

[Docker使用手册]

**[深圳全棉时代有限公司]**

**2018年4月8日**

**[Docker安装、使用、开发部署]**

[1 引言： 1](#_Toc510631951)

[2 简介 1](#_Toc510631952)

[2.1 什么是Docker 1](#_Toc510631953)

[2.2 Docker的优势 1](#_Toc510631954)

[2.3 Docker的不足 2](#_Toc510631955)

[2.4 Docker核心组件 2](#_Toc510631956)

[3 安装Docker 3](#_Toc510631957)

[3.1 安装前准备 3](#_Toc510631958)

[3.1.1 必要条件 3](#_Toc510631959)

[3.1.2 卸载已安装旧版本的Docker 3](#_Toc510631960)

[3.2 开始安装 4](#_Toc510631961)

[4 Docker使用 6](#_Toc510631962)

[4.1 镜像操作 6](#_Toc510631963)

[4.1.1 拉取镜像 6](#_Toc510631964)

[4.1.2 创建镜像 7](#_Toc510631965)

[4.1.3 删除镜像 9](#_Toc510631966)

[4.2 容器操作 9](#_Toc510631967)

[4.2.1 创建应用镜像 9](#_Toc510631968)

[4.2.2 创建容器 11](#_Toc510631969)

[4.2.3 容器重启停 11](#_Toc510631970)

[4.2.4 容器日志 12](#_Toc510631971)

[4.2.5 删除容器 13](#_Toc510631972)

[4.3 仓库操作 13](#_Toc510631973)

[4.3.1 建立本地仓库 13](#_Toc510631974)

[4.3.2 推送镜像到仓库 14](#_Toc510631975)

[4.3.3 从仓库下载镜像 15](#_Toc510631976)

[5 Docker开发部署 15](#_Toc510631977)

[5.1 使用docker-maven-plugin打包部署 15](#_Toc510631978)

[5.1.1 项目准备 15](#_Toc510631979)

[5.1.2 开启docker远程访问 20](#_Toc510631980)

[5.1.3 Maven-Plugin构建镜像 21](#_Toc510631981)

[5.1.4 Maven-Dockerfile构建 23](#_Toc510631982)

[5.2 持续集成部署 25](#_Toc510631983)

[5.2.1 安装Docker Compose工具 25](#_Toc510631984)

[5.2.2 安装DockUI 26](#_Toc510631985)

[5.2.3 配置maven 27](#_Toc510631986)

[5.2.4 接入jenkins 29](#_Toc510631987)

[5.2.5 Docker界面管理 36](#_Toc510631988)

[6 FAQ 39](#_Toc510631989)

[7 附录 44](#_Toc510631990)

[7.1 Docker常用命令: 44](#_Toc510631991)

[7.2 Docker Compose YML语法 46](#_Toc510631992)

# 引言：

本文档指导如何安装Docker、使用，开发部署等

# 简介

## 什么是Docker

Docker是一个开源的引擎，可以轻松的为任何应用创建一个轻量级的、可移植的、自给自足的容器，开发者在笔记本上编译测试通过的容器可以批量地在生产环境中部署，包括VMs（虚拟机）、 bare metal、OpenStack 集群和其他的基础应用平台；

Docker的思想来自于集装箱，集装箱解决了什么问题？在一艘大船上，可以把货物规整的摆放起来。并且各种各样的货物被集装箱标准化了，集装箱和集装箱之间不会互相影响。那么我就不需要专门运送水果的船和专门运送化学品的船了，只要这些货物在集装箱里封装的好好的，那我就可以用一艘大船把他们都运走，docker就是类似的理念。现在都流行云计算了，云计算就好比大货轮。docker就是集装箱

## Docker的优势

1. 轻量级：单机可以轻松支持上百Container，让各种个位数虚拟化的方案相形见绌
2. 快速就绪：一秒以内启动，即使是以资源快速就绪著称的青云IAAS也无法相比
3. 弱安全：Docker能够对多种OS资源进行隔离，但是它本质上依托于内核，因此所有的内核漏洞都是Docker的致命伤
4. 隔离性：可以确保你的应用程序与资源是分隔开的
5. 可移植性：Docker的可移植性非常好，基本在支持各种平台的移植部署
6. 版本控制：Docker容器还可以像git仓库一样，可以让你提交变更到Docker镜像中并通过不同的版本来管理它们

## Docker的不足

1. Docker是基于Linux 64bit的，无法在32bit的linux/Windows/unix环境下使用
2. Docker是基于LXC技术开发的，所以对linux内核要求比较高
3. Docker版本不稳定，自13.x版本之后Dokcer的使用变化很大

## Docker核心组件

1. 镜像（image）： Docker镜像类似于虚拟机镜像，是一个只读模板，并且包含了文件系统。
2. 容器（container）: 容器是从镜像创建的运行实例，利用容器来开发、运行应用，可以将其启动、开始。停止、删除，而这些容器都是相互隔离（独立进程），互不可见的。
3. 仓库（repository）：仓库是集中存放镜像文件的场所，每个仓库中又包含了多个镜像，每个镜像有不同的标签

# 安装Docker

## 安装前准备

### 必要条件

* + Linux64位操作系统，不支持windows
  + 操作系统内核版本为 3.10以上，最好是centos 7以上
  + 确认linux系统已安装yum-utils，device-mapper-persistent-data，lvm2

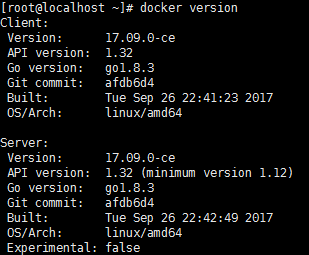
查看linux内核命令：uname -r



注：Docker在2016年很早的时候就明确了将会在企业级方面重点跟进。而在短短的一年时间之内推出的1.12和1.13的版本在功能上确实是很大的进步。而在2017年的3月1号之后，Docker的版本命名开始发生变化，同时将CE版本(社区版本)和EE版本（EE企业版）进行分开

### 卸载已安装旧版本的Docker

检查docker版本docker version



**如果版本过旧建议安装最新版本，以下命令是卸载docker**

sudo yum remove docker \

docker-common \

docker-selinux \

docker-engine

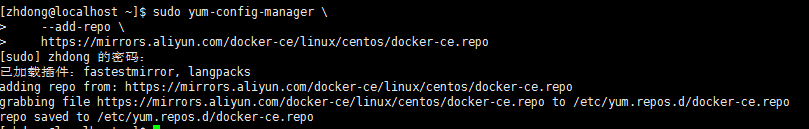
## 开始安装

我们选用Docker-CE版本进行安装，以下是Docker-CE在Cent OS7上安装步骤

1. 配置yum库，因为受国外网速影响，我们添加阿里云源

sudo yum-config-manager \

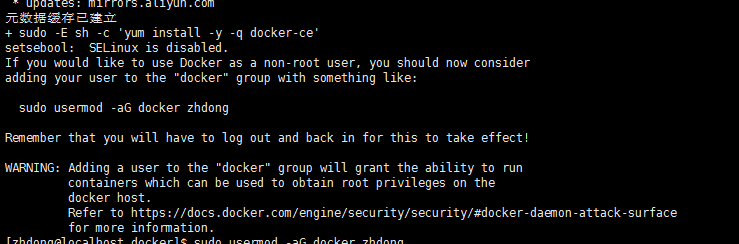
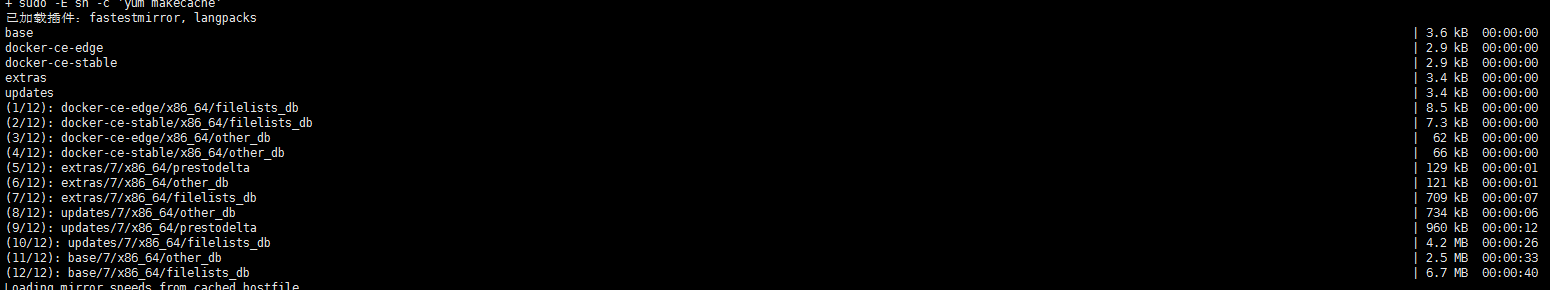
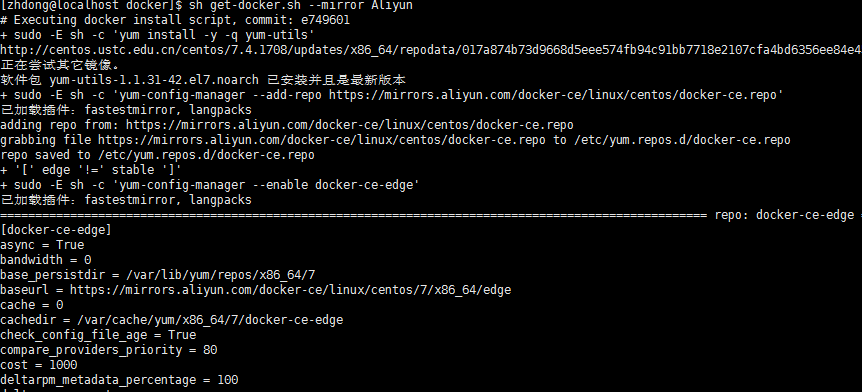
--add-repo \

https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

1. 通过官网提供的一键安装脚本进行安装

yum clean all

curl -fsSL get.docker.com -o get-docker.sh

sh get-docker.sh --mirror Aliyun 

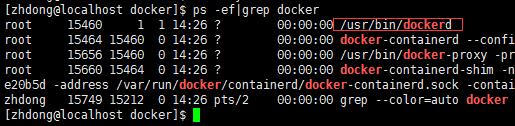
出现警告，安装警告的指示执行命令将当前用户添加到docker组

sudo usermod -aG docker zhdong

1. 启动docker

sudo systemctl start docker

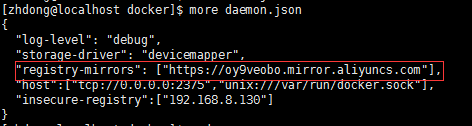
ps –ef|grep docker



1. 配置镜像加速

因为在构建镜像的时候需要用到依赖基础镜像，如Centos镜像，所以需要从官网服务器上下载镜像，由于国外网络非常慢，所以就采用镜像代理来进行加速

修改/etc/docker/daemon.json文件，配置我的镜像加速器（镜像加速器可以从阿里上申请，这个是免费的）



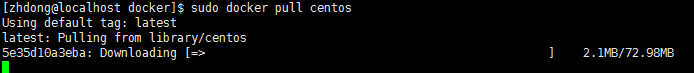
# Docker使用

## 镜像操作

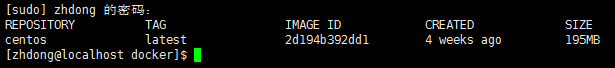
### 拉取镜像

通过docker pull命令可以拉取镜像

拉取centos基础镜像

sudo docker pull centos

查看镜像sudo docker images



### 创建镜像

以下是创建jdk镜像实例（前提是要有centos基础镜像）

1. 从svn下载jdk8

wget <https://172.16.100.176/repos/group4/PurCottonMiddleware/01-Tools/jdk-8u161-linux-x64.tar.gz>

解压到/opt/docker/jdk8（目录自己定）目录

1. 删除jdk多余文件以简化镜像

|  |
| --- |
| rm -rf ./jdk8/\*src.zip  rm -rf ./jdk8/lib/missioncontrol  rm -rf ./jdk8/lib/visualvm  rm -rf ./jdk8/lib/\*javafx\*  rm -rf ./jdk8/jre/lib/plugin.jar  rm -rf ./jdk8/jre/lib/ext/jfxrt.jar  rm -rf ./jdk8/jre/bin/javaws  rm -rf ./jdk8/jre/lib/javaws.jar  rm -rf ./jdk8/jre/lib/desktop  rm -rf ./jdk8/jre/plugin  rm -rf ./jdk8/jre/lib/deploy\*  rm -rf ./jdk8/jre/lib/\*javafx\*  rm -rf ./jdk8/jre/lib/\*jfx\*  rm -rf ./jdk8/jre/lib/amd64/libdecora\_sse.so  rm -rf ./jdk8/jre/lib/amd64/libprism\_\*.so  rm -rf ./jdk8/jre/lib/amd64/libfxplugins.so  rm -rf ./jdk8/jre/lib/amd64/libglass.so  rm -rf ./jdk8/jre/lib/amd64/libgstreamer-lite.so  rm -rf ./jdk8/jre/lib/amd64/libjavafx\*.so  rm -rf ./jdk8/jre/lib/amd64/libjfx\*.so |

1. 在jdk目录下创建Dockerfile，编写以下脚本

|  |
| --- |
| #使用的基础镜像  FROM centos  #把当前目录下的jdk文件夹添加到镜像  RUN mkdir -p /opt/docker/jdk8  ADD . /opt/docker/jdk8  #设置java环境  ENV JAVA\_HOME /opt/docker/jdk8  ENV PATH ${PATH}:${JAVA\_HOME}/bin |

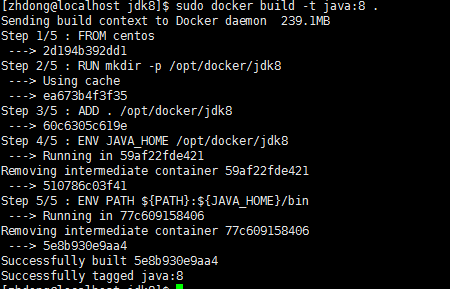
执行命令创建镜像

sudo docker build -t java:8 .

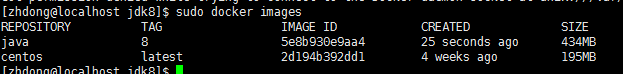
build为docker构建镜像命令，参数：

-t 设置tag名称, 命名规则仓库名/镜像名:tag，仓库名默认是本地就不需要打标签了，镜像名称一定要小写

. 表示使用当前目录下的Dockerfile文件



查看镜像是否创建成功sudo docker images



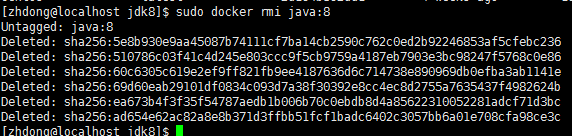
这样一个jdk的镜像就构建成功了

### 删除镜像

sudo docker rmi 镜像id或者名称

例如：

sudo docker rmi java:8



批量删除镜像操作

sudo docker rmi img1 image2 image3

如果镜像存在关联，可以通过如下命令强制删除镜像

sudo docker rmi -f 镜像id或者名称

## 容器操作

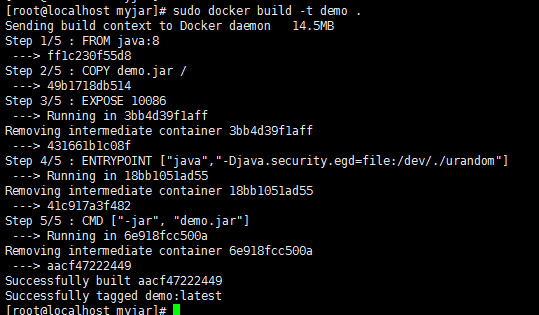
### 创建应用镜像

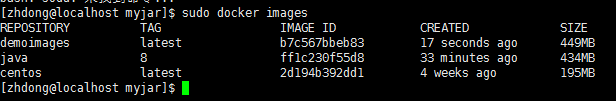
1. 准备一个可运行的spring boot的jar包，能通过java –jar xxx.jar命令运行
2. 将jar包上传到服务器/opt/docker/myjar目录下，在目录下创建Dockerfile文件

|  |
| --- |
| FROM java:8  #设置java环境  #COPY  COPY demo.jar /  EXPOSE 18888  # ENTRYPOINT  ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom"]  # CMD  CMD ["-jar", "demo.jar"] |

1. 构建demo镜像

sudo docker build -t demoimages .





### 创建容器

创建容器命令，创建容器后就会自动启动

sudo docker run -d --name democontainer -p 10086:18888 demoimages

参数说明：

run:容器创建命令

-d : 后台启动

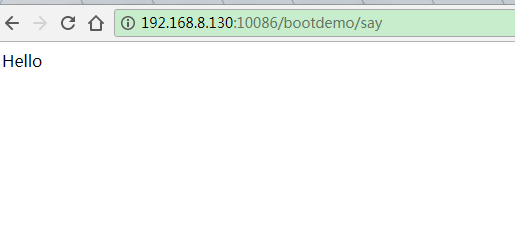
--name:容器名称，可以不指定，系统会默认值给名称

-p:端口映射，宿主机端口：容器端口（Dockerfile里面的EXPOSE 10086）

查看容器是否启动sudo docker ps



用宿主机访问页面：



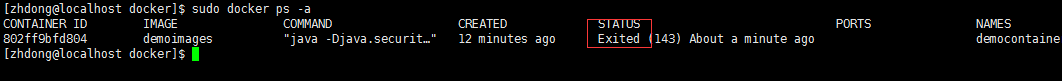
### 容器重启停

容器停止sudo docker stop democontainer





已停止的容器docker ps是默认无法查询到，通过-a参数可以查到所有



Status=Exited..表示容器已经停止了

启动容器：

sudo docker start democontainer



检查是否启动



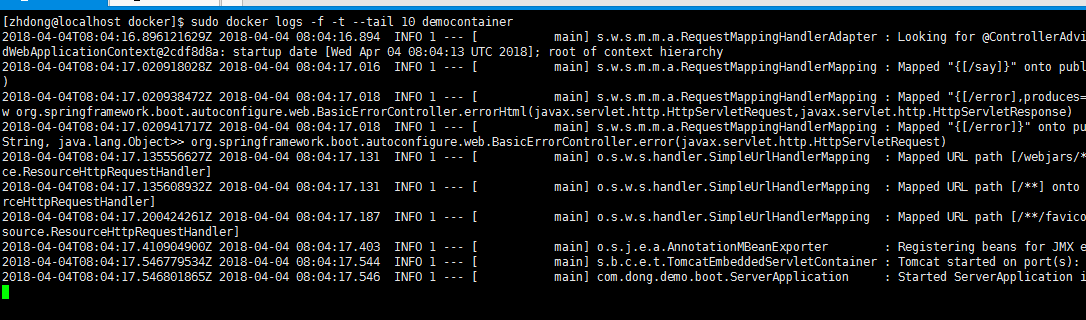
重启容器：

sudo docker restart democontainer



### 容器日志

实时查看容器最近10条日志

sudo docker logs -f -t --tail 10 democontainer

### 删除容器

删除容器

sudo docker rm democontainer



如果正在运行的话，可以通过如下命令强制删除容器

sudo docker rm -f democontainer

批量删除容器

sudo docker rm container1 container2 container3

删除所有容器

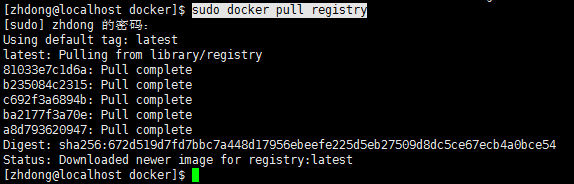
sudo docker rm -f `sudo docker ps -a -q`

## 仓库操作

### 建立本地仓库

1. 下载仓库镜像

sudo docker pull registry



1. 建立本地仓库容器

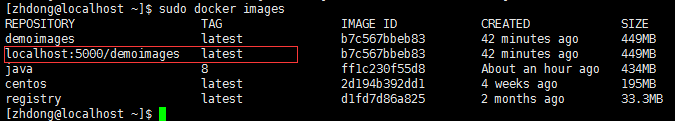
docker run -d -v /home/zhdong/registry:/var/lib/registry -p 5000:5000 --restart=always --name registry registry:latest

* -v /home/zhdong/registry:/var/lib/registry 默认情况下，会将仓库存放于容器内的/var/lib/registry目录下，指定本地目录挂载到容器。
* -p 5000:5000 端口映射
* --restart=always1 在容器退出时总是重启容器,主要应用在生产环境

### 推送镜像到仓库

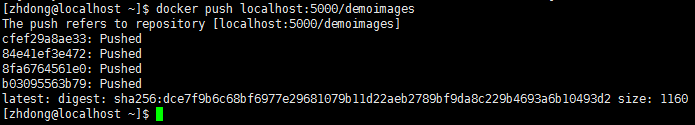
1. 为镜像打标签

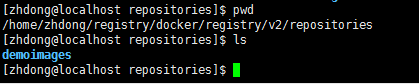
sudo docker tag demoimages localhost:5000/demoimages



1. 推送到本地仓库

**sudo docker push localhost:5000/demoimages**

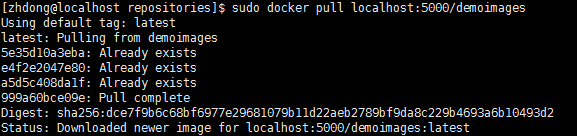




可以到仓库文件夹下看到刚才推送的仓库名称

### 从仓库下载镜像

**sudo docker pull localhost:5000/demoimages**

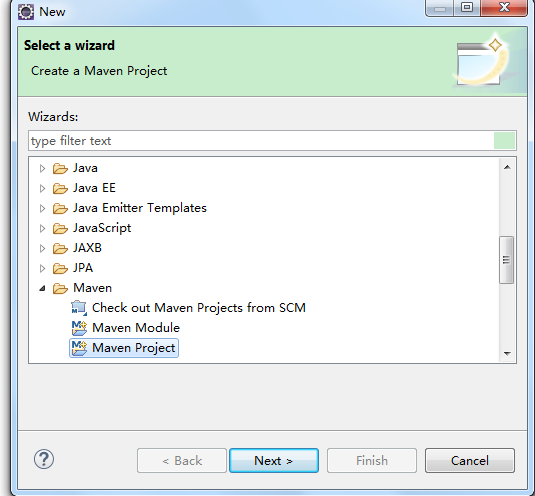


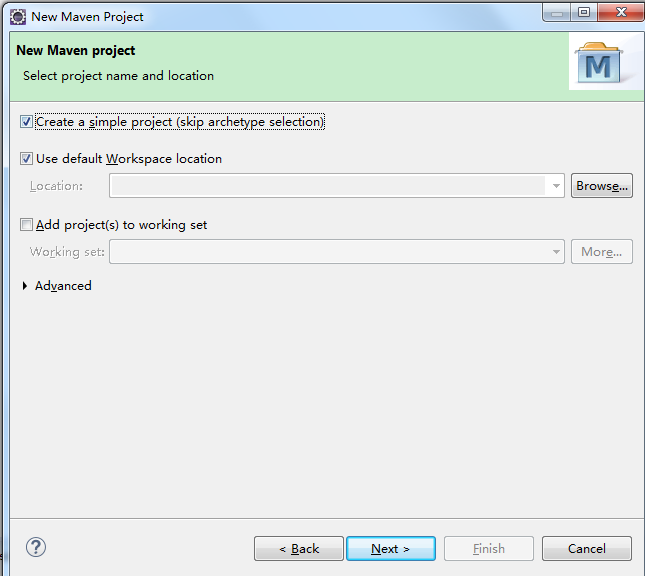
# Docker开发部署

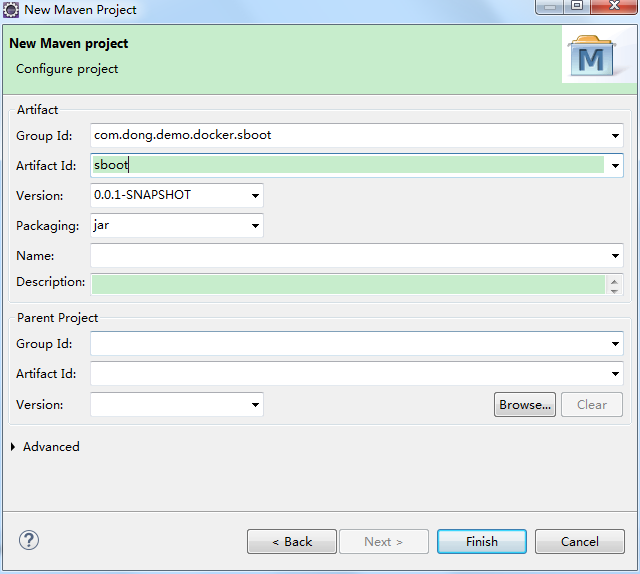
## 使用docker-maven-plugin打包部署

### 项目准备

1. 建立spring boot的工程，如果有spring boot项目跳过此步骤







parent坐标

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.8.RELEASE</version>  <relativePath />  </parent> |

依赖项坐标:

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

1. 建立启动类

|  |
| --- |
| **package** com.dong.demo.boot;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  /\*\*  \* 启动类入口  \*  \* **@author** zhdong  \* **@version** 1.0, 2018年3月22日  \* **@since** JDK1.8  \*/  @SpringBootApplication  **public** **class** ServerApplication  {  **public** **static** **void** main(String[] args)  {  SpringApplication.*run*(ServerApplication.**class**, args);  }  } |

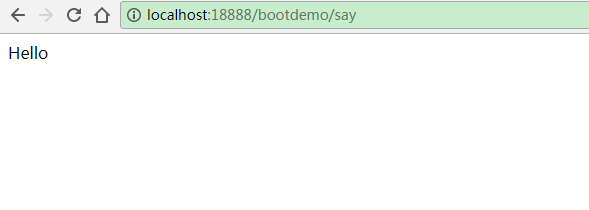
1. 建立Controller

|  |
| --- |
| **package** com.dong.demo.boot.controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** DemoController  {  @RequestMapping(value = "/say")  **public** String say()  {  **return** "Hello";  }  } |

1. src/main/resource目录下建立application.properties

|  |
| --- |
| server.port=18888  server.contextPath=/bootdemo  #http encoding  spring.http.encoding.charset=UTF-8  spring.http.encoding.enabled=true  spring.http.encoding.force=true |

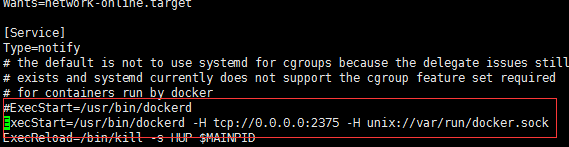
1. 确认启动访问正常



### 开启docker远程访问

1. 修改/usr/lib/systemd/system/docker.service配置，替换ExecStart配置如下：

ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix://var/run/docker.sock

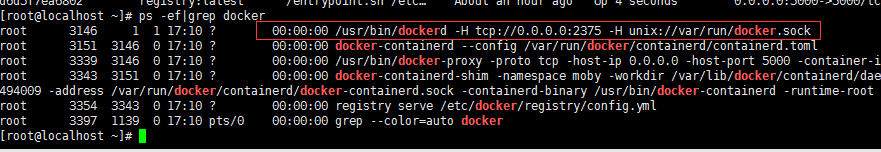


1. 重启docker

systemctl daemon-reload

systemctl restart docker

查看启动是否成功

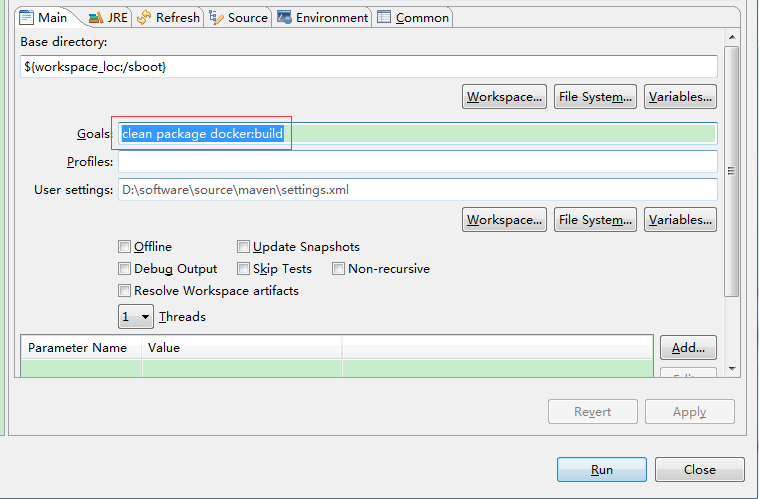


### Maven-Plugin构建镜像

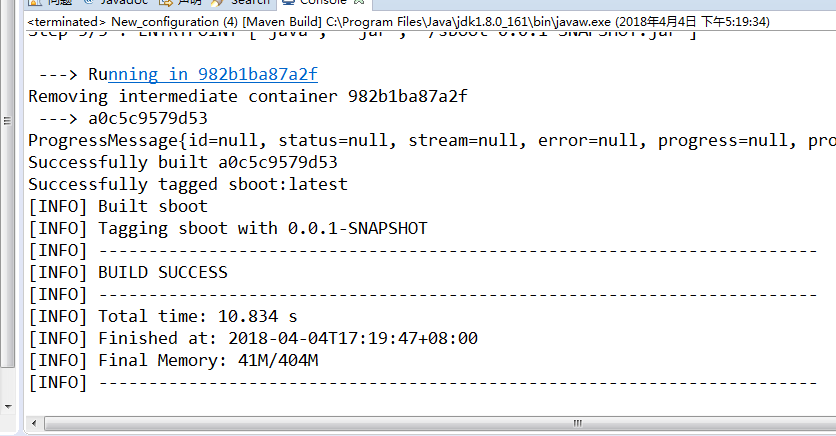
1. Maven-docker打包插件配置如下：

|  |
| --- |
| <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  <plugin>  <groupId>com.spotify</groupId>  <artifactId>docker-maven-plugin</artifactId>  <version>0.4.14</version>  <configuration>  <!-- 注意imageName一定要是符合正则[a-z0-9-\_.]的，而且是只能小写，否则构建不会成功 -->  <!-- 详见：https://github.com/spotify/docker-maven-plugin Invalid repository name ... only [a-z0-9-\_.] are allowed-->  <imageName>${project.artifactId}</imageName>  <forceTags>true</forceTags>  <imageTags>  <imageTag>${project.version}</imageTag>  </imageTags>  <baseImage>java:8</baseImage>  <entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>  <dockerHost>http://192.168.8.130:2375</dockerHost>  <resources>  <resource>  <targetPath>/</targetPath>  <directory>${project.build.directory}</directory>  <include>${project.build.finalName}.jar</include>  </resource>  </resources>  </configuration>  </plugin>  </plugins> |

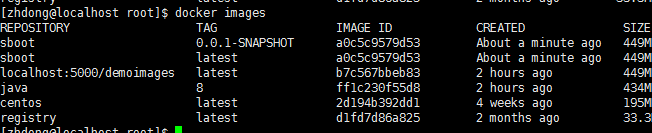
1. 新建打包Run配置，配置Goals：clean package docker:build



1. 运行run



查看docker镜像是否成功创建



### Maven-Dockerfile构建

1. 打包还是使用的docker-plugin插件，但是构建镜像的命令已经转移到Dockfile文件了：

|  |
| --- |
| <plugin>  <groupId>com.spotify</groupId>  <artifactId>docker-maven-plugin</artifactId>  <version>0.4.14</version>  <configuration>  <imageName>${project.artifactId}</imageName>  <forceTags>true</forceTags>  <imageTags>  <imageTag>${project.version}</imageTag>  </imageTags>  <!-- 指定docker文件所在的目录 -->  <dockerDirectory>${project.basedir}/target/docker</dockerDirectory>  <dockerHost>http://192.168.8.130:2375</dockerHost>  <resources>  <resource>  <targetPath>/</targetPath>  <directory>${project.build.directory}</directory>  <include>${project.build.finalName}.jar</include>  </resource>  </resources>  </configuration>  </plugin> |

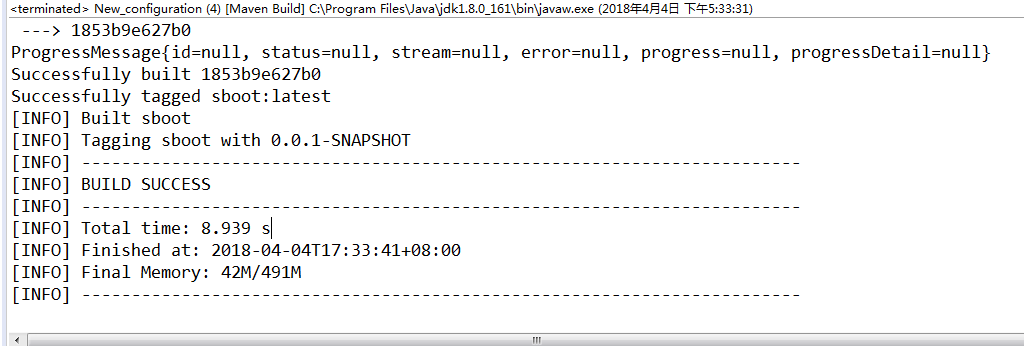
1. 编写Dockerfile

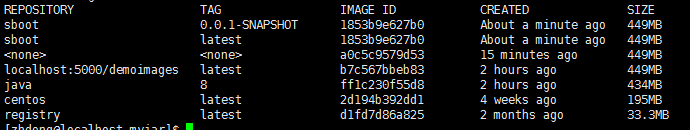
|  |
| --- |
| FROM java:8  #设置java环境  #COPY  COPY @project.build.finalName@.@project.packaging@ /  EXPOSE 18888  # ENTRYPOINT  ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom"]  # CMD  CMD ["-jar", "@project.build.finalName@.@project.packaging@"] |

1. 修改pom，打包时替换Dockerfile的占位符

|  |
| --- |
| <resources>  <!-- 使用@@站位符，输出Dockerfile至docker文件夹 -->  <resource>  <directory>src/main/docker</directory>  <filtering>true</filtering>  <includes>  <include>\*\*/Dockerfile </include>  </includes>  <targetPath>${project.build.directory}/docker</targetPath>  </resource>  </resources> |

1. 运行打包命令

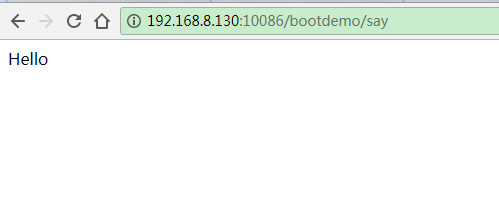




1. 启动容器访问

sudo docker run -d --name democontainer -p 10086:18888 sboot:0.0.1-SNAPSHOT

**访问页面**



## 持续集成部署

### 安装Docker Compose工具

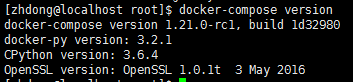
使用root用户，输入安装命令：

curl -L https://github.com/docker/compose/releases/download/1.21.0-rc1/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

然后执行：

chmod +x /usr/local/bin/docker-compose

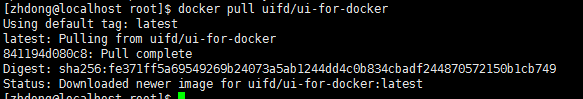
检查安装结果docker-compose --version



### 安装DockUI

1. 拉取dockerui镜像

sudo docker pull uifd/ui-for-docker

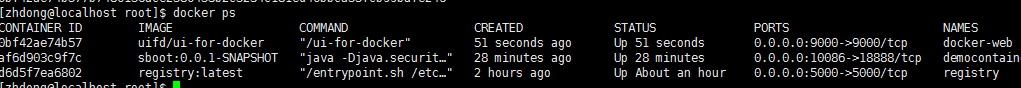


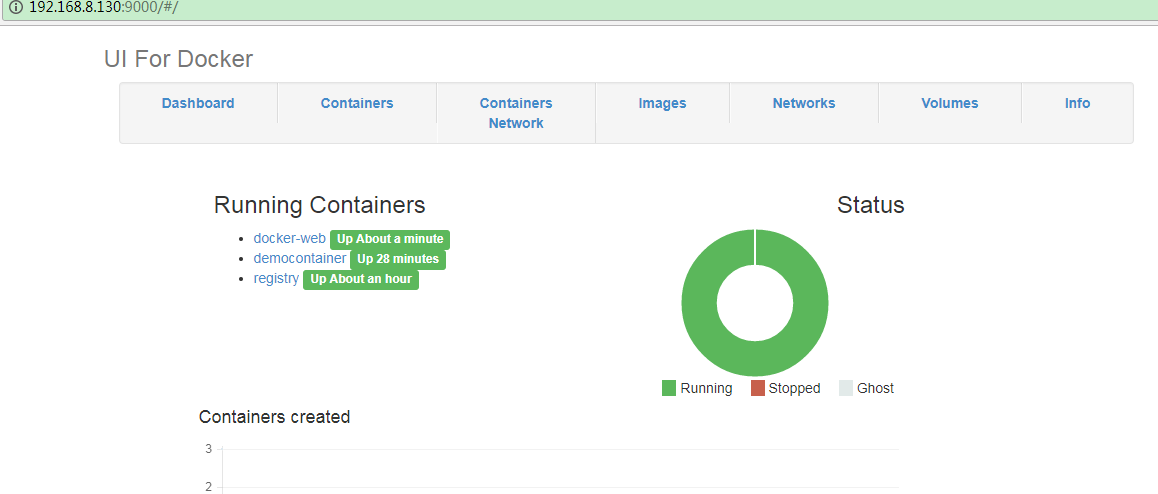
1. 启动dockerUI容器

sodu docker run -it -d --name docker-web -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.sock docker.io/uifd/ui-for-docker



查看容器启动状态





### 配置maven

1. 在src/main/docker下面建立docker-compose.yml文件

|  |
| --- |
| #设置docker-compose version，仓库官网docker-compsose与docker版本对应关系  version: "2.3"  services:  @project.build.finalName@:  #构建镜像  build:  context: ../  dockerfile: docker/Dockerfile  #镜像文件  image: @project.artifactId@:@project.version@  #restart策略  #no – 容器退出时不要自动重启。这个是默认值  #on-failure[:max-retries] – 只在容器以非0状态码退出时重启,可选的，可以退出docker daemon尝试重启容器的次数。  #always – 不管退出状态码是什么始终重启容器，当指定always时，docker daemon将无限次数地重启容器。  #unless-stopped – 不管退出状态码是什么始终重启容器，不过当daemon启动时，如果容器之前已经为停止状态，不要尝试启动它  restart: always  #容器名称  container\_name: @project.artifactId@  #端口映射，数组格式  ports:  - "18888:18888"  #卷轴映射，数组格式，提供给应用写入日志并保存到本地磁盘，容器销毁不会删除实际目录  volumes:  - /applog/demo:/applog/demo |

1. 在src/main/docker下面建立Dockerfile文件

|  |
| --- |
| FROM java:8  #设置java环境  #COPY  COPY sboot-0.0.1-SNAPSHOT.jar /  EXPOSE 18888  # ENTRYPOINT  ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom"]  # CMD  CMD ["-jar", "sboot-0.0.1-SNAPSHOT.jar"] |

1. 修改pom，打包时替换Dockerfile的占位符

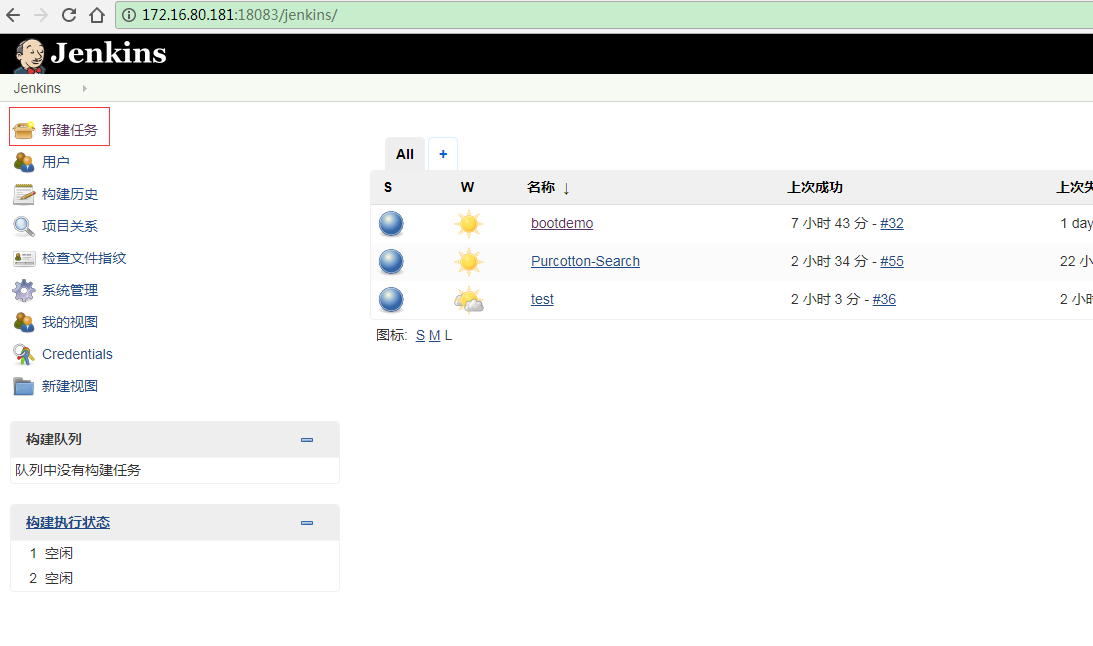
|  |
| --- |
| <resources>  <!-- 使用@@站位符，输出Dockerfile至docker文件夹 -->  <resource>  <directory>src/main/docker</directory>  <filtering>true</filtering>  <includes>  <include>\*\*/Dockerfile</include>  <include>\*\*/docker-compose.yml</include>  </includes>  <targetPath>${project.build.directory}/docker</targetPath>  </resource>  <resource>  <directory>src/main/resources</directory>  <!-- filtering属性用来表示资源文件中的占位符是否需要被替换 -->  <filtering>true</filtering>  <includes>  <include>application.properties</include>  <include>log4j.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <!-- 配置文件外置（在打包时取消注释） -->  <targetPath>${project.build.directory}/classes</targetPath>  </resource>  </resources> |

1. 修改pom，配置打包插件，只需要spring boot的插件就可以了

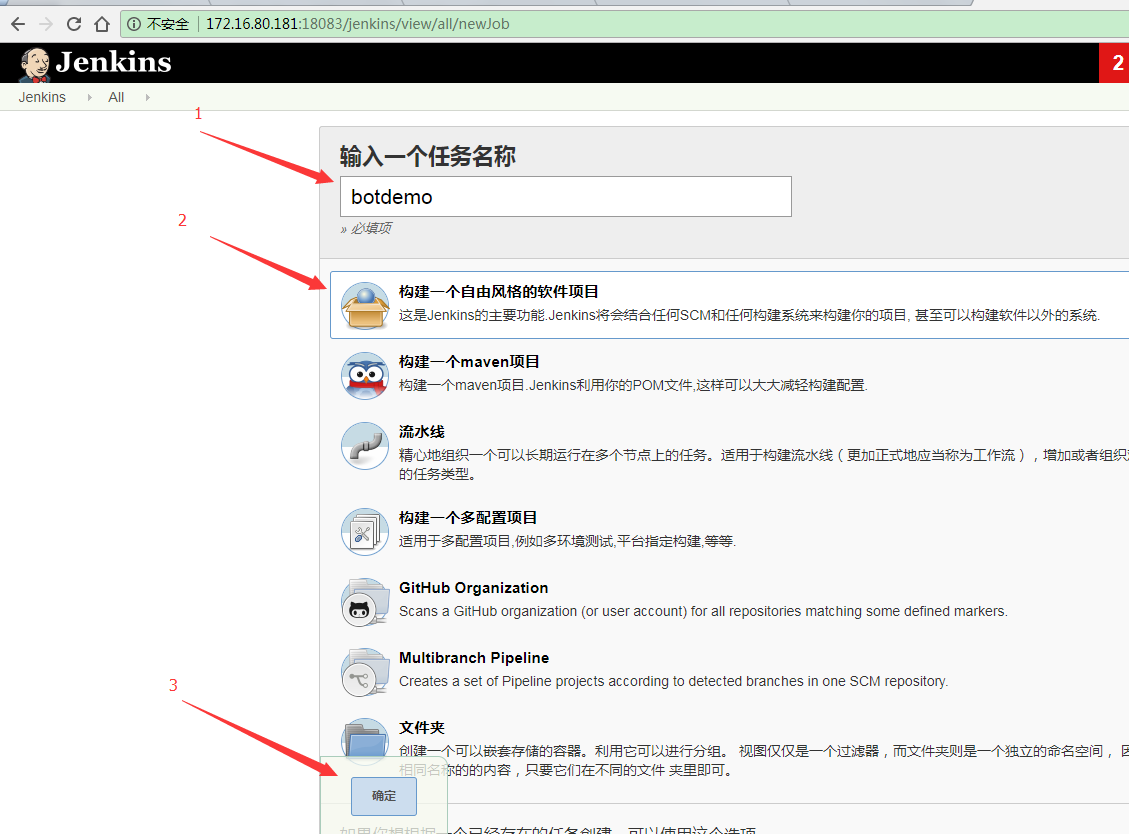
|  |
| --- |
| <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins> |

### 接入jenkins

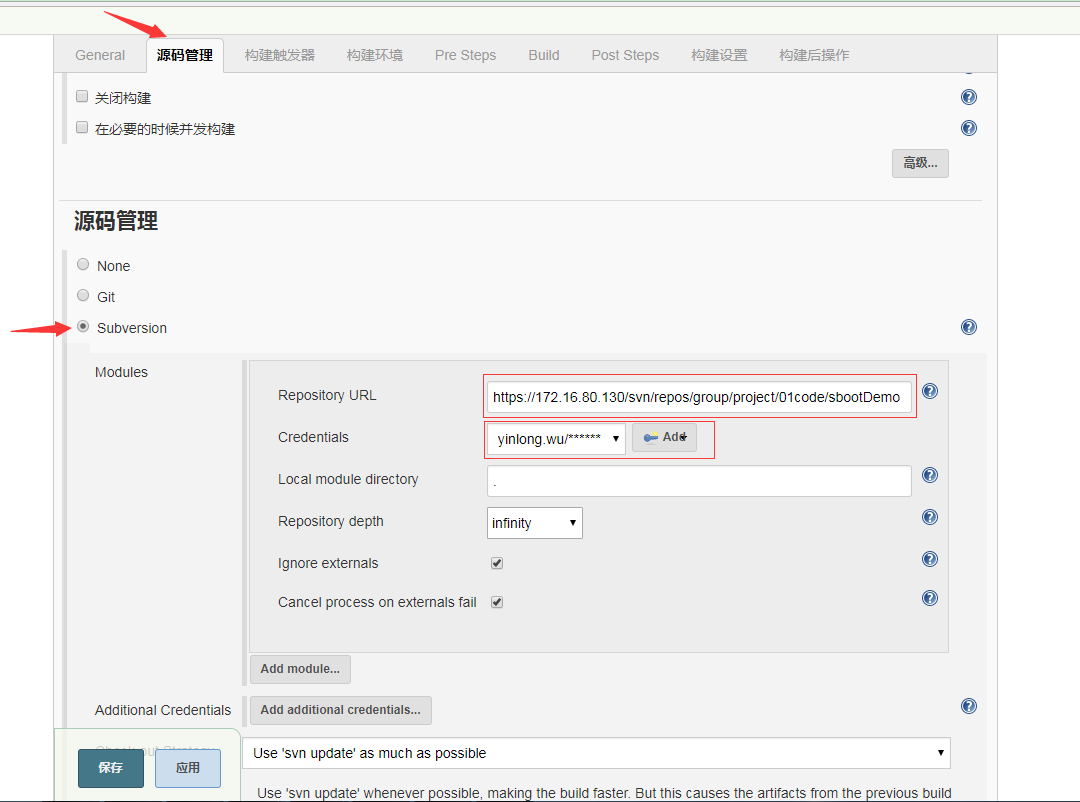
1. 进入jenkins控制台建立任务



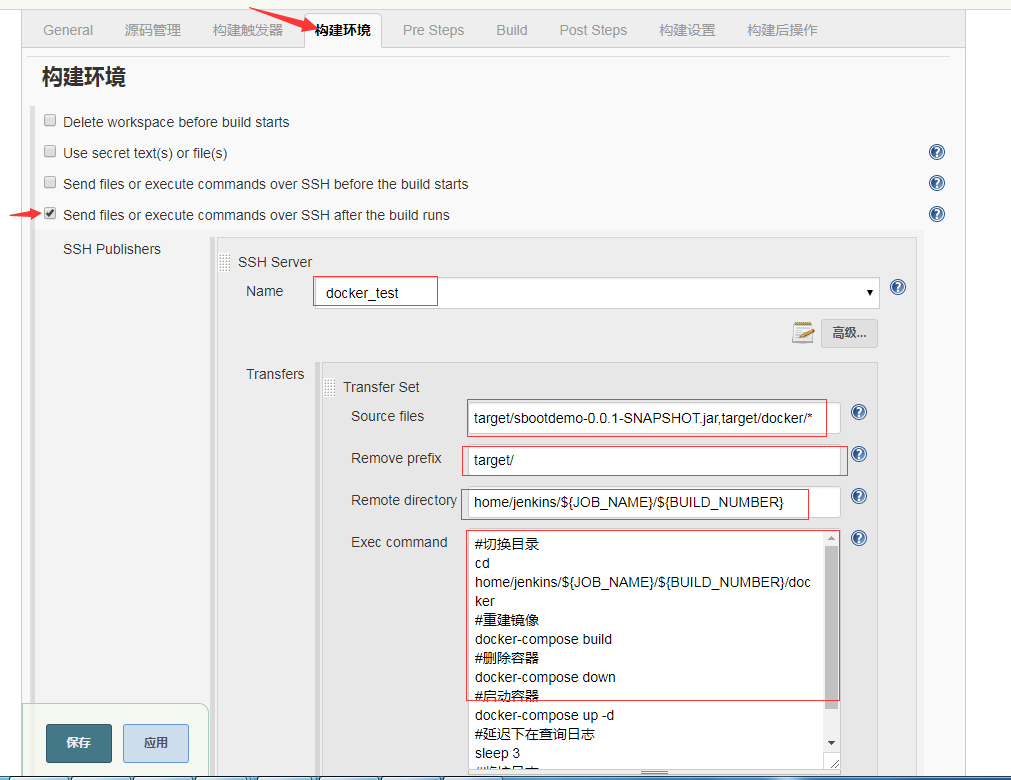
1. 建立自由风格的软件项目



1. 填写svn路径

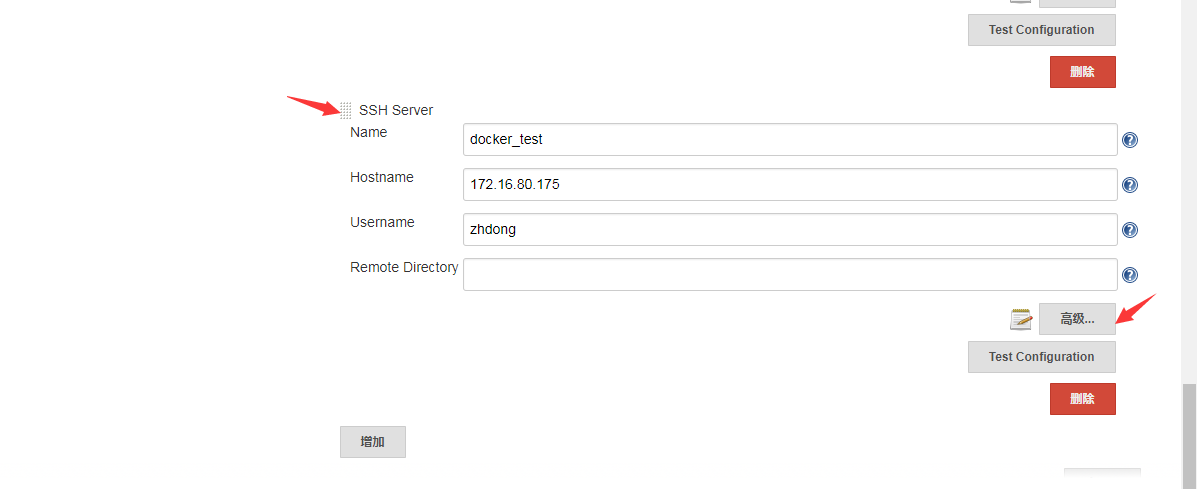


1. 切换“构建环境”页签，填写构建之后运行的命令

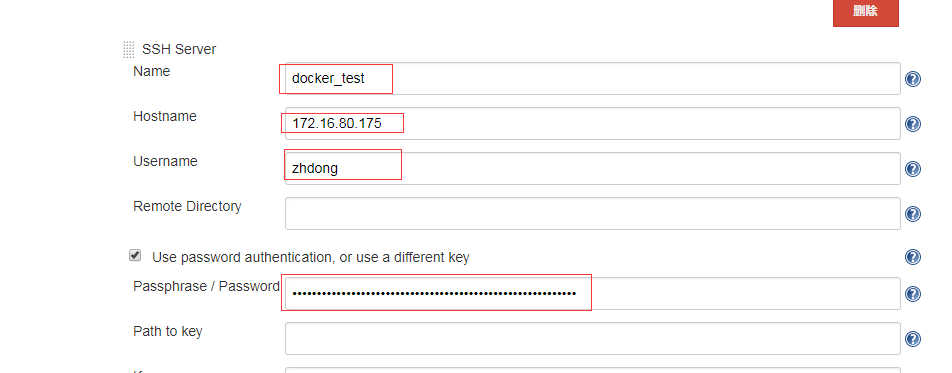


这里需要配置SSH Server服务器信息





配置服务器信息



Source files: target/sbootdemo-0.0.1-SNAPSHOT.jar,target/docker/\*

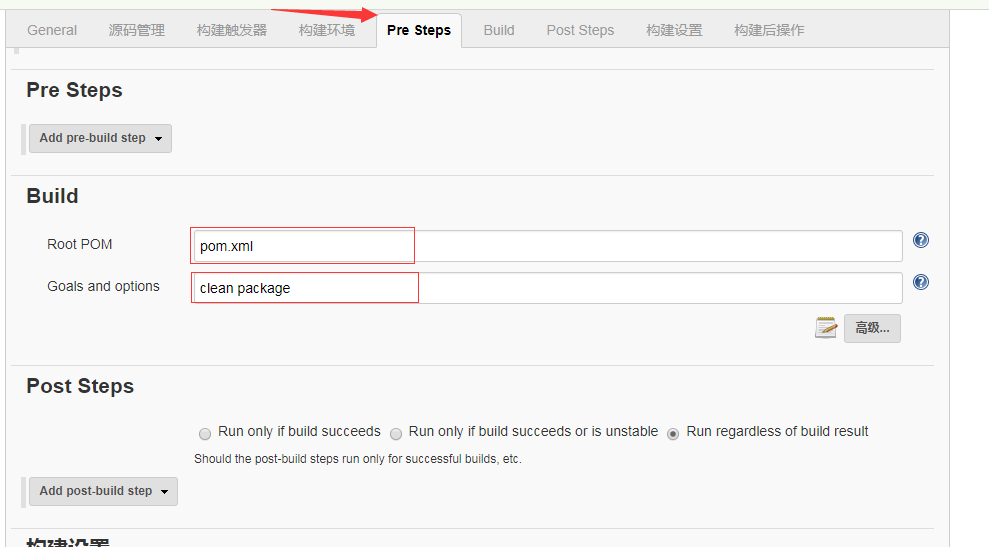
Remove prefix: target/

Remote directory: home/jenkins/${JOB\_NAME}/${BUILD\_NUMBER}

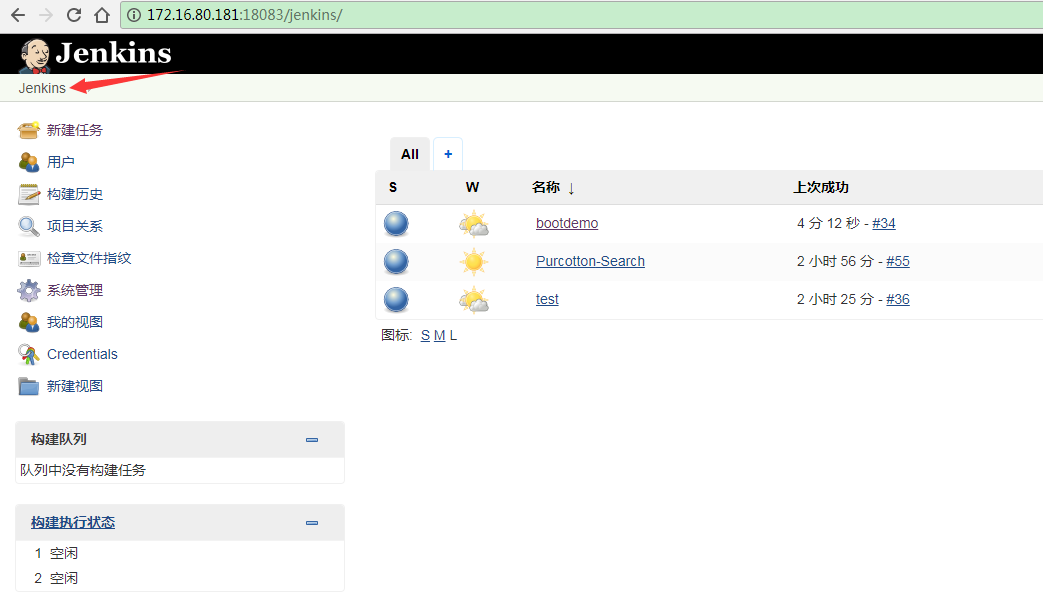
Exec command：

|  |
| --- |
| #切换目录  cd home/jenkins/${JOB\_NAME}/${BUILD\_NUMBER}/docker  #重建镜像  docker-compose build  #删除容器  docker-compose down  #启动容器  docker-compose up -d  #延迟下在查询日志  sleep 3  #监控日志  docker-compose logs |

1. 切换“Pre Steps”页签，添加maven构建参数



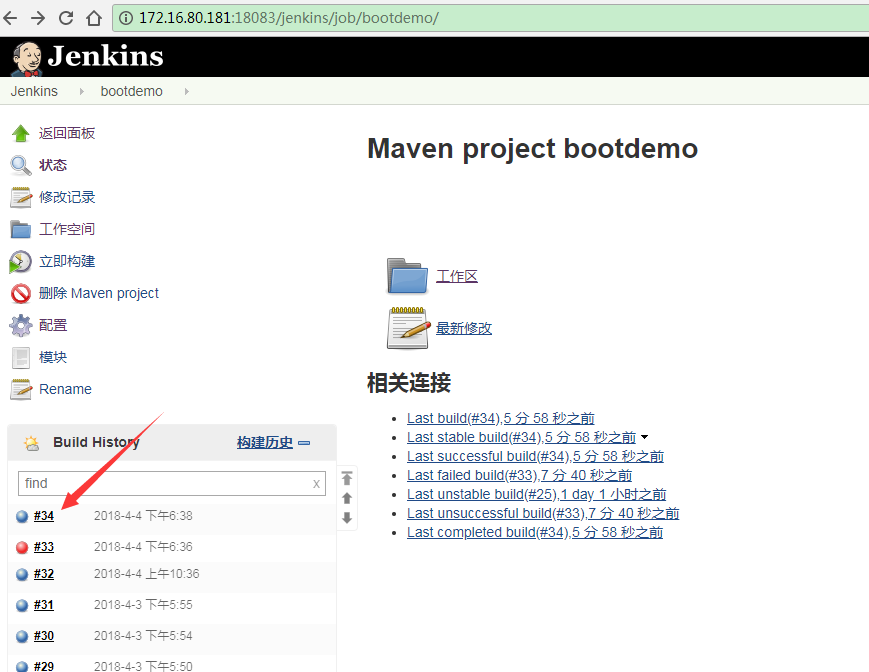
1. 进入首页，查看任务

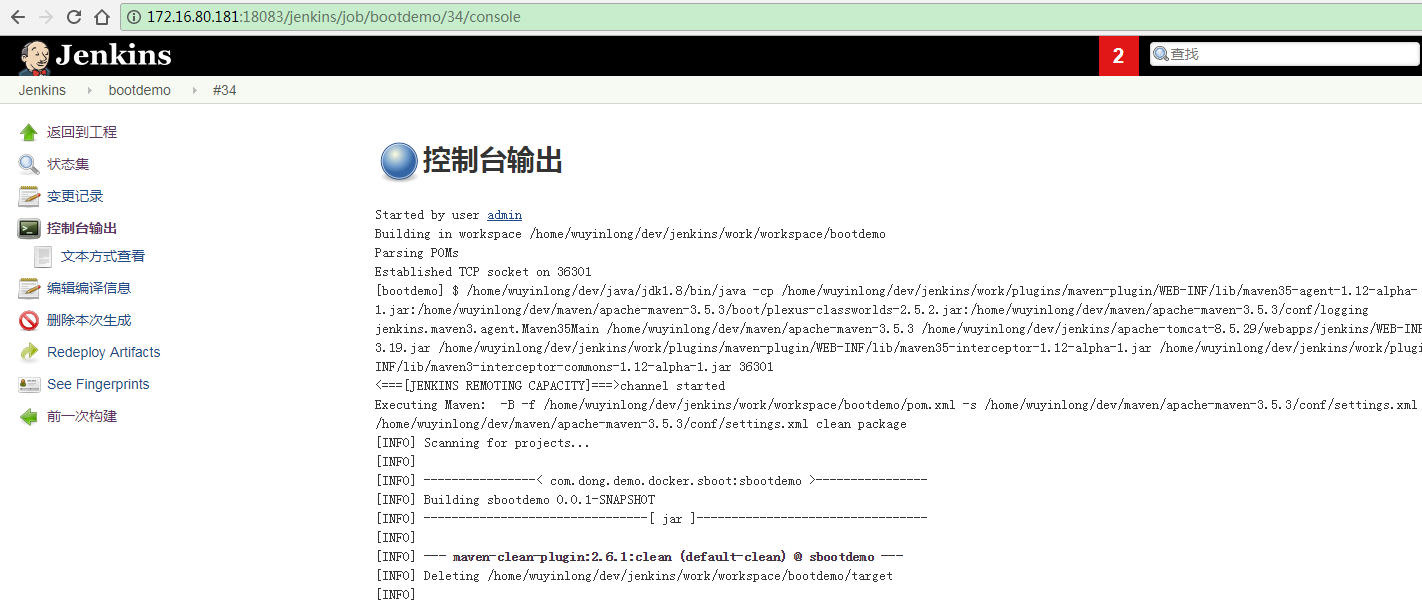


点击构建，执行任务



1. 进入任务查看控制台输出

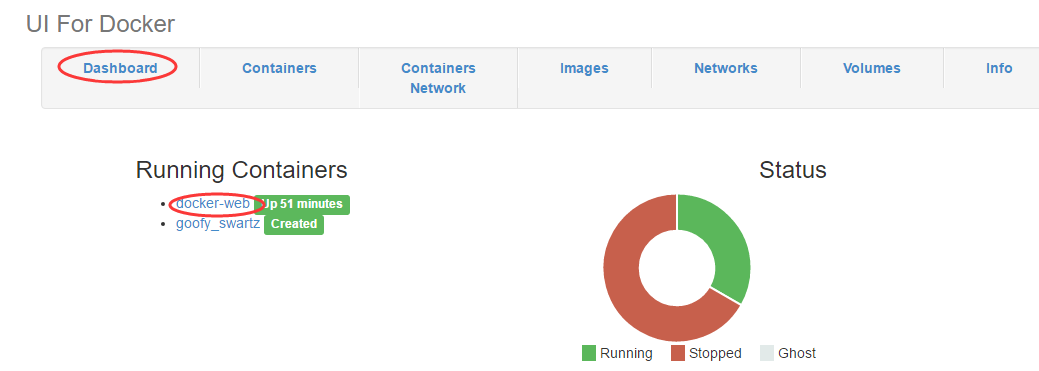


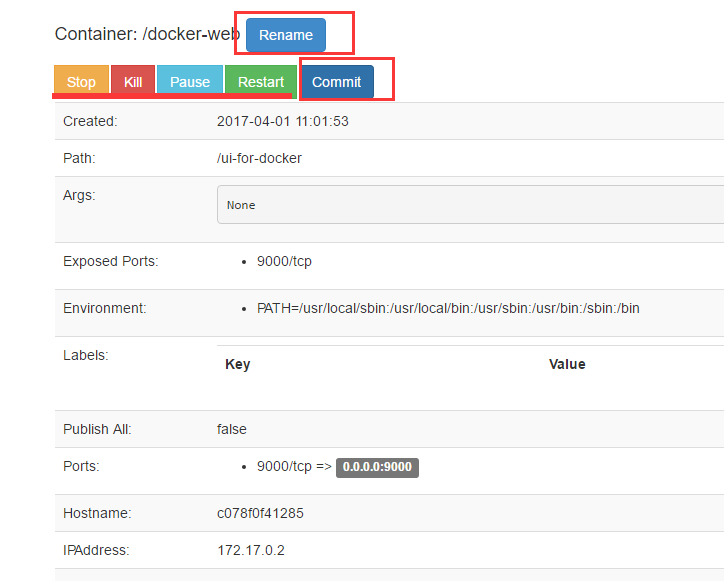


### Docker界面管理

#### Dashboard控制台

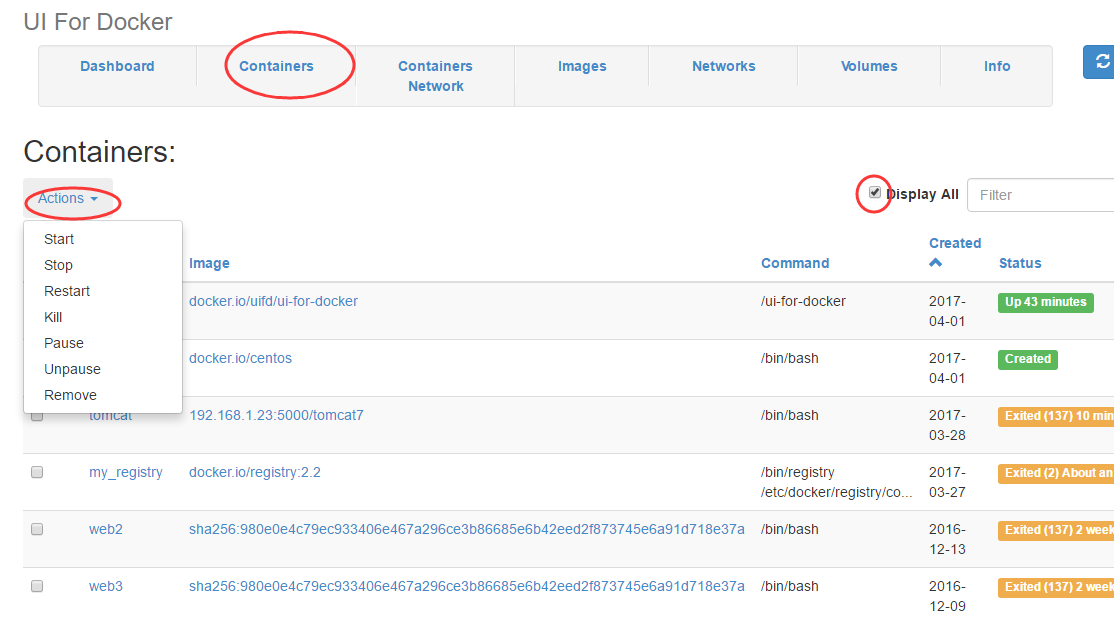
**点击Running Containers下面活跃的容器**，进入容器的管理界面进行相关操作，比如修改容器名，commit提交容器为新的镜像等。





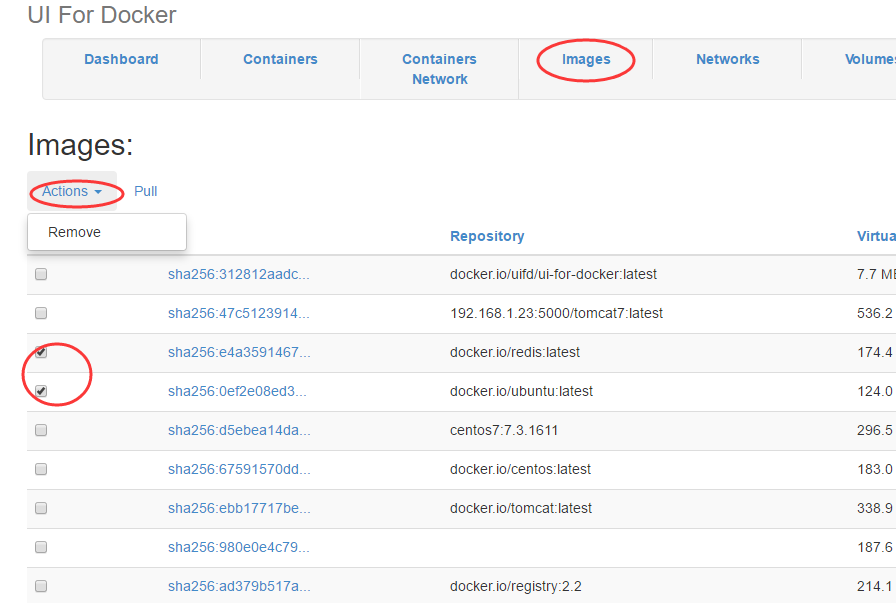
#### container容器管理

**点击Display All** ，可以显示所有创建了的容器，包括没有启动的。然后**点击Action**，可以对容器进行启动，关闭，重启，删除，挂起等操作。



#### images镜像管理

**点击Action**，可以对已有的镜像镜像移除操作。**点击Pull**，可以拉取镜像。**点击镜像ID**进去后可以添加或移除镜像tag



如下截图，Pull镜像的时候，Registry为空，默认从docker hub上拉取镜像。

# FAQ

1. 根据网上很多案例安装Docker，但是都无法成功安装

**解决方法：**通过<https://opsx.alibaba.com/mirror>平台下载docker安装包，进行安装

1. 启动docker容器报错：System error: exec: "/opt/docker/tomcat8/bin/catalina.sh": permission denied

**解决方法：**文件没有执行权限，通过命令执行权限chmod –R 777 catalina.sh

1. 启动docker容器警告：Usage of loopback devices is strongly discouraged for production use. Either use `--storage-opt dm.thinpooldev` or use `--storage-opt dm.no\_warn\_on\_loop\_devices=true` to suppress this warning.

解决方法：因为目前docker支持的存储驱动类型，有aufs，Device mapper，btrfs，overlayfs和zfs，所有驱动都用到写时复制（CoW）的技术。而出现警告的原因，就是在centos系统上（默认不支持aufs），所以暂时不用管

1. Maven pom文件错误'plugin' (position: START\_TAG seen ...<build>\r\n\t\t<plugin>...

**解决方法：**plugin要放到plugins标签里面

1. Connect to localhost:2375 [localhost/127.0.0.1, localhost/0:0:0:0:0:0:0:1] failed:

解决方法：配置maven运行环境变量DOCKER\_HOST指向远程docker地址

1. Exception caught: Must specify baseImage if dockerDirectory is null -> [Help 1]

解决方法： 需要指定baseImage标签

1. 打包错误，Exception caught: error pulling image configuration: Get https://dseasb33srnrn.cloudfront.net/registry-v2/

解决方法：使用阿里云镜像加速服务

1. Maven打包报错Request error: POST http://192.168.8.130:5257/build?t=demo/sbootDemo2: 500

解决方法：先停用docker服务，通过直接运行dockerd调试，找到日志为Got an API for which error does not match any expected type!!!: invalid reference format: repository name must be lowercase error\_type="\*errors.errorString" module=api，原来是仓库名称有问题

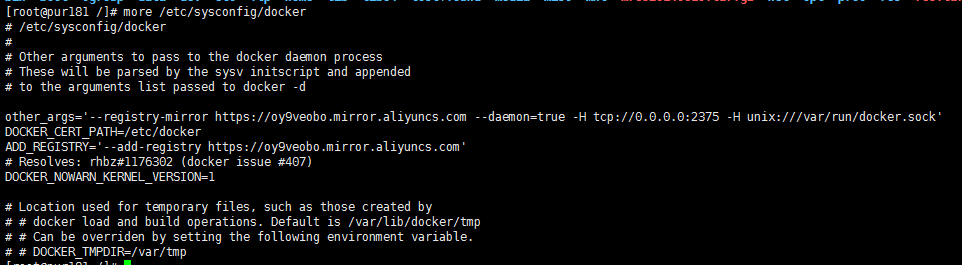
1. 容器启动报错: no main manifest attribute, in /sbootdemo2-0.0.1-SNAPSHOT.jar

解决方法：maven打包有误，增加spring boot打包插件

|  |
| --- |
| <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin> |

1. docker pull 好慢

解决方法：从 17.04 以后，Docker 的源的结构以及包名都进行了调整，17.04之前的版本配置镜像加速方式是修改/etc/sysconfig/docker文件，other\_args='--registry-mirror <https://oy9veobo.mirror.aliyuncs.com>'



1. 修改了 docker 服务配置后不起作用

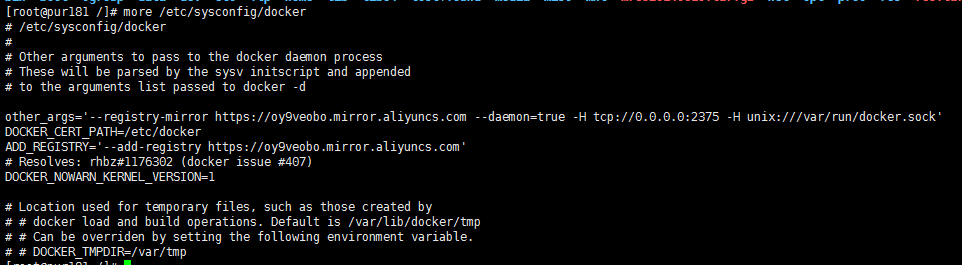
解决方法：在宿主上运行一下 ps -ef | grep dockerd 看看，通过docker –d启动调试查看具体原因

1. push私有仓库失败

解决方法：因为Docker从1.3.X之后，与docker registry交互默认使用的是https，而此处搭建的私有仓库只提供http服务，需要配置--insecure-registry=node01:5000参数

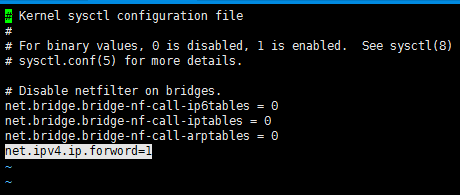
1. 配置守护进程失败

解决方法：17.1配置守护进程的需要/etc/sysconfig/docker文件中修改参数



1. IPv4 forwarding is disabled. Networking will not work.

解决方法：修改vi /usr/lib/sysctl.d/00-system.conf文件，增加“net.ipv4.ip\_forward=1”的属性，临时修改办法：sysctl -w net.ipv4.ip\_forward=1



然后重启网络systemctl restart network

1. ERROR: yaml.scanner.ScannerError: mapping values are not allowed here in "./docker-compose.yml", line 2, column 9

解决方法：yml语法要求比较高，属性冒号的后面一定要加空格

1. ERROR: yaml.scanner.ScannerError: while scanning for the next tokenfound character '\t' that cannot start any token

解决方法：不能通过\t去标识父子关系，要通过两个空格表示

1. services.sbootdemo2-0.0.1-SNAPSHOT.ports contains an invalid type, it should be an array

解决方法：数据格式使用前面加”- ”，多条数据使用换行，并且前面需要对齐

1. The Compose file './docker/docker-compose.yml' is invalid because:Unsupported config option for services.build: 'dockerfile'

解决方法：service没有build属性，需要放到service具体实例下面，如service.demo.build

1. Docker下面为什么会产生< none>:< none>的镜像

解决方法：< none>:< none> 镜像是dangling images（悬挂镜像），这种类型的镜像会产生磁盘空间占用问题，类似于java的内存垃圾，由于docker没有这种自动回收垃圾的机制，所以需要通过以下命令去清理docker rmi $(docker images -f "dangling=true" -q)

1. docker中 CMD 与 ENTRYPOINT 的区别.

解决方法：ENTRYPOINT是容器内部指定的参数，在docker命名命令启动的时候不会被覆盖，容器启动时最开始启动的，CMD则是docker的命令，命名会被docker指定的参数覆盖掉

ENTRYPOINT指令，往往用于设置容器启动后的第一个命令，这对一个容器来说往往是固定的。

CMD指令，往往用于设置容器启动的第一个命令的默认参数，这对一个容器来说可以是变化的。

docker run <command>往往用于给出替换CMD的临时参数

# 附录

## Docker常用命令:

sudo docker run -i -t ubuntu /bin/bash

-i 标志保证容器中的STDIN是开启的

-t 标志告诉Docker为要创建的容器分配一个伪tty终端

ubuntu 表示我们创建容器使用的镜像

/bin/bash 表示当容器创建完成之后，Docker就会执行容器中的/bin/bash命令

docker run --name my\_container -i -t ubuntu /bin/bash

--name为容器指定一个名称，使用指定的容器名称比使用容器ID更方便。

#使用容器ID启动容器

docker start f5a9f05f4214

#使用容器名称启动容器

docker start my\_container

docker stop my\_container

docker kill my\_container

#删除所有容器

docker rm `docker ps -a -q`

`docker ps`命令会列出现有的全部正在运行的容器信息

-a 标志代表列出所有容器，包括运行的和已经停止的

-q 标志表示只需要返回容器的ID而不会返回容器的其他信息

#删除指定容器

docker rm my\_container

如果容器已经不再使用，可以使用`docker rm`命令来删除他们，也可以通过给`docker rm`传递-f标志来删除运行中的Docker容器（Docker 1.6.2+）。

#查看容器信息

docker ps

#通过docker inspect来获得更详细的容器信息

docker inspect my\_container

# ﻿从主机复制到容器

docker cp host\_path containerID:container\_path

反之

从容器复制到主机

docker cp containerID:container\_path host\_path

删除失效镜像

docker volume rm $(docker volume ls -qf dangling=true)

## Dockerfile指令

|  |
| --- |
| #第一行必须指令基于的基础镜像  From ubutu  #维护者信息  MAINTAINER docker\_user docker\_user@mail.com  #镜像的操作指令  apt/sourcelist.list  RUN apt-get update && apt-get install -y ngnix  RUN echo "\ndaemon off;">>/etc/ngnix/nignix.conf  #容器启动时执行指令  CMD /usr/sbin/ngnix |

1、From指令   
From 或者From :

DockerFile第一条必须为From指令。如果同一个DockerFile创建多个镜像时，可使用多个From指令（每个镜像一次）

2、MAINTAINER   
格式为maintainer ，指定维护者的信息

3、RUN   
格式为Run 或者Run [“executable” ,”Param1”, “param2”]   
前者在shell终端上运行，即/bin/sh -C，后者使用exec运行。例如：RUN [“/bin/bash”, “-c”,”echo hello”]   
每条run指令在当前基础镜像执行，并且提交新镜像。当命令比较长时，可以使用“/”换行。

4、CMD指令   
支持三种格式：   
CMD [“executable” ,”Param1”, “param2”]使用exec执行，推荐   
CMD command param1 param2，在/bin/sh上执行   
CMD [“Param1”, “param2”] 提供给ENTRYPOINT做默认参数。

每个容器只能执行一条CMD命令，多个CMD命令时，只最后一条被执行。

5、EXPOSE

格式为 EXPOSE […] 。

告诉Docker服务端容器暴露的端口号，供互联系统使用。在启动Docker时，可以通过-P,主机会自动分配一个端口号转发到指定的端口。使用-P，则可以具体指定哪个本地端口映射过来

例如：   
EXPOSE 22 80 8443

6、ENV

格式为 ENV 。 指定一个环境变量，会被后续 RUN 指令使用，并在容器运行时保持。

例如

ENV PG\_MAJOR 9.3

ENV PG\_VERSION 9.3.4

RUN curl -SL http://example.com/postgres-$PG\_VERSION.tar.xz | tar -xJC /usr/src/postgress && …

ENV PATH /usr/local/postgres-$PG\_MAJOR/bin:$PATH

7、ADD   
格式为 ADD 。

该命令将复制指定的 到容器中的 。 其中 可以是Dockerfile所在目录的一个相对路径；也可以是一个URL；还可以是一个tar文件（自动解压为目录）。则。

8、COPY

格式为 COPY 。

复制本地主机的 （为Dockerfile所在目录的相对路径）到容器中的 。

当使用本地目录为源目录时，推荐使用 COPY 。

9、ENTRYPOINT

两种格式：

ENTRYPOINT [“executable”, “param1”, “param2”]   
ENTRYPOINT command param1 param2 （shell中执行）。   
配置容器启动后执行的命令，并且不可被 docker run 提供的参数覆盖。

每个Dockerfile中只能有一个 ENTRYPOINT ，当指定多个时，只有最后一个起效。

10、VOLUME

格式为 VOLUME [“/data”] 。

创建一个可以从本地主机或其他容器挂载的挂载点，一般用来存放数据库和需要保持的数据等。

11、USER

格式为 USER daemon 。

指定运行容器时的用户名或UID，后续的 RUN 也会使用指定用户。

当服务不需要管理员权限时，可以通过该命令指定运行用户。并且可以在之前创建所需要的用户，例如： RUN groupadd -r postgres && useradd -r -g postgres postgres 。要临时获取管理员权限可以使用 gosu ，而不推荐 sudo 。

12、WORKDIR

格式为 WORKDIR /path/to/workdir 。

为后续的 RUN 、 CMD 、 ENTRYPOINT 指令配置工作目录。

可以使用多个 WORKDIR 指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。例如

WORKDIR /a   
WORKDIR b   
WORKDIR c   
RUN pwd   
则最终路径为 /a/b/c 。

13、ONBUILD

格式为 ONBUILD [INSTRUCTION] 。

配置当所创建的镜像作为其它新创建镜像的基础镜像时，所执行的操作指令。

例如，Dockerfile使用如下的内容创建了镜像 image-A 。

[…]   
ONBUILD ADD . /app/src   
ONBUILD RUN /usr/local/bin/python-build –dir /app/src   
[…]   
如果基于A创建新的镜像时，新的Dockerfile中使用 FROM image-A 指定基础镜像时，会自动执行 ONBUILD 指令内容，等价于在后面添加了两条指令。

FROM image-A

#Automatically run the following

ADD . /app/src

RUN /usr/local/bin/python-build --dir /app/src

使用 ONBUILD 指令的镜像，推荐在标签中注明，例如 ruby:1.9-onbuild 。

## Docker Compose YML语法

默认的模板文件是 docker-compose.yml，其中定义的每个服务都必须通过 image 指令指定镜像或 build 指令（需要 Dockerfile）来自动构建。  
其它大部分指令都跟 docker run 中的类似。  
如果使用 build 指令，在 Dockerfile 中设置的选项(例如：CMD, EXPOSE, VOLUME, ENV 等) 将会自动被获取，无需在 docker-compose.yml 中再次设置。  
image  
指定为镜像名称或镜像 ID。如果镜像在本地不存在，Compose 将会尝试拉去这个镜像。  
例如：

image: ubuntu

image: orchardup/postgresql

image: a4bc65fd

build

指定 Dockerfile 所在文件夹的路径。 Compose 将会利用它自动构建这个镜像，然后使用这个镜像。  
build: /path/to/build/dir

**command**

覆盖容器启动后默认执行的命令。  
command: bundle exec thin -p 3000

**links**

链接到其它服务中的容器。使用服务名称（同时作为别名）或服务名称：服务别名 （[SERVICE:ALIAS](https://blog.csdn.net/u014242496/article/details/73835541)） 格式都可以。  
links:  
- db  
- db:database  
- redis  
使用的别名将会自动在服务容器中的 /etc/hosts 里创建。例如：  
172.17.2.186 db  
相应的环境变量也将被创建。

**external\_links**

链接到 docker-compose.yml 外部的容器，甚至 并非 Compose 管理的容器。参数格式跟 links 类似。  
external\_links:  
- redis\_1  
- project\_db\_1:mysql  
- project\_db\_1:postgresql

ports

暴露端口信息。  
使用宿主：容器 （HOST:CONTAINER）格式或者仅仅指定容器的端口（宿主将会随机选择端口）都可以。

ports:

- "3000"  
- "8000:8000"  
- "127.0.0.1:8001:8001"  
注：当使用 HOST:CONTAINER 格式来映射端口时，如果你使用的容器端口小于 60 你可能会得到错误得结果，因为 YAML 将会解析 xx:yy 这种数字格式为 60 进制。所以建议采用字符串格式。

**expose**

暴露端口，但不映射到宿主机，只被连接的服务访问。  
仅可以指定内部端口为参数  
expose:  
- "3000"  
- "8000"

volumes

卷挂载路径设置。可以设置宿主机路径 （HOST:CONTAINER） 或加上访问模式 （HOST:CONTAINER:ro）。  
volumes:  
- /var/lib/mysql  
- cache/:/tmp/cache  
- ~/configs:/etc/configs/:ro

volumes\_from

从另一个服务或容器挂载它的所有卷。  
volumes\_from:  
- service\_name  
- container\_name

environment

设置环境变量。你可以使用数组或字典两种格式。  
只给定名称的变量会自动获取它在 Compose 主机上的值，可以用来防止泄露不必要的数据。  
environment:  
- RACK\_ENV=development  
- SESSION\_SECRET

**env\_file**

从文件中获取环境变量，可以为单独的文件路径或列表。  
如果通过 docker-compose -f FILE 指定了模板文件，则 env\_file 中路径会基于模板文件路径。  
如果有变量名称与 environment 指令冲突，则以后者为准。  
env\_file: .env  
env\_file:  
- ./common.env  
- ./apps/web.env  
- /opt/secrets.env  
环境变量文件中每一行必须符合格式，支持 # 开头的注释行。  
# common.env: Set Rails/Rack environment  
RACK\_ENV=development

extends

基于已有的服务进行扩展。例如我们已经有了一个 webapp 服务，模板文件为 common.yml。  
# common.yml  
webapp:  
build: ./webapp  
environment:  
\ - DEBUG=false  
\ - SEND\_EMAILS=false  
编写一个新的 development.yml 文件，使用 common.yml 中的 webapp 服务进行扩展。