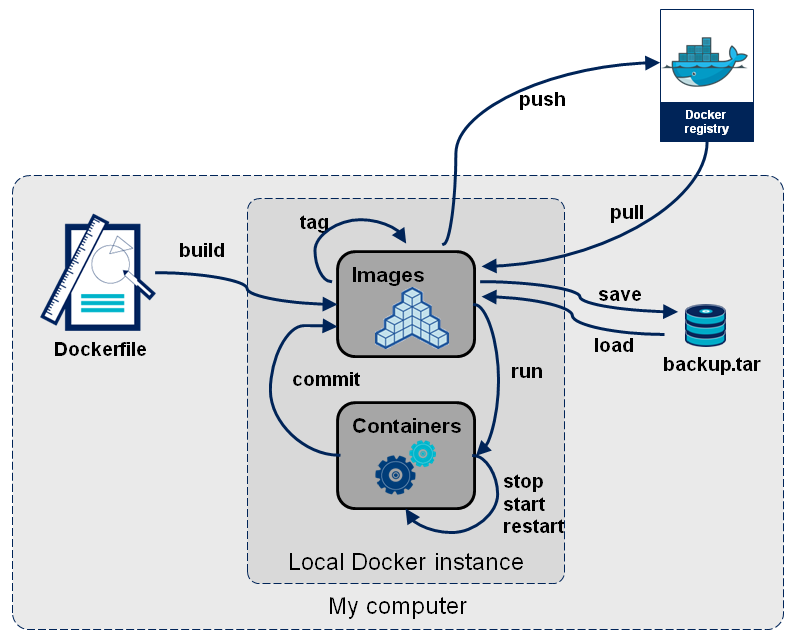
## 一．Docker常用命令



### 1. 查看docker信息（version、info）

1. # 查看docker版本
2. $docker version
4. # 显示docker系统的信息
5. $docker info

### 2. 对image的操作（search、pull、images、rmi、history）

1. # 检索image
2. $docker search image\_name
4. # 下载image
5. $docker pull image\_name
7. # 列出镜像列表; -a, --all=false Show all images; --no-trunc=false Don't truncate output; -q, --quiet=false Only show numeric IDs
8. $docker images
10. # 删除一个或者多个镜像; -f, --force=false Force; --no-prune=false Do not delete untagged parents
11. $docker rmi image\_name
13. # 显示一个镜像的历史; --no-trunc=false Don't truncate output; -q, --quiet=false Only show numeric IDs
14. $docker history image\_name

### 3. 启动容器（run）

docker容器可以理解为在沙盒中运行的进程。这个沙盒包含了该进程运行所必须的资源，包括文件系统、系统类库、shell 环境等等。但这个沙盒默认是不会运行任何程序的。你需要在沙盒中运行一个进程来启动某一个容器。这个进程是该容器的唯一进程，所以当该进程结束的时候，容器也会完全的停止。

1. # 在容器中运行"echo"命令，输出"hello word"
2. $docker run image\_name echo "hello word"
4. # 交互式进入容器中
5. $docker run -i -t image\_name /bin/bash

8. # 在容器中安装新的程序
9. $docker run image\_name apt-get install -y app\_name

### 4. 查看容器（ps）

1. # 列出当前所有正在运行的container
2. $docker ps
3. # 列出所有的container
4. $docker ps -a
5. # 列出最近一次启动的container
6. $docker ps -l

### 5. 保存对容器的修改（commit）

当你对某一个容器做了修改之后（通过在容器中运行某一个命令），可以把对容器的修改保存下来，这样下次可以从保存后的最新状态运行该容器。

1. # 保存对容器的修改; -a, --author="" Author; -m, --message="" Commit message
2. $docker commit ID new\_image\_name

Note：  image相当于类，container相当于实例，不过可以动态给实例安装新软件，然后把这个container用commit命令固化成一个image。

### 6. 对容器的操作（rm、stop、start、kill、logs、diff、top、cp、restart、attach）

1. # 删除所有容器
2. $docker rm `docker ps -a -q`
4. # 删除单个容器; -f, --force=false; -l, --link=false Remove the specified link and not the underlying container; -v, --volumes=false Remove the volumes associated to the container
5. $docker rm Name/ID
7. # 停止、启动、杀死一个容器
8. $docker stop Name/ID
9. $docker start Name/ID
10. $docker kill Name/ID
12. # 从一个容器中取日志; -f, --follow=false Follow log output; -t, --timestamps=false Show timestamps
13. $docker logs Name/ID
15. # 列出一个容器里面被改变的文件或者目录，list列表会显示出三种事件，A 增加的，D 删除的，C 被改变的
16. $docker diff Name/ID
18. # 显示一个运行的容器里面的进程信息
19. $docker top Name/ID
21. # 从容器里面拷贝文件/目录到本地一个路径
22. $docker cp Name:/container\_path to\_path
23. $docker cp ID:/container\_path to\_path
25. # 重启一个正在运行的容器; -t, --time=10 Number of seconds to try to stop for before killing the container, Default=10
26. $docker restart Name/ID
28. # 附加到一个运行的容器上面; --no-stdin=false Do not attach stdin; --sig-proxy=true Proxify all received signal to the process
29. $docker attach ID

Note： attach命令允许你查看或者影响一个运行的容器。你可以在同一时间attach同一个容器。你也可以从一个容器中脱离出来，是从CTRL-C。

### 7. 保存和加载镜像（save、load）

当需要把一台机器上的镜像迁移到另一台机器的时候，需要保存镜像与加载镜像。

1. # 保存镜像到一个tar包; -o, --output="" Write to an file
2. $docker save image\_name -o file\_path
3. # 加载一个tar包格式的镜像; -i, --input="" Read from a tar archive file
4. $docker load -i file\_path
6. # 机器a
7. $docker save image\_name > /home/save.tar
8. # 使用scp将save.tar拷到机器b上，然后：
9. $docker load < /home/save.tar

### 8、 登录registry server（login）

1. # 登陆registry server; -e, --email="" Email; -p, --password="" Password; -u, --username="" Username
2. $docker login

### 9. 发布image（push）

1. # 发布docker镜像
2. $docker push new\_image\_name

### 10.根据Dockerfile 构建出一个容器

1. #build
2. --no-cache=false Do not use cache when building the image
3. -q, --quiet=false Suppress the verbose output generated by the containers
4. --rm=true Remove intermediate containers after a successful build
5. -t, --tag="" Repository name (and optionally a tag) to be applied to the resulting image in case of success
6. $docker build -t image\_name Dockerfile\_path

1. 列出机器上的镜像（images）

docker images

2. 在docker index中搜索image（search）

Usage: docker search TERM

docker search seanlo

3. 从docker registry server 中下拉image或repository（pull）

docker pull [OPTIONS] NAME[:TAG]

docker pull centos

4. 推送一个image或repository到registry（push）

与上面的pull对应，可以推送到Docker Hub的Public、Private以及私服，但不能推送到Top Level Repository。

# docker push seanlook/mongo

# docker push registry.tp-link.net:5000/mongo:2014-10-27

5. 从image启动一个container（run）

docker run命令首先会从特定的image创之上create一层可写的Container，然后通过start命令来启动它。停止的container可以重新启动并保留原来的修改。run命令启动参数有很多，以下是一些常规使用说明，更多部分请参考http://www.cnphp6.com/archives/24899

当利用 docker run 来创建容器时，Docker 在后台运行的标准操作包括：

a.检查本地是否存在指定的镜像，不存在就从公有仓库下载

b.利用镜像创建并启动一个容器

c.分配一个文件系统，并在只读的镜像层外面挂载一层可读写层

d.从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去

e.从地址池配置一个 ip 地址给容器

f.执行用户指定的应用程序

g.执行完毕后容器被终止

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

docker run -d -p 127.0.0.1:33301:22 centos6-ssh

后台运行(-d)、并暴露端口(-p)

6. 将一个container固化为一个新的image（commit）

当我们在制作自己的镜像的时候，会在container中安装一些工具、修改配置，如果不做commit保存起来，那么container停止以后再启动，这些更改就消失了。

docker commit <container> [repo:tag]

后面的repo:tag可选

只能提交正在运行的container，即通过docker ps可以看见的容器，

7. 开启/停止/重启container（start/stop/restart）

容器可以通过run新建一个来运行，也可以重新start已经停止的container，但start不能够再指定容器启动时运行的指令，因为docker只能有一个前台进程。

8. 查看image或container的底层信息（inspect）

inspect的对象可以是image、运行中的container和停止的container。

9. 删除一个或多个container、image（rm、rmi）

你可能在使用过程中会build或commit许多镜像，无用的镜像需要删除。但删除这些镜像是有一些条件的：

同一个IMAGE ID可能会有多个TAG（可能还在不同的仓库），首先你要根据这些 image names 来删除标签，当删除最后一个tag的时候就会自动删除镜像；

承上，如果要删除的多个IMAGE NAME在同一个REPOSITORY，可以通过docker rmi <image\_id>来同时删除剩下的TAG；若在不同Repo则还是需要手动逐个删除TAG；

还存在由这个镜像启动的container时（即便已经停止），也无法删除镜像；

10. docker build 使用此配置生成新的image

build命令可以从Dockerfile和上下文来创建镜像：

docker build [OPTIONS] PATH | URL | -

docker build -t <镜像名> <Dockerfile路径>

11. 查看容器的信息container（ps）

docker ps命令可以查看容器的CONTAINER ID、NAME、IMAGE NAME、端口开启及绑定、容器启动后执行的COMMNAD。经常通过ps来找到CONTAINER\_ID。

docker ps 默认显示当前正在运行中的container

docker ps -a 查看包括已经停止的所有容器

docker ps -l 显示最新启动的一个容器（包括已停止的）

12. 其他命令

docker还有一些如login、cp、logs、export、import、load、kill等不是很常用的命令，比较简单。

查看容器日志

docker logs -f <容器名orID>

杀死一个容器

docker kill <容器名orID>

从Container中拷贝文件出来

docker cp 7bb0e258aefe:/etc/debian\_version .

拷贝7bb0e258aefe中的/etc/debian\_version到当前目录下。

容器命令

**查看容器信息（docker ps ）：**

[root@testhadoop-slave1 qinys]# docker ps   
CONTAINER ID        IMAGE            COMMAND    CREATED      STATUS       PORTS    NAMES  
284953cde84b        centos/qinys     /bin/bash  8 days ago   Up 8 days             test\_container    
参数： -a  
  
**要获取容器的输出信息(docker logs):**  
docker logs container\_name  
例如： docker logs test\_container

**停止容器(docker stop):**

dockerstop container\_name

**重新启动处于终止状态的容器(docker start):**

docker  start container\_name

**导出容器快照到本地文件(docker export)：**  
首先获取容器id：  
docker ps -a  
导出容器到本地镜像库：  
docker export container\_id > centos.tar  
  
**导入容器快照为镜像(docker import)：**  
(1)容器在本地：  
cat centos.tar | docker import - registry.intra.weibo.com/yushuang3/centos:v2.0  
(2)容器在网络上：  
docker import http://example.com/exampleimage.tgz registry.intra.weibo.com/yushuang3/centos:v2.0  
注意：  
用户既可以使用 docker load 来导入镜像存储文件到本地镜像库，  
也可以使用 docker import 来导入一个容器快照到本地镜像库。  
这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息（即仅保存容器当时的快照状态），  
而镜像存储文件将保存完整记录，体积也要大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。  
  
**删除容器(docker rm)：**  
docker ps -a 获取容器name  
docker rm container\_name

## 二、镜像加速器

Docker Hub 在国外，有时候拉取 Image 极其缓慢，可以使用国内的镜像来实现加速

阿里云

echo "DOCKER\_OPTS=\"--registry-mirror=https://yourlocation.mirror.aliyuncs.com\"" | sudo tee -a /etc/default/docker

sudo service docker restart

其中 https://yourlocation.mirror.aliyuncs.com 是您在阿里云注册后的专属加速器地址：

DaoCloud

sudo echo “DOCKER\_OPTS=\”\$DOCKER\_OPTS –registry-mirror=http://your-id.m.daocloud.io -d\”” >> /etc/default/docker

sudo service docker restart

其中 http://your-id.m.daocloud.io 是您在 DaoCloud 注册后的专属加速器地址：

Windows 10

对于使用 WINDOWS 10 的系统，在系统右下角托盘图标内右键菜单选择

Settings ，打开配置窗口后左侧导航菜单选择 Docker Daemon 。编辑窗口内

的JSON串，填写如阿里云、DaoCloud之类的加速器地址，如：

{

"registry-mirrors": [

"https://zmwa1utj.mirror.aliyuncs.com"

],

"insecure-registries": []

}

编辑完成，点击Apply保存后Docker服务会重新启动。

## 三、Dockerfile常用指令

指令的一般格式为指令名称 参数 。

### 1.FROM

支持三种格式：

FROM <image>

FROM <image>:<tag>

FROM <image>@<digest>

FROM指令必须指定且需要在Dockerfile其他指令的前面，指定的基础image可以是官方远程仓库中的，也可以位于本地仓库。后续的指令都依赖于该指令指定的image。当在同一个Dockerfile中建立多个镜像时，可以使用多个FROM指令。

### 2.MAINTAINER

格式为：

MAINTAINER <name>

用于指定维护者的信息。

### 3.RUN

支持两种格式：

RUN <command>

或 RUN ["executable", "param1", "param2"]。

RUN <command> 在shell终端中运行命令，在Linux中默认是/bin/sh -c 在Windows中是 cmd /s /c  
RUN ["executable", "param1", "param2"] 使用exec执行。指定其他终端可以通过该方式操作，例如：RUN ["/bin/bash", "-c", "echo hello"] ，该方式必须使用["]而不能使用[']，因为该方式会被转换成一个JSON 数组。

### 4.CMD

支持三种格式：

CMD ["executable","param1","param2"] (推荐使用)

CMD ["param1","param2"] (为ENTRYPOINT指令提供预设参数)

CMD command param1 param2 (在shell中执行)

CMD指令的主要目的是为执行容器提供默认值。每个Dockerfile只有一个CMD命令，如果指定了多个CMD命令，那么只有一条会被执行，如果启动容器的时候指定了运行的命令，则会覆盖掉CMD指定的命令。

### 5.LABEL

格式为：

LABEL <key>=<value> <key>=<value> <key>=<value> ...

为镜像添加元数据。使用 "和 \ 转换命令行，示例：

LABEL "com.example.vendor"="ACME Incorporated"

LABEL com.example.label-with-value="foo"

LABEL version="1.0"

LABEL description="This text illustrates \

that label-values can span multiple lines."

### 6.EXPOSE

格式为：

EXPOSE <port> [<port>...]

为Docker容器设置对外的端口号。在启动时，可以使用-p选项或者-P选项。

示例：

# 映射一个端口示例EXPOSE port1# 相应的运行容器使用的命令

docker run -p port1 image# 也可以使用-P选项启动

docker run -P image

# 映射多个端口示例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# 还可以指定需要映射到宿主机器上的某个端口号

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

### 7.ENV

格式为：

ENV <key> <value>

ENV <key>=<value> ...

指定环境变量，会被后续RUN指令使用，并在容器启动后，可以通过docker inspect查看这个环境变量，也可以通过docker run --env <key>=<value> 来修改环境变量

示例：

ENV JAVA\_HOME /path/to/java # 设置环境变量JAVA\_HOME

### 8.ADD

格式为：

ADD <src>... <dest>

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，也可以是一个URL，还可以是一个压缩包

注意：

src必须在构建的上下文内，不能使用例如：ADD ../somethine /something ，因为docker build 命令首先会将上下文路径和其子目录发送到docker daemon

如果src是一个URL，同时dest不以斜杠结尾，dest将会被视为文件，src对应内容文件将会被下载到dest

如果src是一个URL，同时dest以斜杠结尾，dest将被视为目录，src对应内容将会被下载到dest目录

如果src是一个目录，那么整个目录其下的内容将会被拷贝，包括文件系统元数据

如果文件是可识别的压缩包格式，则docker会自动解压

### 9.COPY

格式为：

COPY <src>... <dest>

COPY ["<src>",... "<dest>"] （shell中执行）

复制本地端的src到容器的dest。和ADD指令类似，COPY不支持URL和压缩包。

### 10.ENTRYPOINT

格式为：

ENTRYPOINT ["executable", "param1", "param2"]

ENTRYPOINT command param1 param2

指定Docker容器启动时执行的命令，可以多次设置，但是只有最后一个有效。

### 11.VOLUME

格式为：

VOLUME ["/data"]

使容器中的一个目录具有持久化存储数据的功能，该目录可以被容器本身使用，也可以共享给其他容器。当容器中的应用有持久化数据的需求时可以在Dockerfile中使用该指令。

### 12.USER

格式为：

USER 用户名

设置启动容器的用户，默认是root用户。

### 13.WORKDIR

格式为：

WORKDIR /path/to/workdir

切换目录指令，类似于cd命令，对RUN、CMD、ENTRYPOINT生效。

### 14.ARG

格式为：

ARG <name>[=<default value>]

ARG指令定义一个变量。

### 15.ONBUILD

格式为：

ONBUILD [INSTRUCTION]

指定当建立的镜像作为其他镜像的基础时，所执行的命令。

### 16.其他

STOPSINGAL  
HEALTHCHECK  
SHELL

### 17.使用Dockerfile构建Docker镜像

1.创建文件，命名为Dockerfile

# 基于哪个镜像

FROM java:8

# 将本地文件夹挂载到当前容器

VOLUME /tmp

# 拷贝文件到容器，也可以直接写成ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar #/app.jar

ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jarRUN bash -c 'touch /app.jar'

# 开放8761端口

EXPOSE 8761

# 配置容器启动后执行的命令

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

构建docker镜像，执行：

docker build -t eacdy/test1 .

# 格式：docker build -t 标签名称 Dockerfile的相对位置

构建成功：Successfully built a7cc6f4de088 。

启动镜像

docker run -p 8761:8761 eacdy/test1

访问http://Docker宿主机IP:8761 ，我们会发现Eureka能够正常被访问。

## 四、使用Maven插件构建Docker镜像

Docker maven插件有很多种比如如下的

|  |  |
| --- | --- |
| **插件名称** | **官方地址** |
| docker-maven-plugin | [https://github.com/spotify/docker-maven-plugin](https://github.com/spotify/docker-maven-plugin" \t "http://git.oschina.net/itmuch/spring-cloud-book/blob/master/3%20%E4%BD%BF%E7%94%A8Docker%E6%9E%84%E5%BB%BA%E5%BE%AE%E6%9C%8D%E5%8A%A1/_blank) |
| docker-maven-plugin | [https://github.com/fabric8io/docker-maven-plugin](https://github.com/fabric8io/docker-maven-plugin" \t "http://git.oschina.net/itmuch/spring-cloud-book/blob/master/3%20%E4%BD%BF%E7%94%A8Docker%E6%9E%84%E5%BB%BA%E5%BE%AE%E6%9C%8D%E5%8A%A1/_blank) |
| docker-maven-plugin | [https://github.com/bibryam/docker-maven-plugin](https://github.com/bibryam/docker-maven-plugin" \t "http://git.oschina.net/itmuch/spring-cloud-book/blob/master/3%20%E4%BD%BF%E7%94%A8Docker%E6%9E%84%E5%BB%BA%E5%BE%AE%E6%9C%8D%E5%8A%A1/_blank) |

我们以第一种为例子

1. 简单使用

在pom.xml中添加下面这段

<build>

<plugins>

<!-- docker的maven插件，官网：https://github.com/spotify/docker-maven-plugin -->

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.4.12</version>

<configuration>

<!-- 注意imageName一定要是符合正则[a-z0-9-\_.]的，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>microservice-discovery-eureka</imageName>

<baseImage>java</baseImage>

<entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

</resource>

</resources>

</configuration>

</plugin>

</plugins>

</build>

执行命令：

mvn clean package docker:build

1. Dockerfile

建立文件Dockerfile

FROM java:8

VOLUME /tmp

ADD microservice-discovery-eureka-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"]

修改pom.xml

<build>

<plugins>

<!-- docker的maven插件，官网：https://github.com/spotify/docker-maven-plugin -->

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.4.12</version>

<configuration>

<!-- 注意imageName一定要是符合正则[a-z0-9-\_.]的，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>microservice-discovery-eureka-dockerfile</imageName>

<!-- 指定Dockerfile所在的路径 -->

<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>

</plugins>

</build>

1. 将Docker镜像push到DockerHub上

首先修改Maven的全局配置文件settings.xml，添加以下段落

<servers>

<server>

<id>docker-hub</id>

<username>你的DockerHub用户名</username>

<password>你的DockerHub密码</password>

<configuration>

<email>你的DockerHub邮箱</email>

</configuration>

</server>

</servers>

在DockerHub上创建repo,例如：test，如下图

DockerHub

项目pom.xml修改为如下：注意imageName的路径要和repo的路径一致

<build>

<plugins>

<!-- docker的maven插件，官网：https://github.com/spotify/docker-maven-plugin -->

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.4.12</version>

<configuration>

<!-- 注意imageName一定要是符合正则[a-z0-9-\_.]的，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 -->

<!-- 如果要将docker镜像push到DockerHub上去的话，这边的路径要和repo路径一致 -->

<imageName>eacdy/test</imageName>

<!-- 指定Dockerfile所在的路径 -->

<dockerDirectory>${project.basedir}/src/main/docker</dockerDirectory>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

</resource>

</resources>

<!-- 以下两行是为了docker push到DockerHub使用的。 -->

<serverId>docker-hub</serverId>

<registryUrl>https://index.docker.io/v1/</registryUrl>

</configuration>

</plugin>

</plugins>

</build>

执行命令：

mvn clean package docker:build -DpushImage

搞定，等构建成功后，我们会发现Docker镜像已经被push到DockerHub上了。

4.将镜像push到私有仓库

在很多场景下，我们需要将镜像push到私有仓库中去，这边为了讲解的全面性，私有仓库采用的是配置登录认证的私有仓库。

和push镜像到DockerHub中一样，我们首先需要修改Maven的全局配置文件settings.xml，添加以下段落

<servers>

<server>

<id>docker-registry</id>

<username>你的DockerHub用户名</username>

<password>你的DockerHub密码</password>

<configuration>

<email>你的DockerHub邮箱</email>

</configuration>

</server>

</servers>

将项目的pom.xml改成如下，

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.4.12</version>

<configuration>

<!-- 路径为：私有仓库地址/你想要的镜像路径 -->

<imageName>reg.itmuch.com/test-pull-registry</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>

<!-- 与maven配置文件settings.xml一致 -->

<serverId>docker-registry</serverId>

</configuration>

</plugin>

执行：

mvn clean package docker:build -DpushImage

稍等片刻，将会push成功。

如果想要从私服上下载该镜像，执行：

docker login reg.itmuch.com # 然后输入账号和密码

docker pull reg.itmuch.com/test-pull-registry  
5.将插件绑定在某个phase执行

在很多场景下，我们有这样的需求，例如执行mvn clean package 时，自动地为我们构建docker镜像，可以吗？答案是肯定的。我们只需要将插件的goal 绑定在某个phase即可。

所谓的phase和goal，可以这样理解：maven命令格式是：mvn phase:goal ，例如mvn package docker:build 那么，package 和 docker 都是phase，build 则是goal 。

下面是示例：

首先配置属性：

<properties>

<docker.image.prefix>reg.itmuch.com</docker.image.prefix>

</properties>

插件配置：

<build>

<plugins>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<executions>

<execution>

<id>build-image</id>

<phase>package</phase>

<goals>

<goal>build</goal>

</goals>

</execution>

</executions>

<configuration>

<imageName>${docker.image.prefix}/${project.artifactId}</imageName>

<baseImage>java</baseImage>

<entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

</resource>

</resources>

</configuration>

</plugin>

</plugins>

</build>

如上，我们只需要添加：

<executions>

<execution>

<id>build-image</id>

<phase>package</phase>

<goals>

<goal>build</goal>

</goals>

</execution>

</executions>

即可。本例指的是讲docker的build目标，绑定在package这个phase上。也就是说，用户只需要执行mvn package ，就自动执行了mvn docker:build 。

附1：

FROM frolvlad/alpine-oraclejdk8:slim

VOLUME /tmp

ADD cachetest-0.0.1-SNAPSHOT.jar app.jar

RUN sh -c **'touch** /app.jar'

ENV JAVA\_OPTS=""

ENTRYPOINT [ "sh", "-c", "java $JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]

附2：

docker maven plugin是个简单的可以管理Docker容器maven插件，这个插件将会根据你的配置，在构建时启动容器，构建结束时停止容器并删除，如果本地找不到镜像，Docker会自动去中央仓库下载。

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.4.11</version>

<configuration>

<imageName>${docker.image.prefix}/${project.artifactId}</imageName>

<dockerDirectory>src/main/docker</dockerDirectory>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

</resource>

</resources>

</configuration>

</plugin>

## Docker Compose

Compose 是一个用于定义和运行多容器的Docker应用的工具。使用Compose，你可以在一个配置文件（yaml格式）中配置你应用的服务，然后使用一个命令，即可创建并启动配置中引用的所有服务。