**通过现有容器生成**（其实就是更新现有的镜像来创建新镜像）

启动一个容器

在容器中做一些操作，比如安装软件，更新一些东西，然后退出。

关键是这条命令 docker commit –m “update” –a=”author” 4cd691297f92 new\_image\_name

这条命令基于容器4cd691297f92 创建镜像new\_image\_name

**通过docker build来创建镜像。**需要一个Dockerfile——描述创建镜像的内容

Dockerfile描述

每一个指令都会在镜像上创建一个新的层，每一个指令的前缀都必须是大写的。

第一条FROM，指定使用哪个镜像源

RUN 指令告诉docker 在镜像内执行命令，安装了什么。。。

然后，我们使用 Dockerfile 文件，通过 docker build 命令来构建一个镜像。

|  |
| --- |
| FROM openjdk:8-jdk-alpine  ENV SPRING\_OUTPUT\_ANSI\_ENABLED=ALWAYS \  YINGXIN\_SLEEP=0 \  JAVA\_OPTS=""  VOLUME ["/usr/local/logs/xxx/"]  RUN echo "Asia/Shanghai" > /etc/timezone  RUN apk add --no-cache tini  # Add a yinghuo user to run our application so that it doesn't need to run as root  RUN adduser -D -s /bin/sh userxxx  WORKDIR /home/userxxx  ADD entrypoint.sh entrypoint.sh  RUN chmod 755 entrypoint.sh && chown userxxx:userxxx entrypoint.sh  USER userxxx  ADD target/\*.war app.war  ENTRYPOINT ["/sbin/tini", "./entrypoint.sh"]  EXPOSE 9098 |

#docker build –t image\_name dockerfile\_directory

至此镜像已经创建好了，如果需要推送的远程镜像仓库（私有库或者hub库），需要满足一定的要求。

**如果要推送到私有库，镜像名需要指定私有库的地址端口和用户镜像名**

例如 docker tag hello-world localhost:5000/hello-world

#docker tag image image:tag

根据需要使用

docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

**使用maven构建docker镜像**

很好的一篇介绍文章

https://www.jianshu.com/p/6717adc2ad94

在实施CI/CD的过程中，实现一键发布用的最多的工具就是Jenkins了，在Jenkins上通过配置将每一个步骤串联起来，现在出现了pipeline让这个过程更简单了，一般的持续集成的流程是：

1. 从代码仓库下载代码（git或者svn）；
2. 通过工具（maven或者gradle）编译出可执行程序包（jar或者war）；
3. 使用dockerfile配置build出docker镜像；
4. 将docker镜像发布至镜像仓库；
5. 将镜像部署到云平台；
6. 多环境分发镜像。

Docker镜像的管理可以类比作Jar包的管理，在虚拟机时代我们要部署程序时，都是将jar包或者war部署到机器上，然后执行它，docker容器化了之后，部署的是docker镜像，一般是部署在paas云平台上。

| **-** | **Jar包管理** | **Docker镜像管理** |
| --- | --- | --- |
| 官方仓库 | [https://hub.docker.com/](https://link.jianshu.com?t=https%3A%2F%2Fhub.docker.com%2F) | [http://mvnrepository.com/](https://link.jianshu.com?t=http%3A%2F%2Fmvnrepository.com%2F) |
| 私有仓库 | Nexus | harbor |
| 本地仓库命令 | maven：mvn install | docker build |
| 发布至远程仓库 | maven：mvn deploy | docker push |

## dockerfile-maven-plugin

该maven插件可以实现有java源代码直接到docker镜像，配置了dockerfile-maven-plugin插件的maven工具就像是一个管道，一端输入Java源代码，另一端直接输出docker镜像。

早先有一个插件docker-maven-plugin 可以不要Dockerfile，将构建镜像的操作配置在插件里面，这样不够清晰，不够灵活。不推荐使用。

引入插件依赖

|  |
| --- |
| <properties>  <dockerfile.maven.version>1.4.0</dockerfile.maven.version>  <docker.registry.name.prefix>192.168.6.37:80/galaxy</docker.registry.name.prefix>  </properties>  <plugin>  <groupId>com.spotify</groupId>  <artifactId>dockerfile-maven-plugin</artifactId>  <version>${dockerfile.maven.version}</version>  <dependencies>  <dependency>  <groupId>javax.activation</groupId>  <artifactId>activation</artifactId>  <version>1.1.1</version>  </dependency>  </dependencies>  <executions>  <execution>  <id>default</id>  <goals>  <goal>build</goal>  <goal>push</goal>  </goals>  </execution>  </executions>  <configuration>  <repository>${docker.registry.name.prefix}/${project.artifactId}</repository>  <tag>${project.version}</tag>  <username>username</username>  <password>password</password>  <buildArgs>  <JAR\_FILE>${project.build.finalName}.jar</JAR\_FILE>  </buildArgs>  </configuration>  </plugin> |

Dockerfile放在classpath的根目录下

用命令 mvn clean package

从<execution>的配置中可以看到，在执行maven命令的时候会默认执行docker的build和push。

单独执行dockerfile操作的maven命令是：mvn dockerfile:build 和 mvn dockerfile:push等。<configuration>中配置docker镜像的仓库的地址等。

使用gradle创建镜像并发布到镜像仓库

使用se.transmode.gradle:gradle-docker的文章

<https://blog.csdn.net/weixin_33713503/article/details/87404186>

<https://blog.csdn.net/baidu_22254181/article/details/80789068>

compile group: 'gradle.plugin.com.palantir.gradle.docker', name: 'gradle-docker', version: '0.22.1'

compile group: 'com.bmuschko', name: 'gradle-docker-plugin', version: '4.10.0'

构建镜像用命令

gradlew clean bootWar buildDocker

gradle构建镜像 需要在docker环境才能执行buildDocker任务

下面给个例子：

Build.gradle文件

|  |
| --- |
| buildscript {  repositories {  mavenLocal()  maven { url "http://10.100.5.141:8081/content/repositories/sss-dev/" }  maven { url "http://10.100.5.141:8081/content/repositories/Snapshots/" }  maven { url "http://10.100.5.141:8081/content/repositories/sss-prod/" }  maven { url "http://maven.aliyun.com/nexus/content/groups/public" }  maven { url 'http://maven.aliyun.com/nexus/content/repositories/google' }  maven { url 'http://maven.aliyun.com/nexus/content/repositories/gradle-plugin' }  maven { url "http://repo.spring.io/plugins-release" }  maven { url "http://plugins.gradle.org/m2/" }  jcenter()  }  dependencies {  classpath 'com.bmuschko:gradle-docker-plugin:3.2.5'  }  }      task copyDockerFiles(type: Copy) {  description = "Copy Dockerfile and required data to build directory"  from 'docker'  from "${project.buildDir}/libs"  into { "${project.buildDir}/docker" }  include "\*"  exclude "\*\*/\*.yml"  }  task buildDocker(type: DockerBuildImage, dependsOn: 'copyDockerFiles') {  description = "Package application as Docker image"  group = "Docker"  inputDir = project.file("${project.buildDir}/docker")  tags = ["sss:latest".toString(), "sss:${project.version}".toString()]  } |

Dockerfile文件

FROM openjdk8:v1.0

RUN echo "Asia/Shanghai" > /etc/timezone

ENV LANG zh\_CN.UTF-8

ENV LANGUAGE zh\_CN:zh

ENV LC\_ALL zh\_CN.UTF-8

RUN apk add --update ttf-dejavu && apk add curl && rm -rf /var/cache/apk/\*

ENV SPRING\_OUTPUT\_ANSI\_ENABLED=ALWAYS \

JHIPSTER\_SLEEP=0 \

JAVA\_OPTS=""

CMD echo "The application will start in ${JHIPSTER\_SLEEP}s..." && \

sleep ${JHIPSTER\_SLEEP} && \

java ${JAVA\_OPTS} -Djava.security.egd=file:/dev/./urandom -jar /app.war

EXPOSE 8080

ADD \*.war /app.war

HEALTHCHECK --interval=30s --timeout=20s --retries=5 \

CMD curl -fs http://localhost:8086/management/health || exit 1