|  |

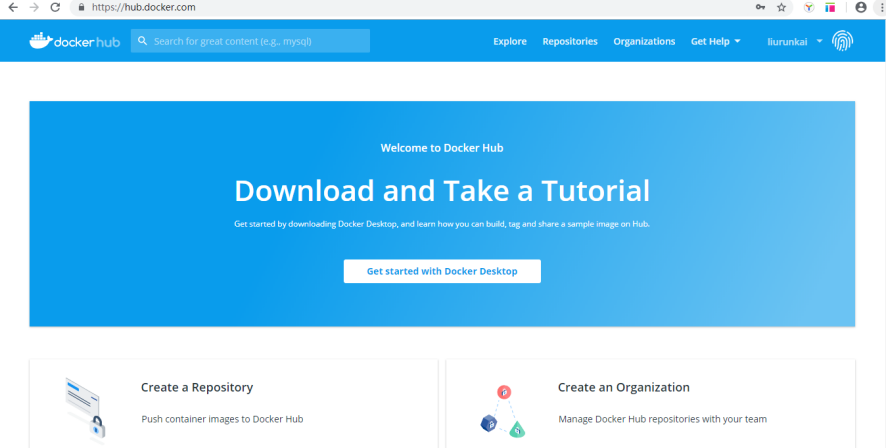
1. 测试

# 管理docker镜像

## 11.1 使用docker hub管理镜像

### 11.1.2 注册与登录(<https://hub.docker.com> )

账号：liurunkai 密码：lrk291988731



使用命令docker login在虚拟机进行登录，输入账号、密码

docker logout 退出登录

|  |
| --- |
|  |

### 11.1.2 创建仓库（这一步可以不用，直接跳到11.1.3推送镜像）

|  |  |
| --- | --- |
|  |  |

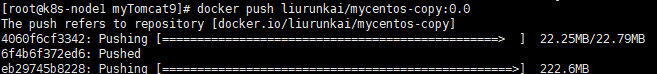
点击create进行创建

|  |
| --- |
|  |

### 11.1.3 推送镜像

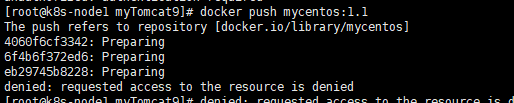
docker push REPOSITORY:TAG

这里可能推送失败，因为dockerhub是国外的网站



推送镜像时，REPOSITORY必须是xxx/xxx 例如liurunkai/mycentos-copy，否则可能会报错：

denied: requested access to the resource is denied(拒绝：拒绝对资源的请求访问)



|  |
| --- |
|  |

docker push liurunkai/config-client:0.0.1

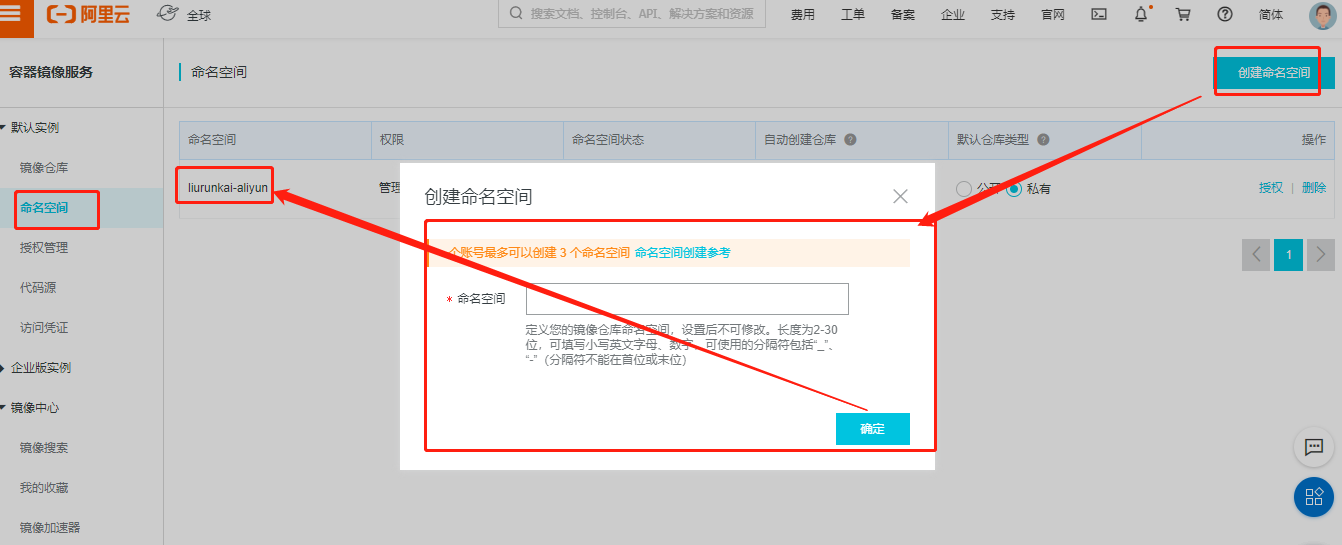
|  |
| --- |
|  |

查看已推送的镜像

|  |
| --- |
|  |
|  |

### 11.1.4 发布镜像到阿里云容器

（1）登录阿里云，搜索容器镜像服务，创建命名空间(隔离)，一个账号只能创建3个命名空间



（2）创建镜像仓库

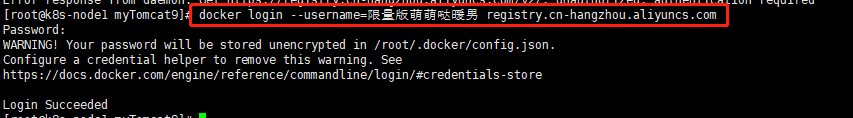
 -------> 

1. 点击镜像仓库查看详情以及操作命令



1. 登录阿里云，密码：lrk291988731

docker login --username=限量版萌萌哒暖男 registry.cn-hangzhou.aliyuncs.com



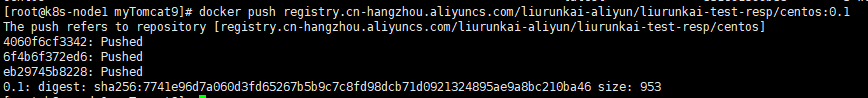
1. 给镜像打标签

docker tag [ImageId] registry.cn-hangzhou.aliyuncs.com/liurunkai-aliyun/liurunkai-test-resp:[镜像版本号]



1. 推送镜像

docker push registry.cn-hangzhou.aliyuncs.com/liurunkai-aliyun/liurunkai-test-resp:[镜像版本号]

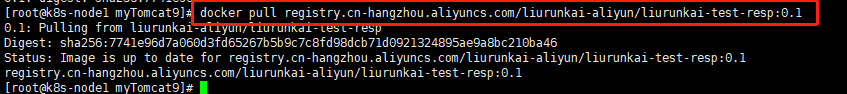


推送成功后在镜像版本查看

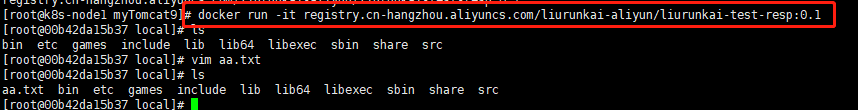


1. 拉取镜像

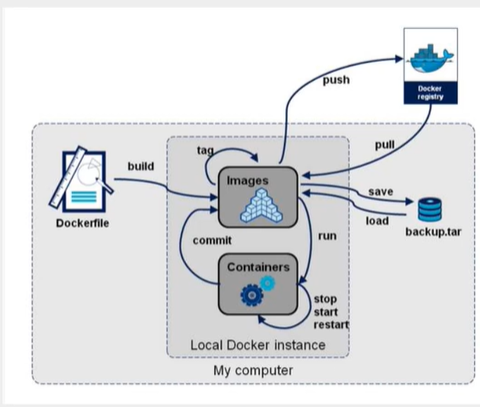
docker pull registry.cn-hangzhou.aliyuncs.com/liurunkai-aliyun/liurunkai-test-resp:[镜像版本号]



1. 启动镜像



### 11.1.5 docker流程



## 11.2使用私有仓库管理镜像

### 11.2.1搭建私有仓库

docker run -d -p 5000:5000 --restart=always --name registry2 registry:2

|  |
| --- |
|  |

### 11.2.2 将镜像推送到私有仓库

# 12. 使用maven插件构建docker镜像

## 12.1 添加maven的docker插件(以docker-server为例)

|  |
| --- |
|  |

## 12.2 执行以下命令，构建Docker镜像

mvn clean package docker:build

会报错：connected refused localhost:2375

解决方式：（1）传到服务器上去执行

（2）在本地安装docker for windows

右键docker小图标，点击settings

|  |
| --- |
|  |

重新执行上述命令，会看到如下输出，说明成功

|  |
| --- |
|  |

查看镜像：docker images

|  |
| --- |
|  |

## 12.3 启动镜像

docker run -d -p 8080:8080 liurunkai/config-server:0.0.1

|  |
| --- |
|  |

## 12.4 访问测试

<http://localhost:8080/application-test.yml>

|  |
| --- |
|  |

可以成功访问，说明启动成功

# 13.插件读取Dockerfile进行构建

## 13.1 以config-client为例，main/docker新建Dockerfile

|  |
| --- |
| **FROM** java:8 **VOLUME /**tmp **ADD** config-client-0.0.1**-**SNAPSHOT.jar app.jar **RUN** bash **-**c **'touch /app.jar' EXPOSE** 9000 **ENTRYPOINT** [**"java"**,**"-Djava.security.egd=file:dev/./urandom"**,**"-jar"**,**"/app.jar"**] |
|  |

## 13.2 修改pom.xml

|  |
| --- |
| <**plugin**>  <**groupId**>com.spotify</**groupId**>  <**artifactId**>docker-maven-plugin</**artifactId**>  <**version**>0.4.12</**version**>  <**configuration**>  <**imageName**>liurunkai/config-client:0.0.1</**imageName**>  <**dockerDirectory**>${project.basedir}/src/main/docker</**dockerDirectory**>  <**resources**>  <**resource**>  <**targetPath**>/</**targetPath**>  <**directory**>${project.build.directory}</**directory**>  <**include**>${project.build.finalName}.jar</**include**>  </**resource**>  </**resources**>  </**configuration**> </**plugin**> |

这个pom.xml没有指定baseImage和entryPoint，而是使用dockerDirectory指定Dockerfile所在的路径，这样就可以使用Dockerfile构建Docker镜像了

## 13.3 构建Docker镜像

方式一：跟之前一样，执行如下命令

docker clean package docker:build -X test

|  |
| --- |
|  |

方式二：将插件绑定在某个phase执行

修改pom.xml: 添加如下配置

|  |
| --- |
| <**executions**>  <**execution**>  <**id**>build-image</**id**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>build</**goal**>  </**goals**>  </**execution**> </**executions**> |
|  |

只需执行mvn package，就会自动执行mvn package docker:build

|  |
| --- |
|  |

## 13.4 推送镜像(使用maven插件)

### 13.4.1 修改maven的全局配置文件settings.xml

|  |
| --- |
| <server>  <id>docker-hub</id>  <username>liurunkai</username>  <password>lrk291988731</password>  <configuration>  <email>17600272806@163.com</email>  </configuration>  </server> |

### 13.4.2修改pom.xml

|  |
| --- |
| <**serverId**>docker-hub</**serverId**> |
|  |

### 13.4.3推送镜像

mvn clean package docker:build -DpushImage

其中pushImage表示推送镜像

## 13.5 启动并测试

docker run -d -p 8081:8081 liurunkai/config-client:0.0.1

|  |
| --- |
|  |

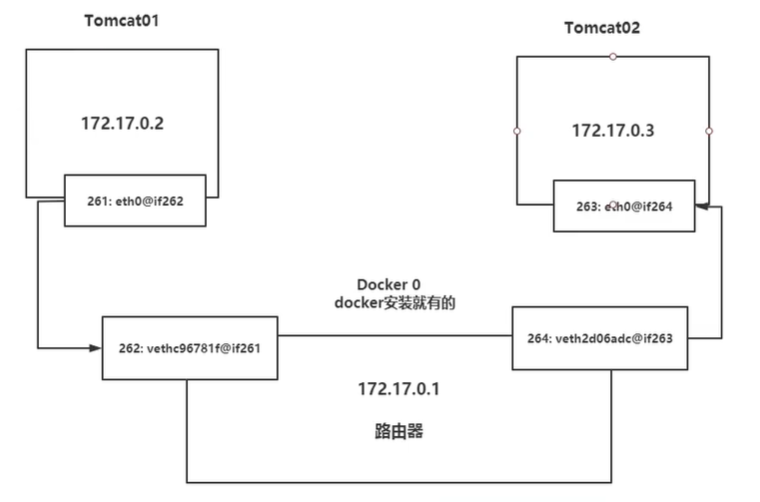
# Docker网络原理

## 14.1 docker容器之间如何访问

原理：我们每启动一个docker容器，docker就会给docker容器分配一个ip，我们只要安装了docker，就会有一个网卡docker0，使用桥接模式，使用的技术veth-pair

veth-pair是一对虚拟设备接口，他们都是成对出现的(容器内和容器外成对出现)，一端连着协议，一端彼此相连

tomcat01和tomcat02是公用的一个网络，docker0，所有的容器在不指定网络的情况下，都是docker0路由的，docker会给容器分配一个默认的可用ip地址

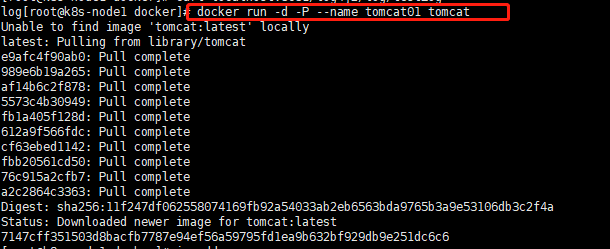


### 14.1.1 容器之间ip可以ping通，但是ping不通域名

（1）查看网络地址：ip addr

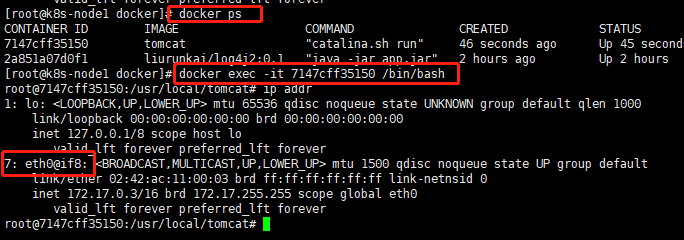


1. 启动一个tomcat

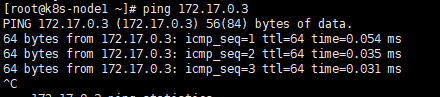


1. 进入容器查看容的网络地址：ip addr

可以看到两个地址：lo和7: eth0@if8

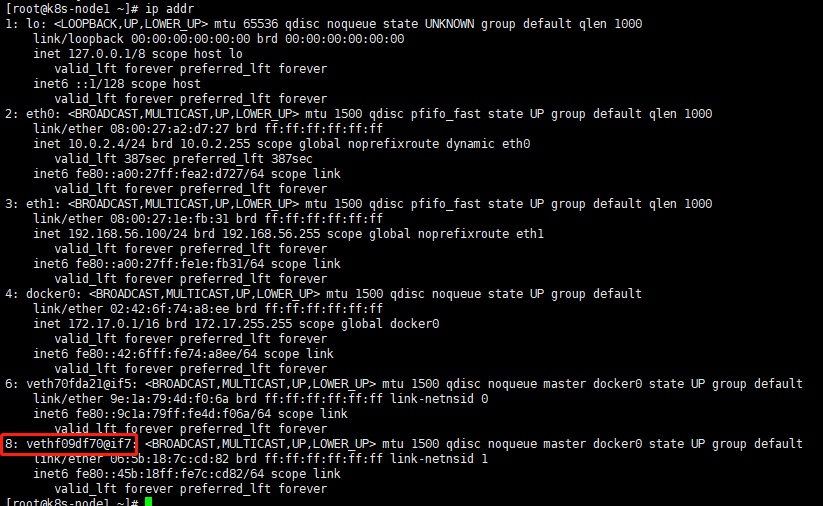


1. 使用linux测试ping 容器地址：可以ping通



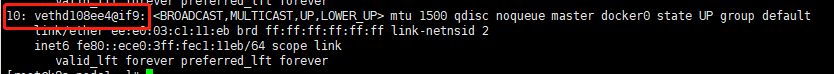
1. 再次查看虚拟机的网络地址：ip addr

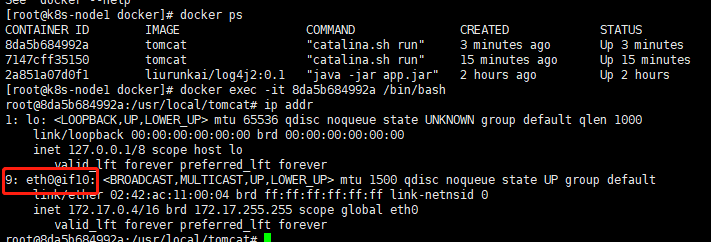
会多出一个地址：8: vethf09df70@if7



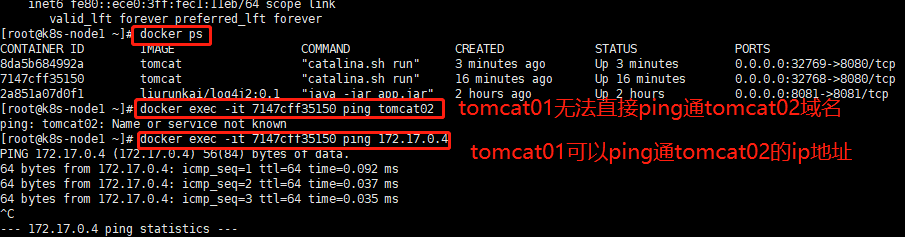
1. 再启动一个tomcat，虚拟机ip addr又多了一对网卡







1. 我们使用tomcat01可以ping通tomcat02的ip地址，但是ping不通tomcat02域名



### 14.1.2 容器之间可以使用域名ping

#### 14.1.2.1 使用--link

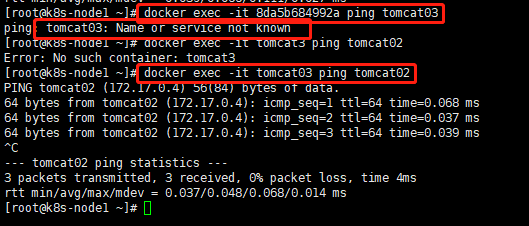
（1）启动一个tomcat03，连接到tomcat02

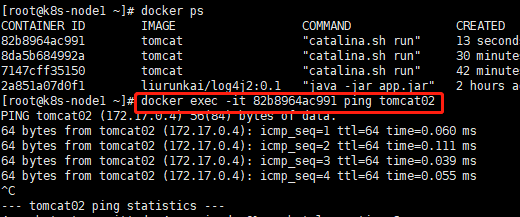
docker run -d -P --name tomcat03 --link tomcat02 tomcat



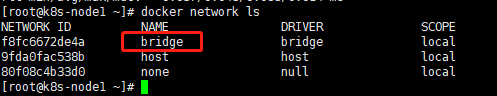
1. docker exec -it 82b8964ac991 ping tomcat02：可以ping通

但是tomcat02不能ping通tomcat03

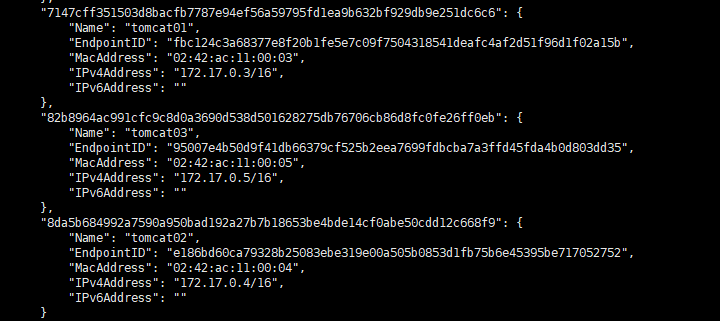




查看docker网络，默认birdge：docker network ls

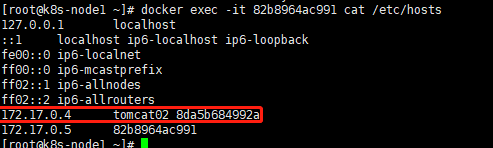


docker network inspect f8fc6672de4a



查看tomcat3容器的host文件

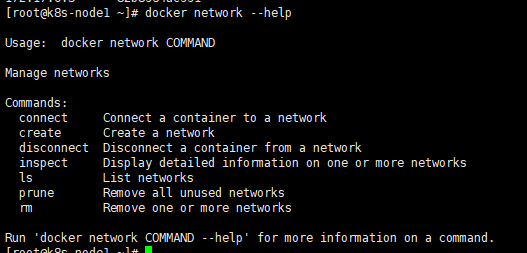
tomcat03可以ping通tomcat02，是因为--link相当于在hosts下面配置tomcat02的ip和域名对应关系



#### 14.1.2.2 自定义网络

好处：不同的集群使用不同的网络，保证了集群的安全和健康，eg：redis和mysql使用不同的网络，但是redis和mysql不同的网络之间可以相互连通

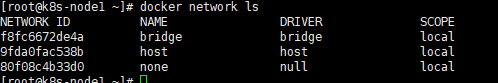
（1）docker network --help



1. 查看docker网络模式：docker network ls

默认3个：birdge(桥接，docker默认)，host(仅主机)，none(不配置网络)

还有一个：container(容器内网络连通)



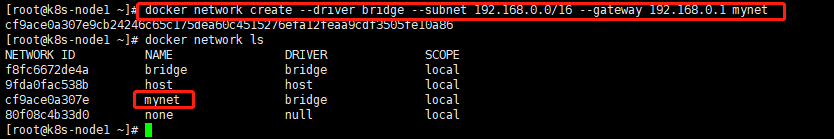
1. 创建自己的网络：

--driver bridge：桥接

--subnet：子网

--gateway：网关

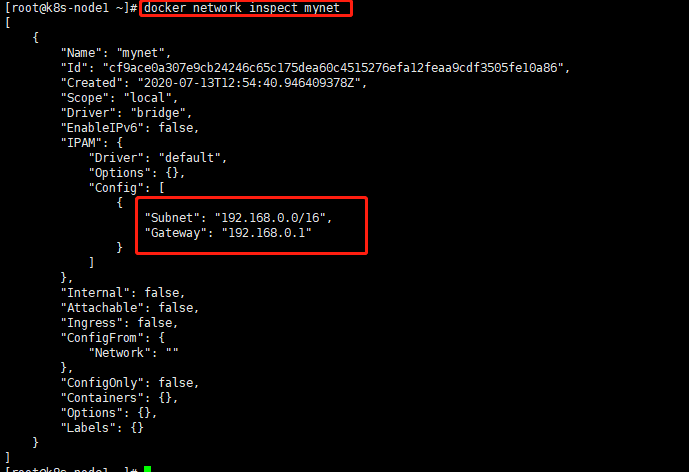
docker network create --driver bridge --subnet 192.168.0.0/16 --gateway 192.168.0.1 mynet



我们之前启动：docker run -d -P --name tomcat01 tomcat相当于如下命令：

docker run -d -P --name tomcat01 --net bridge tomcat

1. 查看自己的网络：docker network inspect mynet



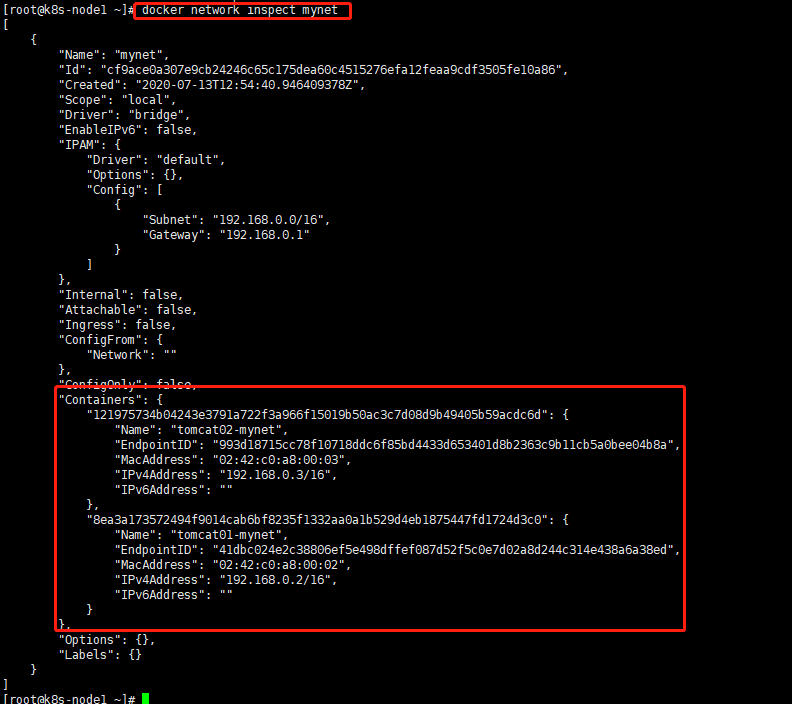
1. 启动两个tomcat

docker run -d -P --name tomcat01-mynet --net mynet tomcat

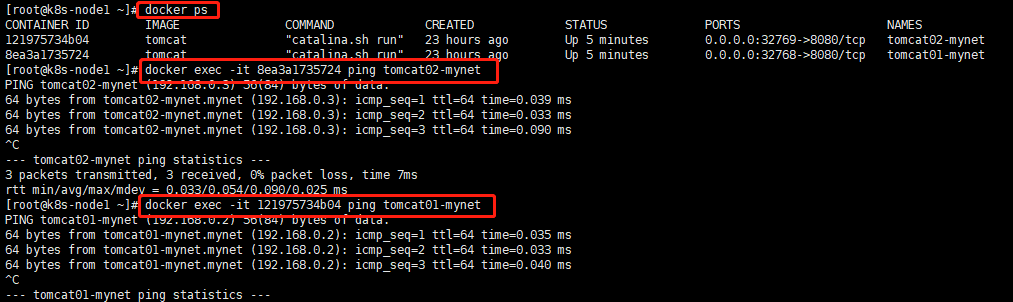
docker run -d -P --name tomcat02-mynet --net mynet tomcat

查看mynet网络内部：tomcat01-mynet和tomcat02-mynet关联

docker network inspect mynet



1. tomcat01-mynet和tomcat02-mynet通过容器名可以相互ping通



#### 14.1.2.3 不同的网络之间连通

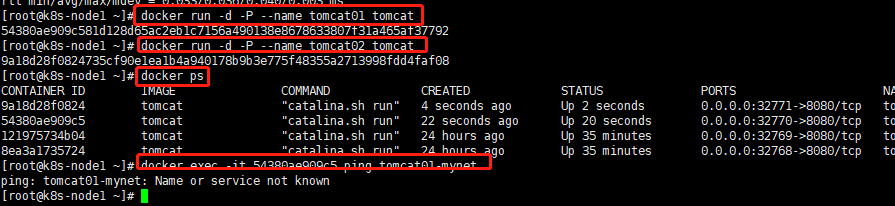
docker0和mynet连通：实质上是docker0网络上面的容器和mynet网络连通，不能直接是docker0和mynet连通

1. 启动两个tomcat，tomcat01和tomcat02，使用默认的docker0网络

docker run -d -P --name tomcat01 tomcat

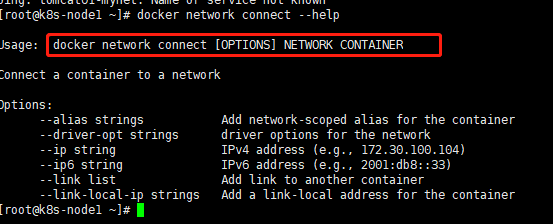
docker run -d -P --name tomcat02 tomcat

现在有4个容器，默认tomcat01和tomcat01-mynet不能连通的



1. 使用docker network connet使docker0容器连通mynet网络

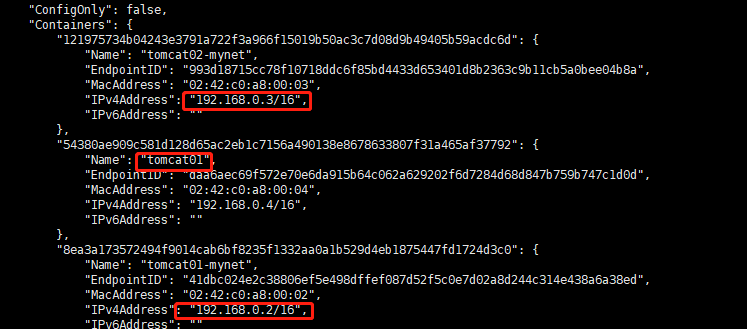
docker network connet 网络 容器名

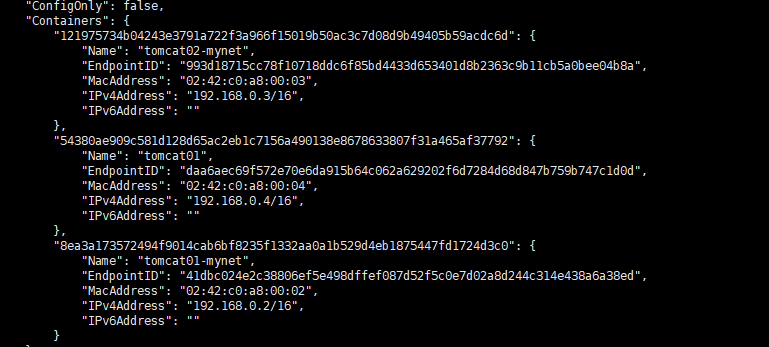


docker network connect mynet tomcat01

连通之后就将tomcat01放到了mynet网络下

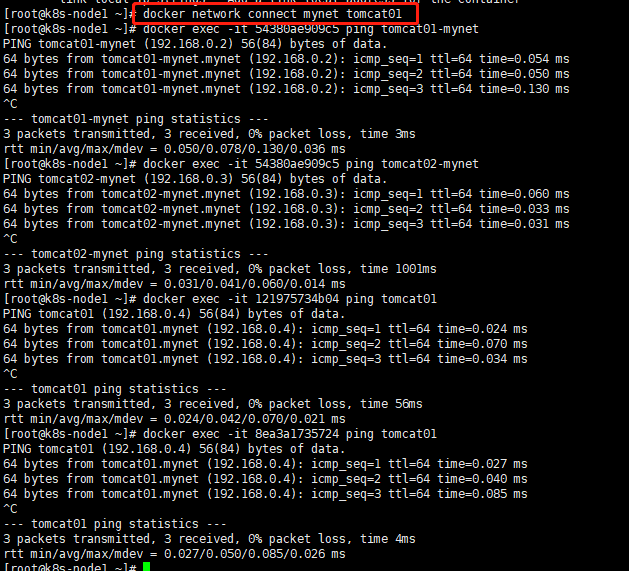
docker network inspect mynet





再次使用tomcat01去ping 网络mynet 下tomcat01-mynet或tomcat02-mynet

或者使用tomcat01-mynet去ping网络docker0下面的tomcat01或tomcat02



# 15. docker底层原理

Docker是一个Client-server的结构系统，docker的守护进程运行在主机上(宿主机)，通过socket从客户端访问，docker-server接收到docker-client的指令，就会执行这个指令

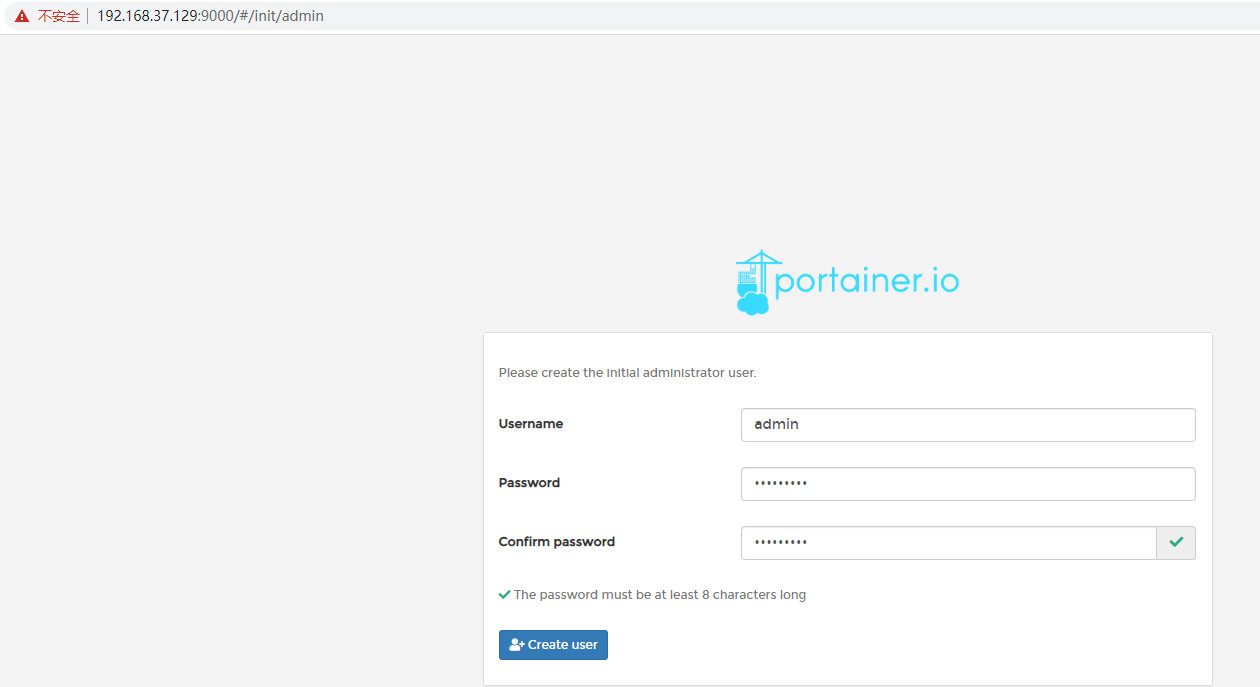
# 16.docker图形化界面管理工具

## 16.1 portainer

（1）安装portainer

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

1. 访问测试

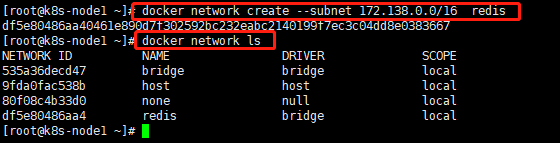


# 实战

## 17.1 使用自己的网络部署redis集群

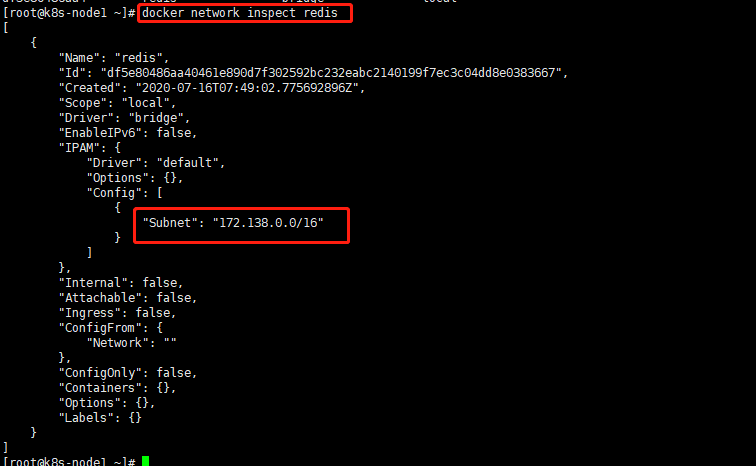
（1）创建redis网络

docker network create --driver bridge --subnet 172.138.0.0/16 --gateway 172.138.0.1 redis



1. 查看redis网络

docker network inspect redis



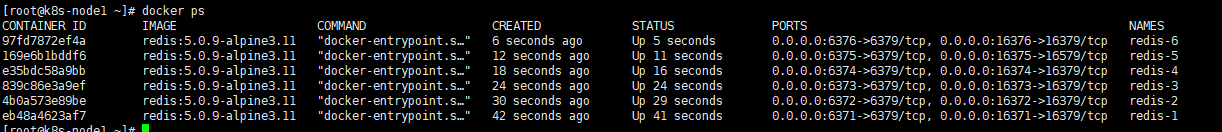
1. 创建redis-node节点

|  |
| --- |
| for port in $(seq 1 6); \ do \ mkdir -p /usr/local/redis-cluster/redis-${port}/conf touch /usr/local/redis-cluster/redis-${port}/conf/redis.conf cat << EOF >>/usr/local/redis-cluster/redis-${port}/conf/redis.conf port 6379 bind 0.0.0.0 cluster-enabled yes cluster-config-file nodes.conf cluster-node-timeout 5000 cluster-announce-port 6379 cluster-announce-bus-port 16379 appendonly yes EOF done |



1. 启动6个redis

|  |
| --- |
| docker run -p 6371:6379 -p 16371:16379 --name redis-1 \ -v /usr/local/redis-cluster/redis-1/data:/data \ -v /usr/local/redis-cluster/redis-1/conf/redis.conf:/etc/redis/redis.conf \ -d --net redis redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf |
| docker run -p 6372:6379 -p 16372:16379 --name redis-2 \ -v /usr/local/redis-cluster/redis-2/data:/data \ -v /usr/local/redis-cluster/redis-2/conf/redis.conf:/etc/redis/redis.conf \ -d --net redis redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf |
| docker run -p 6373:6379 -p 16373:16379 --name redis-3 \ -v /usr/local/redis-cluster/redis-3/data:/data \ -v /usr/local/redis-cluster/redis-3/conf/redis.conf:/etc/redis/redis.conf \ -d --net redis redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf |
| docker run -p 6374:6379 -p 16374:16379 --name redis-4 \ -v /usr/local/redis-cluster/redis-4/data:/data \ -v /usr/local/redis-cluster/redis-4/conf/redis.conf:/etc/redis/redis.conf \ -d --net redis redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf |
| docker run -p 6375:6379 -p 16375:16579 --name redis-5 \ -v /usr/local/redis-cluster/redis-5/data:/data \ -v /usr/local/redis-cluster/redis-5/conf/redis.conf:/etc/redis/redis.conf \ -d --net redis redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf |
| docker run -p 6376:6379 -p 16376:16379 --name redis-6 \ -v /usr/local/redis-cluster/redis-6/data:/data \ -v /usr/local/redis-cluster/redis-6/conf/redis.conf:/etc/redis/redis.conf \ -d --net redis redis:5.0.9-alpine3.11 redis-server /etc/redis/redis.conf |



1. 进入redis-1

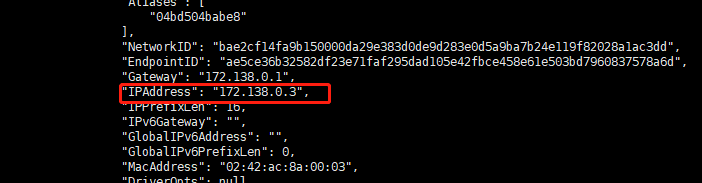
docker exec -it redis-1 /bin/sh

默认的pwd目录是/data目录



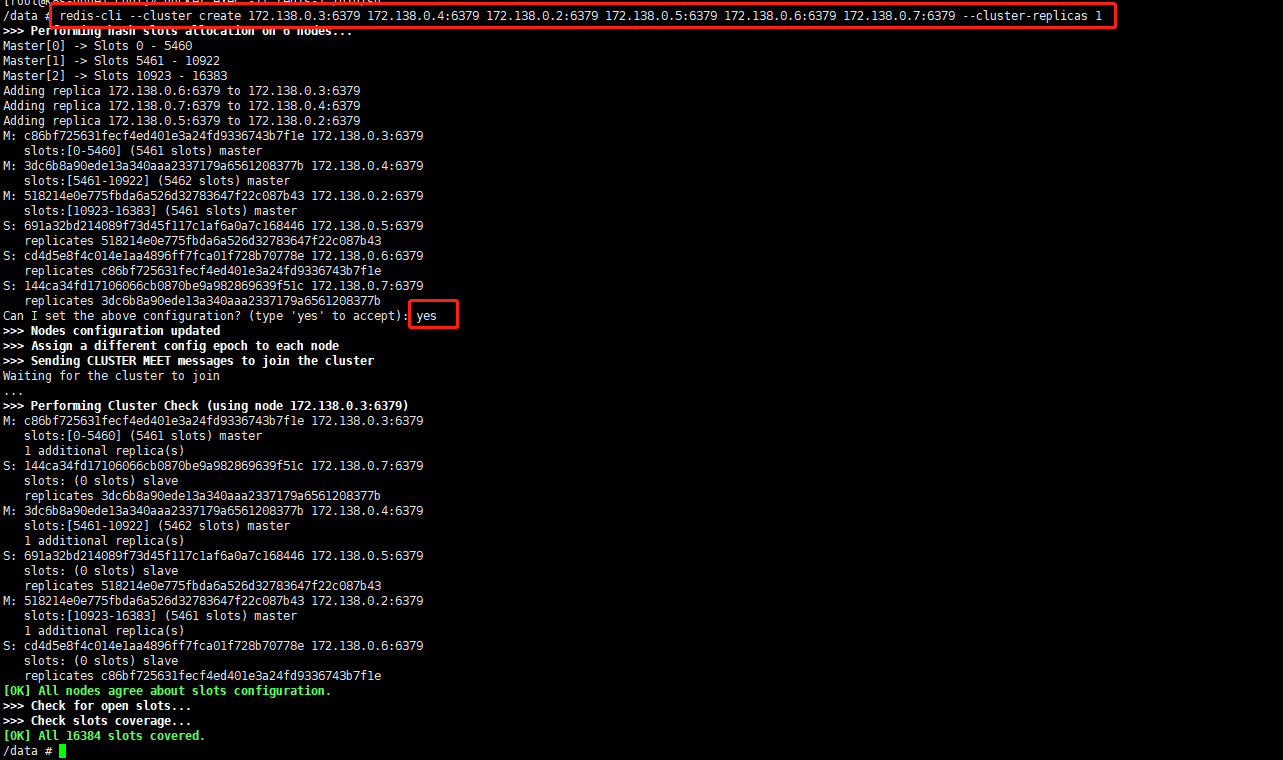
1. 启动集群

docker inspect 容器名：查看容器id地址



在redis-1的data目录下执行：

redis-cli --cluster create 172.138.0.3:6379 172.138.0.4:6379 172.138.0.2:6379 172.138.0.5:6379 172.138.0.6:6379 172.138.0.7:6379 --cluster-replicas 1

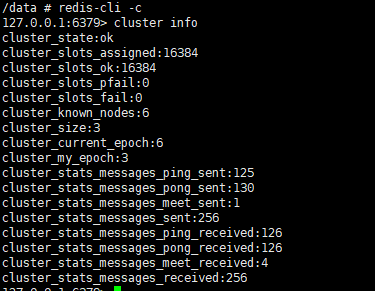


1. 连接集群

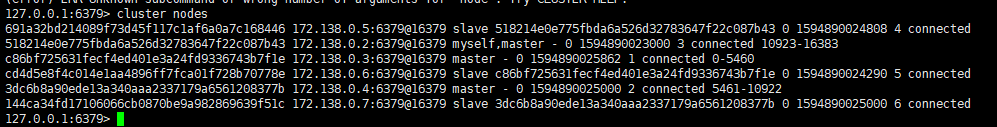
redis-cli -c (redis-cli是连接单机)

1. 查看集群信息

cluster info



cluster nodes



1. 测试高可用，停掉master，会看到slave自动称为master

