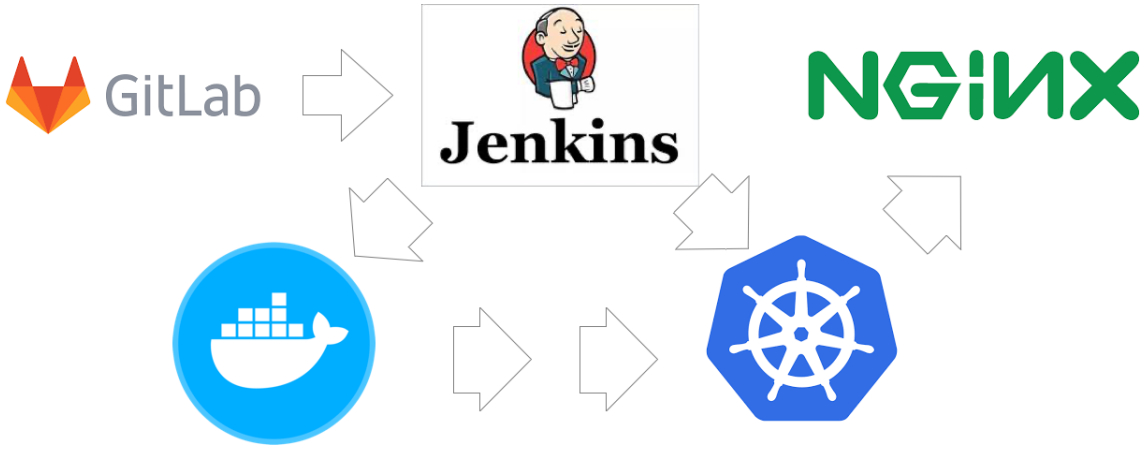
讲师(老司机)

容器虚拟化技术和自动化部署



jenkins-实战篇

jenkins入门

jenkins自由风格项目

git插件

```
jenkins工作台->系统管理->节点管理->可选插件->git
```

自由风格项目测试

```
进入jenins容器
docker exec -it jenkins sh

jenkins容器已经有如下环境：
java -version
git -version

jenkins工作台->->自由风格项目

```

jenkins分布式

master节点负责调度任务，agent节点负责执行任务。

配置固定节点

```
jenkins工作台->系统管理->节点管理->新增从节点
```

agent节点安装软件

均使用免安装方式进行安装

JDK8

```
下载地址：
https://www.oracle.com/webapps/redirect/signon?nexturl=https://download.oracle.com/otn/java/jdk/8u261-b12/a4634525489241b9a9e1aa73d9e118e6/jdk-8u261-linux-x64.tar.gz

tar -zxf jdk-8u241-linux-x64.tar.gz
```

maven3.6

```
下载地址：
https://mirrors.tuna.tsinghua.edu.cn/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz

tar -zxf apache-maven-3.6.3-bin.tar.gz.gz
mv apache-maven-3.6.3 maven
```

git2.28

```
1.安装依赖环境：
yum install -y curl-devel expat-devel gettext-devel openssl-devel zlib-devel gcc perl-ExtUtils-MakeMaker

2.删除yum方式安装的git：
添加依赖时自动yum安装了git1.8版本。需要先移除git1.8版本。
yum -y remove git
```

```
官网下载速度非常慢。国内加速地址大部分为windows版本。登录https://github.com/git/git/releases查看git的最新版。不要下载带有-rc的，因为它代表了一个候选发布版本。
https://www.kernel.org/pub/software/scm/git/git-2.28.0.tar.gz
```

```
tar -zxvf git-2.28.0.tar.gz

cd git-2.28.0

配置git安装路径
./configure --prefix=/opt/git/
编译并且安装
make && make install
```

统一配置

```
vi /etc/profile

export PATH
export JAVA_HOME=/opt/jdk1.8.0_241
export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
export MAVEN_HOME=/opt/maven
export PATH=$MAVEN_HOME/bin:$PATH
export PATH=$PATH:/opt/git/bin

source /etc/profile
```

配置软连接

```
master节点ssh连接agent节点时需要/usr/bin/有java命令。配置java的软连接、同理配置maven、git的软连接。如果软件是yum安装的，则需要检查/usr/bin中是否有相关命令。如果没有。也需要做软连接。

ln -s /opt/jdk1.8.0_241/bin/java /usr/bin/
ln -s /opt/mvn/bin/mvn /usr/bin/
ln -s /opt/git/bin/git /usr/bin
```

java方式连接agent

下载jar包

```
mkdir -p /data/workspaces
cd /data/workspaces

在google浏览器中复制jar地址
wget http://192.168.198.153:8080/jnlpJars/agent.jar

如果没有安装wget命令，选择yum方式安装：
yum install -y wget
```

启动连接

```
复制google浏览器中的启动命令：
java -jar agent.jar -jnlpUrl
http://192.168.198.153:8080/computer/jenkinsagent-154/slave-agent.jnlp -
secret db7f1e3fc92b1d57af545cae7d836c110d3994f73b618abd94ab0d63c29cfe20 -
workDir "/data/workspaces"
```

自由风格项目测试

```
配置好master和agent节点，创建一个自由风格项目，测试agent节点各种环境是否正常。
java -version
mvn -v
git version
docker version
```

jar包后台启动

```
https://www.bilibili.com/video/BV1fJ411Y73b?p=5
vi jenkinsagentstart.sh
#!/bin/bash
nohup java -jar agent.jar -jnlpUrl
http://192.168.198.153:8080/computer/jenkinsagent-154/slave-agent.jnlp -
secret db7f1e3fc92b1d57af545cae7d836c110d3994f73b618abd94ab0d63c29cfe20 -
workDir "/data/workspaces" &

chmod 777 jenkinsagentstart.sh

./jenkinsagentstart.sh

查看nohup启动日志：
tail -f nohup.out
```

SSH方式连接agent

免密配置

master节点要免密登录agent节点

```
生成秘钥
ssh-keygen -t rsa

复制公钥
ssh-copy-id -i /root/.ssh/id_rsa.pub root@192.168.198.154

免密登录测试
ssh 192.168.198.154


```

配置凭据

```
jenkins工作台->系统管理->凭据管理(manager credentials)

类型:SSH Username with private key

```

修改agent节点

```
jenkins工作台->系统管理->节点管理->选择agent节点->配置从节点->启动方式->Launch agents via SSH
```

```

```

自由风格项目测试

错误一

```
配置好master和agent节点，创建一个自由风格项目，测试agent节点各种环境是否正常。

注意事项：必须要增加#!/bin/bash。如果不增加，jenkins会出现Build step 'Execute shell' marked build as failure错误。脚本内容如下：

#!/bin/bash
java -version
mvn -v
git version
docker version
```

错误二

```
点击"立即构建"，发现java git docker命名都正常执行，而mvn命名未正常执行。
分析：
是因为jenkins远程调用agent节点时不会执行 source /etc/profile文件。那我们文件的配置不会生效。所以需要在我们的脚本中加入相关命令即可。脚本内容如下：

#!/bin/bash
source /etc/profile
java -version
mvn -v
git version
docker -v
```

gitlab安装

centos系统安装

提示各位小伙伴，安装之前一定要先做好快照。如果出错了。就回复快照信息。

安装相关依赖

```
yum -y install policycoreutils openssh-server openssh-clients postfix
```

启动ssh服务&设置为开机启动

```
systemctl enable sshd && sudo systemctl start sshd
```

设置postfix开机自启，并启动，postfix支持gitlab发信功能

```
systemctl enable postfix && systemctl start postfix
```

下载gitlab包，并且安装

```
清华大学地址：
https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el6/

在线下载安装包：
wget https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el6/gitlab-ce-12.7.6-ce.0.el6.x86_64.rpm

安装：
rpm -i gitlab-ce-12.7.6-ce.0.el6.x86_64.rpm
```

修改gitlab配置

```
修改gitlab访问地址和端口，默认为80，我们不进行修改。
external_url 'http://192.168.66.152'
```

重载配置及启动gitlab

```
gitlab-ctl reconfigure
gitlab-ctl restart
```

容器化安装

官网地址

```
https://hub.docker.com/r/gitlab/gitlab-ce
```

基础镜像

```
英文版
docker pull gitlab/gitlab-ce:12.7.6-ce.0

中文版
docker pull twang2218/gitlab-ce-zh:11.1.4

最新版:不是很稳定的版本
docker pull gitlab/gitlab-ce:13.3.2-ce.0
```

运行容器

```
运行镜像：运行时间比较长，大约需要3-10分钟。可以查看日志情况。
docker run -itd --name gitlab  -p 443:443 -p 80:80 -p 222:22 --restart always -m 4GB -v /data/gitlab/config:/etc/gitlab -v /data/gitlab/logs:/var/log/gitlab -v /data/gitlab/data:/var/opt/gitlab  -e TZ=Asia/Shanghai  gitlab/gitlab-ce:12.7.6-ce.0
```

配置gitlabe

```
配置项目访问地址：
external_url 'http://192.168.198.152'

配置ssh协议所使用的访问地址和端口
gitlab_rails['gitlab_ssh_host'] = '192.168.198.152'
gitlab_rails['time_zone'] = 'Asia/Shanghai'
gitlab_rails['gitlab_shell_ssh_port'] = 222
```

登录gitlab

登录gitlab：用户名默认为root。第一次登录需要设置密码。本教程将密码设置为12345678

```
username：root
password:12345678
```

常用命令练习

```
进入容器，练习常用gitlab命令：

docker exec -it gitlab /bin/bash

gitlab-ctl reconfigure
gitlab-ctl restart
gitlab-ctl status
```

创建组

```
组分三类：
Private：私有的
Internal：内部的
Public：公共的
```

创建项目

```
项目分类：
根据组的分类进行分类。

创建项目注意事项：
不需要创建README，否则本地项目无法上传到gitlab服务器上。
```

创建用户

```
1.创建用户
用户权限分两种：
Regular:普通权限用户
Admin:具有管理员权限的用户

2.给用户分配密码
```

将用户加入群组

```
给群组中的用户分配权限分五种：
Guest:可以创建issue、发表评论、不能读写版本库。
Reporter:可以克隆代码，不能提交、QA、PM可以赋予这个权限。
Developer:可以克隆代码、开发、提交、push，普通开发可以赋予这个权限。
Maintainer:可以创建项目、添加tag、保护分支、添加项目成员、编辑项目，核心开发人员可以赋予这个权限。
Owner:可以设置项目访问权限、-Visibility Level、删除项目、迁移项目、管理组成员、开发组组长可以赋予这个权限。
```

上传项目

```
使用idea开发工具演示
1.创建本地仓库
VCS->Enable Version Control Integration...

2.建立缓冲区
项目右键->git->Add

3.将代码提交到本地仓库
项目右键->git->Commit Directory

4.设置远程gitlab仓库地址
项目右键->git->Repository->Remote

5.将本地代码推送到远程gitlab仓库
项目右键->git->Repository->push
```

pipeline项目

Pipeline简介

概念

Pipeline，简单来说，就是一套运行在 Jenkins 上的工作流框架，将原来独立运行于单个或者多个节点的任务连接起来，实现单个任务难以完成的复杂流程编排和可视化的工作。

使用Pipeline好处

来自翻译自官方文档：

> 代码：Pipeline以代码的形式实现，通常被检入源代码控制，使团队能够编辑，审查和迭代其传送流
> 程。 持久：无论是计划内的还是计划外的服务器重启，Pipeline都是可恢复的。 可停止：Pipeline可接

> 收交互式输入，以确定是否继续执行Pipeline。 多功能：Pipeline支持现实世界中复杂的持续交付要
> 求。它支持fork/join、循环执行，并行执行任务的功能。 可扩展：Pipeline插件支持其DSL的自定义扩
> 展 ， 以及与其他插件集成的多个选项。

创建 Jenkins Pipeline任务

- Pipeline 脚本是由 **Groovy** 语言实现的，但是我们没必要单独去学习 Groovy
- Pipeline 支持两种语法：**Declarative**(声明式)和 **Scripted Pipeline**(脚本式)语法
- Pipeline 也有两种创建方法：可以直接在 Jenkins 的 Web UI 界面中输入脚本；也可以通过创建一个 Jenkinsfile 脚本文件放入项目源码库中（一般我们都推荐在 Jenkins 中直接从源代码控制(SCM)中直接载入 Jenkinsfile Pipeline 这种方法）。

安装git插件

```
jenkins工作台->系统管理->节点管理->可选插件->git
```

安装Pipeline插件

安装插件后，创建任务的时候多了“流水线”类型 。初始化jenkins环境时已经默认安装了pipeline插件。

```
jenkins工作台->系统管理->节点管理->可选插件->pipeline
```

Pipeline语法快速入门

Scripted脚本式-Pipeline

```
新建任务
pipeline-test02

选择模板
scripted pipeline

片段生成器中选择echo

node ('jenkinsagent-154') {

    stage('Preparation') { // for display purposes
        echo 'hello pipeline'
    }

}
```

Declarative声明式-Pipeline

```
新建任务
pipeline-test02

选择模板
Hello world
```

agent配置

```
agent选项：
any ： 在任何可用的机器上执行pipeline
none ： 当在pipeline顶层使用none时，每个stage需要指定相应的agent
```

流水线语法

- stages：代表整个流水线的所有执行阶段。通常stages只有1个，里面包含多个stage
- stage：代表流水线中的某个阶段，可能出现n个。一般分为拉取代码，编译构建，部署等阶段。
- steps：代表一个阶段内需要执行的逻辑。steps里面是shell脚本，git拉取代码，ssh远程发布等任意内
  容。

```
任务->流水线->点击链接 "流水线语法"

选择 Declarative Directive Generator

Directives->Sample Directive->选择agent:Agent选项
Agent选择 Label:Run on an agent matching a label

Label:输入agent节点标签内容。"jenkinsagent-154"

点击Generator Declarative Directive按钮，复制生成内容替换任务的agent any部分

agent {
  label 'jenkinsagent-154'
}
```

测试Declarative任务

```
点击 立即构建
```

升级案例

```
pipeline {
   agent {
      label 'jenkinsagent-154'
    }
    stages {
        stage('检测环境') {
            steps {
                sh label: '', script: '''java -version
                mvn -v
                git version
```

```
                docker -v'''
            }
        }
        stage('拉取代码') {
            steps {
               echo '拉取代码'
            }
        }
        stage('编译构建') {
            steps {
               echo '编译构建'
            }
        }
        stage('项目部署') {
            steps {
               echo '项目部署'
            }
        }
    }
}
```

测试pipeline项目

出现mvn命令没有找到错误。

解决方案一

```
配置jenkinsagent-154节点。在节点信息中增加环境变量配置

测试脚本。脚本正常执行
```

解决方案二

```
增加mvn命令的软连接，将mvn命令追加至/usr/local/bin目录中，具体命令如下：
ln -s /opt/maven/bin/mvn /usr/local/bin/

测试脚本。脚本正常执行
```

Declarative pipeline和Scripted pipeline的比较

共同点

两者都是pipeline代码的持久实现，都能够使用pipeline内置的插件或者插件提供的steps，两者都可以利用共享库扩展。

区别

两者不同之处在于语法和灵活性。Declarative pipeline对用户来说，语法更严格，有固定的组织结构，更容易生成代码段，使其成为用户更理想的选择。但是Scripted pipeline更加灵活，因为Groovy本身只能对结构和语法进行限制，对于更复杂的pipeline来说，用户可以根据自己的业务进行灵活的实现和扩展。

集成gitlab

http方式

```
gitlabe->clone->选择http方式
http://192.168.198.152/lagou/jenkinsdemo.git
```

配置凭据

```
jenkins工作台->系统管理->凭据管理(manager credentials)

类型:Username with password

```

修改脚本

```
1.片段生成器中选择check out


2.修改pipeline-test03任务中的拉取代码阶段:
        stage('拉取代码') {
            steps {
               echo 'gitlab拉取代码'
               checkout([$class: 'GitSCM', branches: [[name: '*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'b26bd84e-e0cb-4b90-8469-1c2a46213466', url: 'http://192.168.198.152/lagou/jenkinsdemo.git']]])
            }
        }
```

测试任务

```
点击立即构建
```

ssh方式

免密配置

gitlabe-152节点免密登录配置。gitlab服务器保存公钥信息。方便访问gitlab-152服务器。

```
1.生成秘钥
ssh-keygen -t rsa

2.查看公钥信息
cat /root/.ssh/id_rsa.pub


3.gitlabe服务器配置:
当前用户->setting->SSH Key->点击 add key按钮
```

配置凭据

```
1.jenkins工作台->系统管理->凭据管理(manager credentials)。保存gitlab-152服务器的私钥信息。

2.类型:SSH Username with private key
```

修改脚本

```
修改pipeline-test03任务中的拉取代码阶段：
        stage('拉取代码') {
            steps {
             echo 'gitlab拉取代码'
             checkout([$class: 'GitSCM', branches: [[name: '*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'c8634952-4993-4455-b164-35427823144f', url: 'ssh://git@192.168.198.152:222/lagou/jenkinsdemo.git']]])
            }
        }
```

测试任务

```
点击立即构建
```

Pipeline Script from SCM

刚才我们都是直接在Jenkins的UI界面编写Pipeline代码，这样不方便脚本维护，建议把Pipeline脚本放在项目中（一起进行版本控制）

Jenkinsfile文件

在jenkinsdemo项目根目录创建/Jenkinsfile文件。Jenkinsfile文件内容如下：

```
pipeline {
   agent {
      label 'jenkinsagent-154'
    }

    stages {
        stage('检测环境') {
            steps {
                sh label: '', script: '''java -version
                mvn -v
                git version
                docker -v'''
            }
        }
```

```
        stage('拉取代码') {
            steps {
                echo 'gitlab拉取代码'
                checkout([$class: 'GitSCM', branches: [[name: '*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'c8634952-4993-4455-b164-35427823144f', url: 'ssh://git@192.168.198.152:222/lagou/jenkinsdemo.git']]])
            }
        }
        stage('编译构建') {
            steps {
                echo 'mvn 编译构建'
            }
        }
        stage('项目部署') {
            steps {
                echo 'java项目部署'
            }
        }
    }
}
```

修改pipeline项目

```
配置 SCM相关配置
```

测试pipeline项目

```
查看控制台输出信息
```

agent节点配置maven

为pipeline项目增加maven打包jenkinsdemo项目配置信息

maven配置

/opt/maven/conf/settings.xml文件配置

1.配置仓库地址

```
创建本地仓库：
mkdir -p /data/maven/repository

设置本地仓库目录
<localRepository>/data/maven/repository</localRepository>
```

2.阿里云镜像仓库地址

```
<mirror>
    <id>nexus-aliyun</id>
    <mirrorOf>*</mirrorOf>
    <name>Nexus aliyun</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

3.maven工程JDK8编译配置

```
<profile>
    <id>jdk-1.8</id>
    <activation>
        <activeByDefault>true</activeByDefault>
        <jdk>1.8</jdk>
    </activation>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>

 <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
    </properties>
</profile>
```

修改Jenkinsfile文件

修改Jenkinsfile文件中的编译构建步骤。增加mvn 相关命令。

```
在片段生成器中找到shell命令相关配置，修改Jenkinsfile文件如下：

pipeline {
   agent {
      label 'jenkinsagent-154'
    }

    stages {
        stage('检测环境') {
            steps {
                sh label: '', script: '''java -version
                mvn -v
                git version
                docker -v'''
            }
        }
        stage('拉取代码') {
            steps {
                echo 'gitlab拉取代码'
                checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: 'c8634952-4993-4455-b164-35427823144f',
url: 'ssh://git@192.168.198.152:222/lagou/jenkinsdemo.git']]])
            }
        }
        stage('编译构建') {
            steps {
```

```
                echo 'mvn 编译构建'
                sh label: '', script: 'mvn clean package'
            }
        }
        stage('项目部署') {
            steps {
                echo 'java项目部署'
            }
        }
    }
}
```

修改Jenkinsfile文件

修改Jenkinsfile文件中的项目部署步骤。增加shell相关命令。pipeline一个stage的steps中不支持多条shell命令。可以将shell命令写在同一行中，命令和命令之间用&&符号隔开。

```
在片段生成器中找到shell命令相关配置，修改Jenkinsfile文件如下：

pipeline {
   agent {
      label 'jenkinsagent-154'
    }

    stages {
        stage('拉取代码') {
            steps {
               checkout([$class: 'GitSCM', branches: [[name: '*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'dffadad1-62bd-4b16-8438-cc36be8b8d8d', url: 'http://192.168.198.152/lagou/jenkinsdemo.git']]])
            }
        }
        stage('编译构建') {
            steps {
               sh label: '', script: 'mvn clean package -Dmaven.test.skip=true'
            }
        }
        stage('项目部署') {
            steps {
               sh label: '', script: 'cd target/ && pwd && java -jar jenkinsdemo.jar'
            }
        }
    }
}

```

测试pipeline项目

```
在浏览器中访问项目

http://192.168.198.154:8080
```

手工制作Docker镜像

制作步骤汇总

```
实验步骤：
1.编写Dockerfile文件
2.使用mvn命令打包工程
3.使用docker build命令构建镜像
4.使用docker run命令运行镜像
5.浏览器端测试实验结果
```

Dockerfile回顾

Dockerfile其实就是我们用来构建Docker镜像的源码，当然这不是所谓的编程源码，而是一些命令的集合，只要理解它的逻辑和语法格式，就可以很容易的编写Dockerfile。简单点说，Dockerfile可以让用户个性化定制Docker镜像。因为工作环境中的需求各式各样，网络上的镜像很难满足实际的需求。

Dockerfile常见命令

命令	作用
FROM	image_name:tag，选择基础镜像
MAINTAINER	user_name,声明镜像的作者
ENV	设置环境变量 (可以写多条)
RUN	编译镜像时运行的脚本(可以写多条)。用于指定 docker build 过程中要运行的命令，即是创建 Docker 镜像（image）的步骤
CMD	设置容器的启动命令
ENTRYPOINT	设置容器的入口程序
ADD	将宿主机的文件复制到容器内，如果是一个压缩文件，将会在复制后自动解压
COPY	和ADD相似，但是如果有压缩文件并不能解压
WORKDIR	设置工作目录
ARG	设置编译镜像时加入的参数
VOLUMN	设置容器的挂载卷

面试题一

CMD和ENTRYPOINT的区别

RUN、CMD 和 ENTRYPOINT 这三个 Dockerfile 指令看上去很类似，很容易混淆。简单的说：

1. RUN 执行命令并创建新的镜像层，RUN 经常用于安装软件包。用于指定 docker build 过程中要运行的命令，即是创建 Docker 镜像（image）的步骤
2. CMD 设置容器启动后默认执行的命令及其参数，但 CMD 能够被 `docker run` 后面跟的命令行参数替换。 Dockerfile 中只能有一条 CMD 命令，如果写了多条则最后一条生效。CMD不支持接收

docker run的参数。

3. ENTRYPOINT 入口程序是容器启动时执行的程序， docker run 中最后的命令将作为参数传递给入口程序 ，ENTRYPOINY类似于 CMD 指令，但可以接收docker run的参数 。

面试题二

ADD和COPY的区别

1. `ADD`指令可以添加URL资源,或者说可以直接从远程添加文件到镜像中，而`COPY`不具备这样的能力
2. 如果没有特别要求，尽可能用 COPY，可以减少发生不明异常的情况；如果确实需要 ADD 的独特特性，那么还是得清楚自己的 ADD 用法是正确的。

Dockerfile文件

```
FROM openjdk:8-alpine3.9
# 作者信息
MAINTAINER laosiji Docker springboot "laosiji@lagou.com"
# 修改源
RUN echo "http://mirrors.aliyun.com/alpine/latest-stable/main/" > /etc/apk/repositories && \
    echo "http://mirrors.aliyun.com/alpine/latest-stable/community/" >> /etc/apk/repositories

# 安装需要的软件，解决时区问题
RUN apk --update add curl bash tzdata && \
    rm -rf /var/cache/apk/*

#修改镜像为东八区时间
ENV TZ Asia/Shanghai
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/app.jar"]
```

dockerfile-maven-plugin

简介

越来越多的项目开始了docker容器化部署的进化，在容器化之前我们部署一个项目，可能由源代码产出一个jar或者war即可直接发布了，启动之后是一个java进程；容器化之后，由源代码产出的是一个docker镜像，而启动的是一个容器。多了这么多步骤是为了容器化之后的运维便利性，从现在来看，容器化是势不可挡的趋势。为了实现的我们CI/CD的终极目标：一键发布，这里介绍一个maven plugin（输入源代码，输出docker镜像）。

作为一个Docker项目，都绕不过Dockerfile文件构建、打标签和发布等操作。如果能够将对Dockerfile文件的这些操作包含进Maven的生命周期管理中，将极大简化Docker项目的构建发布过程。Dockerfile Maven是Spotify公司提供的一款Maven插件（还包含一个Maven扩展），用以辅助Docker项目（借助于Maven管理）进行Dockerfile文件构建、标签和发布。

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

上述流程在工具齐全的情况下，是相当灵活好用的，公司一般都是这么使用的，而且也能将职责明确。但是当工具不够齐全的时候，或者说个人单打独斗的时候，会使用的工具有限，就寄希望于一个工具能够搞定更多的事情。dockerfile-maven-plugin 就是这样一个maven工具的插件。

设计目标

这是一个将Docker与Maven无缝集成的Maven插件，可以方便地使用Maven打包Docker image。在dockerfile-maven-plugin插件出现之前，还有一个maven插件是docker-maven-plugin，是由同一个作者创造，作者明确表示推荐使用dockerfile-maven-plugin，并会持续升级；而docker-maven-plugin不在添加任何新功能，只接受修复bug。两个插件的设计思想是有差别的，前者需要独立编写Dockerfile文件，后者允许没有Dockerfile文件，所有的docker相关配置都写在pom文件的plugin中，这样使maven插件显得很笨重，并且如果脱离了这个maven插件还得重写编写Dockerfile，不够灵活。

- 不要试图做任何事情。 这个插件使用Dockerfiles构建Docker项目的而且是强制性的。
- 将Docker构建过程集成到Maven构建过程中。如果绑定默认phases，那么当你键入mvn package时，你会得到一个Docker镜像。 当你键入mvn deploy时，你的图像被push。
- 让goals记住你在做什么。 你可以输入 mvn dockerfile:build及后面的 mvn dockerfile:build和 mvn dockerfile:push 都没有问题。这也消除了之前像 mvn dockerfile:build -DalsoPush这样的命令；相反，你可以只使用 mvn dockerfile:build dockerfile:push。

与Maven build reactor集成。你可以在一个项目中依赖另一个项目所构建的Docker image，Maven将按照正确的顺序构建项目。当你想要运行涉及多个服务的集成测试时，这非常有用。

版本说明

老版本

```
插件名称：
docker-maven-plugin

github官网地址：
https://github.com/spotify/docker-maven-plugin
```

最新版本

```
<dependency>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>1.2.2</version>
</dependency>
```

新版本

该插件需要Java 7或更高版本以及Apache Maven 3或更高版本。要运行集成测试或在开发中使用该插件，需要有一个能正常工作的Docker。Docker已经允许远程连接访问。dockerfile-maven-plugin要求用户必须提供Dockerfile用于构建镜像，从而将Docker镜像的构建依据统一到Dockerfile上，这与过时的docker-maven-plugin是不同的。

```
插件名称：
dockerfile-maven-plugin

github官网地址：
https://github.com/spotify/dockerfile-maven
```

最新版本

```
官网很久没有更新新版本
<dependency>
    <groupId>com.spotify</groupId>
    <artifactId>dockerfile-maven-plugin</artifactId>
    <version>1.4.13</version>
</dependency>
```

docker-maven-plugin插件入门

推荐大家在学习之前对jenkinsmater-153、jenkinsagent-154、gitlab-152三台服务器进行快照保存操作。

idea集成docker

idea安装docker插件。Dockerfile、docker-compose.yml文件大部分内容会有提示信息。方便开发人员编写配置文件。

```
官网地址：
https://plugins.jetbrains.com/plugin/7724-docker/versions
```

jenkinsagent-154配置

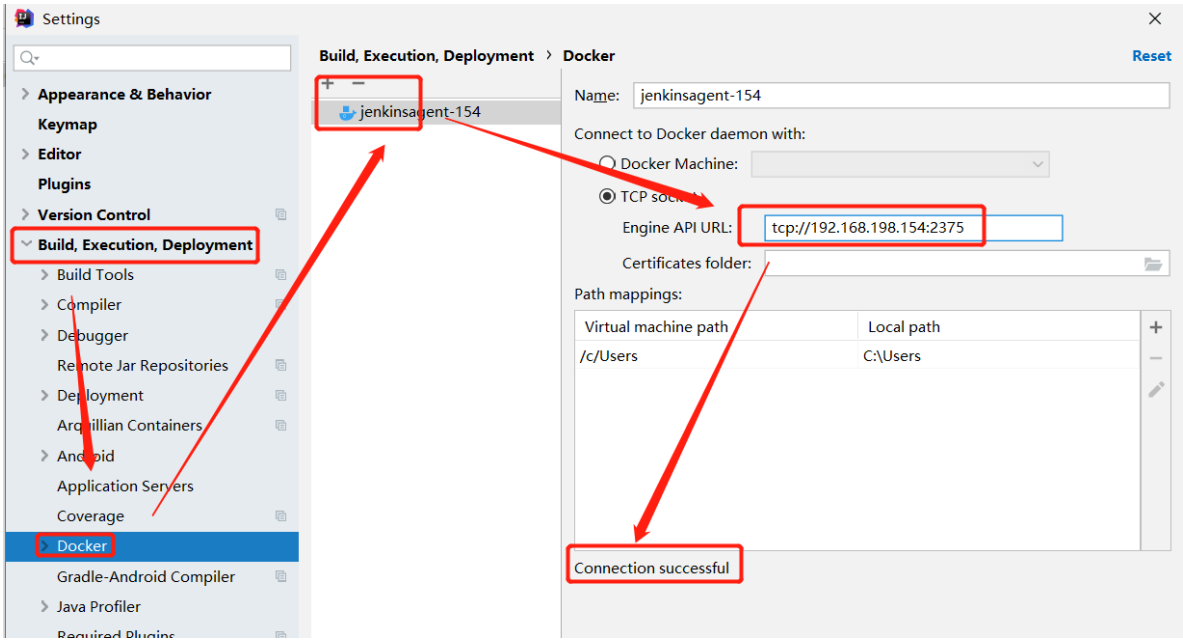
修改jenkinsagent-154服务器docker.service服务信息，允许其他主机远程访问154服务器的docker。

```
vi /usr/lib/systemd/system/docker.service

在ExecStart行最后增加，开放远程主机访问权限。
-H tcp://0.0.0.0:2375

最后增加修改内容如下：
ExecStart=/usr/bin/dockerd -H fd:// --
containerd=/run/containerd/containerd.sock -H tcp://0.0.0.0:2375


重启docker
systemctl daemon-reload
systemctl restart docker

查看docker进程，发现docker守护进程在已经监听2375的tcp端口
ps -ef|grep docker


查看系统的网络端口，检查tcp的2375端口,docker的守护进程是否监听
netstat -tulp
```

配置idea

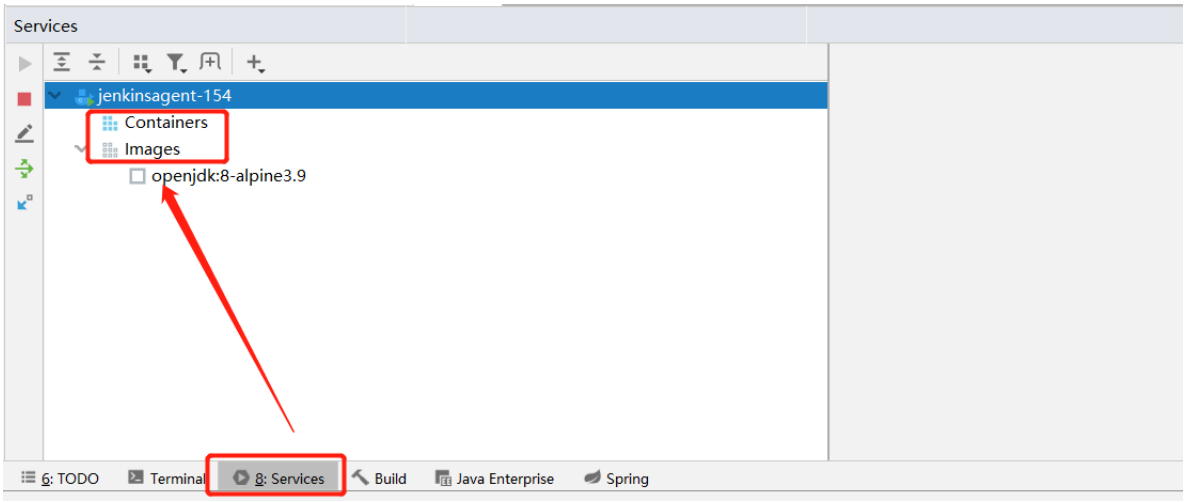
配置插件

```
settings->build execution...->docker->点击"+"按钮，新增jenkinsagent-154服务器
docker配置信息

配置内容如下：
name:jenkinsagent-154
TCP Socket:
    Engine API URL:tcp://192.168.198.154:2375

配置成功后，会在下方显示connection successful
```



操作docker

配置成功后，会在idea开发工具下方窗口"8.services"里显示信息，右键点击connect。连接成功可以查看到container和images等信息。可以对container和images进行各种相关操作。



新建微服项目

新增jenkinsdemo1工程。

pom.xml文件

```
<build>
        <finalName>${project.artifactId}</finalName>
        <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>1.2.2</version>
                <configuration>
                    <!--修改imageName节点的内容，改为私有仓库地址和端口，再加上镜像id和TAG,我们要直接传到私服-->
                    <!--配置最后生成的镜像名，docker images里的，我们这边取项目名:版本-->

 <imageName>${project.build.finalName}:${project.version}</imageName>
                    <!--也可以通过以下方式定义image的tag信息。
                    <imageTags>
                        <imageTag>1.0</imageTag>
                    </imageTags>
                    -->
                    <!--来指明Dockerfile文件的所在目录-->
                    <dockerDirectory>${project.basedir}</dockerDirectory>
                    <dockerHost>http://192.168.198.154:2375</dockerHost>
```

```
                    <!--入口点，project.build.finalName就是project标签下的build标签下的filename标签内容，testDocker-->
                    <!--相当于启动容器后，会自动执行java-jar/testDocker.jar-->
                    <entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>
                    <!--是否推送到docker私有仓库，旧版本插件要配置maven的settings文件。小伙伴们可以自行查阅资料研究一下。
                    <pushImage>true</pushImage>

 <registryUrl>192.168.198.155:5000/lagouedu</registryUrl>
                    -->
                    <!-- 这里是复制 jar 包到 docker 容器指定目录配置 -->
                    <resources>
                        <resource>
                            <targetPath>/</targetPath>
                            <directory>${project.build.directory}</directory>
                            <!--把哪个文件上传到docker，相当于Dockerfile里的add app.jar /-->

 <include>${project.build.finalName}.jar</include>
                        </resource>
                    </resources>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Dockerfile

```
FROM openjdk:8-alpine3.9
# 作者信息
MAINTAINER laosiji Docker springboot "laosiji@lagou.com"
# 修改源
RUN echo "http://mirrors.aliyun.com/alpine/latest-stable/main/" > /etc/apk/repositories && \
    echo "http://mirrors.aliyun.com/alpine/latest-stable/community/" >> /etc/apk/repositories

# 安装需要的软件，解决时区问题
RUN apk --update add curl bash tzdata && \
    rm -rf /var/cache/apk/*

#修改镜像为东八区时间
ENV TZ Asia/Shanghai
ADD /target/jenkinsdemo1.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/app.jar"]
```

controller

```java
@RestController
public class JenkinsDemoController {

    @GetMapping("/")
    public String hello() {
        return "idea docker docker-maven-plugin hello!!!";
    }
}
```

打包部署

```
idea在ternimal窗口中运行如下命令
mvn clean package -Dmavn.test.skip=true docker:build
```

在idea中运行容器

使用idea与docker集成插件生成容器。

使用dockerfile-maven-plugin插件完善项目

pom.xml

在pom文件中配置dockerfile插件信息

```xml
    <build>
        <finalName>jenkinsdemo</finalName>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>com.spotify</groupId>
                <artifactId>dockerfile-maven-plugin</artifactId>
                <version>1.4.13</version>
                <configuration>
                    <repository>${project.build.finalName}</repository>
                    <tag>1.0</tag>
                    <buildArgs>
```

```
<JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
                    </buildArgs>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

可选配置

跳过测试环节的插件配置。

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <configuration>
                    <skipTests>true</skipTests>
                </configuration>
            </plugin>
```

Dockfile

在项目根目录创建Dockerfile文件

```
FROM openjdk:8-alpine3.9
# 作者信息
MAINTAINER laosiji Docker springboot "laosiji@lagou.com"
# 修改源
RUN echo "http://mirrors.aliyun.com/alpine/latest-stable/main/" > /etc/apk/repositories && \
    echo "http://mirrors.aliyun.com/alpine/latest-stable/community/" >> /etc/apk/repositories

# 安装需要的软件，解决时区问题
RUN apk --update add curl bash tzdata && \
    rm -rf /var/cache/apk/*

#修改镜像为东八区时间
ENV TZ Asia/Shanghai
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/app.jar"]
```

修改Jenkinsfile文件

更新Jenkinsfile文件中项目部署环节

```
        stage('项目部署') {
            steps {
               sh label: '', script: 'mvn dockerfile:build'
            }
        }
```

完整Jenkinsfile文件信息

```
pipeline {
   agent {
      label 'jenkinsagent-154'
    }
    stages {
        stage('拉取代码') {
            steps {
               checkout([$class: 'GitSCM', branches: [[name: '*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: '52247b8c-05a0-444e-bfe0-1a560ff86ba2', url: 'ssh://git@192.168.198.152:222/lagou/jenkinsdemo.git']]])
            }
        }
        stage('编译构建') {
            steps {
               sh label: '', script: 'mvn clean package -Dmaven.test.skip=true'
            }
        }
        stage('项目部署') {
            steps {
               sh label: '', script: 'mvn dockerfile:build'
            }
        }
    }
}
```

测试pipeline任务

```
构建成功后，在jenkinsagent-154节点查看镜像生成信息
docker images
```

完善pipeline任务

新增删除镜像阶段

脚本内容

在jenkinsagent-154服务器新建测试脚本。

```
cd /data
vi test.sh

脚本内容如下：
#!/bin/bash
echo '检查镜像是否存在'
imageid=`docker images | grep jenkins | awk '{print $3}'`
if [ "$imageid" != "" ];then
  echo '删除镜像'
  docker rmi -f  $imageid
fi

```

```
给脚本授权
chmod 777 test.sh

执行脚本
./test.sh

检查镜像是否被删除
docker images
```

修改Jenkinsfile文件

在编译构建阶段后新增删除镜像阶段

```
        stage('删除镜像') {
            steps {
               sh label: '', script: '''echo \'检查镜像是否存在\'
               imageid=`docker images | grep jenkinsdemo | awk \'{print $3}\'`
               if [ "$imageid" != "" ];then

                 echo \'删除镜像\'
                 docker rmi -f  $imageid
               fi'''
            }
        }
```

测试pipeline任务

```
构建成功后，在jenkinsagent-154节点查看镜像生成信息
docker images
```

多次构建后，积累的无用镜像

构建多次后，本地会遗留多个名为，tag也是的镜像。 这些都是上一次构建的结果，在经历了新一轮的构建后，其镜像名和tag被新镜像所有，所以自身只能显示名为，tag也是，清理这些镜像的命令是docker image prune，然后根据提示输入"y"，镜像即可被清理：

```
docker image prune
提示信息如下
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Deleted Images:

```

新增运行容器阶段

修改Jenkinsfile文件

在构建镜像阶段后新增运行容器阶段

```
        stage('运行容器') {
            steps {
               sh label: '', script: 'docker run -itd --name=jenkinsdemo -p 8080:8099 jenkinsdemo:1.0'
            }
        }
```

测试pipeline任务

```
构建成功后，在jenkinsagent-154节点查看镜像生成信息
docker images

docker ps -a

http://192.168.198.154:8080
```

新增删除容器阶段

脚本内容

在jenkinsagent-154服务器修改test.sh测试脚本。

```
cd /data
vi test.sh

脚本内容如下：
#!/bin/bash
echo '检查容器是否存在'
containerid=`docker ps -a | grep -w jenkinsdemo | awk '{print $1}'`
if [ "$containerid" != "" ];then
  echo '容器存在，停止容器'
  docker stop $containerid
  echo '删除容器'
  docker rm $containerid
fi

echo '检查镜像是否存在'
imageid=`docker images | grep jenkinsdemo | awk '{print $3}'`
if [ "$imageid" != "" ];then

  echo '删除镜像'
  docker rmi -f  $imageid
fi


执行脚本
./test.sh

检查容器是否被删除
docker ps -a

检查镜像是否被删除
docker images
```

修改Jenkinsfile文件

在编译构建阶段后新增删除容器阶段

```
        stage('删除容器') {
            steps {
                sh label: '', script: '''echo \'检查容器是否存在\'
                containerid=`docker ps -a | grep -w jenkinsdemo | awk \'{print $1}\'`
                if [ "$containerid" != "" ];then
                  echo '容器存在，停止容器'
                  docker stop $containerid
                  echo '删除容器'
                  docker rm $containerid
                fi'''
            }
        }
```

测试pipeline任务

```
构建成功后，在jenkinsagent-154节点查看镜像生成信息
docker images

docker ps -a

浏览器端访问项目：
http://192.168.198.154:8080
```

harbor私服

本章节讨论如何将镜像推送到harbor仓库，再从harbor仓库拉取镜像。运行镜像。

初始化环境

在jenkinsagent-154服务器执行test.sh脚本。删除产生的容器、镜像信息

```
cd /data
./test.sh

docker ps -a
docker images
```

配置harbor私服

jenkinsagent-154服务器配置docker登录harbor私服信息。

配置私服

```
vi /etc/docker/daemon.json
"insecure-registries":["192.168.198.155:5000"]

重启docker服务:
systemctl daemon-reload
systemctl restart docker
```

登录私服

```
docker login -u admin -p Harbor12345 192.168.198.155:5000


退出私服
docker logout 192.168.198.155:5000
```

修改pom文件

新增harbor私服地址、用户名、密码，镜像tag等配置项。

```
            <plugin>
                <groupId>com.spotify</groupId>
                <artifactId>dockerfile-maven-plugin</artifactId>
                <version>1.4.13</version>
                <configuration>

 <repository>192.168.198.155:5000/lagouedu/${project.build.finalName}
</repository>
                    <username>admin</username>
                    <password>Harbor12345</password>
                    <tag>1.0</tag>
                    <buildArgs>

 <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
                    </buildArgs>
                </configuration>
            </plugin>
```

修改Jenkinsfile文件

修改构建镜像阶段、运行容器阶段信息。

- 构建镜像阶段新增dockerfile:push推送镜像信息
- 运行容器阶段修改镜像名称

```
        stage('构建镜像') {
            steps {
               sh label: '', script: 'mvn dockerfile:build dockerfile:push'
            }
        }
        stage('运行容器') {
            steps {
               sh label: '', script: 'docker run -itd --name=jenkinsdemo -p 8080:8099 192.168.198.155:5000/lagouedu/jenkinsdemo:1.0'
            }
        }
```

测试pipeline任务

```
构建成功后，在jenkinsagent-154节点查看镜像生成信息
docker images

docker ps -a

浏览器端访问项目：
http://192.168.198.154:8080
```

jib插件

简介

今天给大家介绍的是由Google出品的容器镜像构建类库--Jib, 通过Jib可以非常简单快速的为你的Java应用构建Docker 和 OCI 镜像, 无需深入学习docker, 无需编写Dockerfile, 以 Maven插件、Gradle插件和Java lib的形式提供。

三种使用Jib的方法：

1. Maven插件：jib-maven-plugin；
2. Gradle插件：jib-gradle-plugin；
3. Java库：Jib Core；

Jib目标

- **Fast**- 快速部署您的更改。Jib将您的应用程序分成多个层，从类中分离依赖项。现在您不必等待Docker重建整个Java应用程序 - 只需部署更改的层即可。
- **Reproducible**- 使用相同内容重建容器图像始终生成相同的图像。不用担心再次触发不必要的更新。
- **Daemonless**- 减少CLI依赖性。从Maven或Gradle中构建Docker镜像，然后推送到您选择的任何注册中心。不再编写Dockerfiles并调用docker build / push。

官网地址

```
github官网地址
https://github.com/GoogleContainerTools/jib
```

最新版本

```
<dependency>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>2.5.2</version>
</dependency>
```

基础镜像

```
docker pull openjdk:8-alpine3.9

重新打标签
docker tag openjdk:8-alpine3.9 192.168.198.155:5000/lagouedu/openjdk:8-alpine3.9

登录harbor-155私服
docker login 192.168.198.155:5000
username:admin
password:Harbor12345


上传镜像
docker push 192.168.198.155:5000/lagouedu/openjdk:8-alpine3.9

删除jenkinsagent-154镜像
docker rmi -f 192.168.198.155:5000/lagouedu/openjdk:8-alpine3.9 openjdk:8-alpine3.9
```

项目配置

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>com.google.cloud.tools</groupId>
                <artifactId>jib-maven-plugin</artifactId>
                <version>2.5.2</version>
                <configuration>
```

```
                    <!--from节点用来设置镜像的基础镜像，相当于Docerkfile中的FROM关键字-->
                    <from>
                        <!--使用harbor-155上的openjdk镜像-->
                        <image>192.168.198.155:5000/lagouedu/openjdk:8-alpine3.9</image>
                        <!--harbor-155服务器的登录信息-->
                        <auth>
                            <username>admin</username>
                            <password>Harbor12345</password>
                        </auth>
                    </from>

                    <to>
                        <!--镜像名称和tag，使用了mvn内置变量${project.version}，表示当前工程的version-->
                        <image>192.168.198.155:5000/lagouedu/jenkinsdemo:${project.version}</image>
                        <auth>
                            <username>admin</username>
                            <password>Harbor12345</password>
                        </auth>
                    </to>

                    <container>
                        <!--配置jvm虚拟机参数-->
                        <jvmFlags>
                            <jvmFlag>-Xms512m</jvmFlag>
                        </jvmFlags>
                        <!--配置使用的时区-->
                        <environment>
                            <TZ>Asia/Shanghai</TZ>
                        </environment>
                        <!--要暴露的端口-->
                        <ports>
                            <port>8080</port>
                        </ports>
                    </container>
                    <!--可以进行HTTP-->
                    <allowInsecureRegistries>true</allowInsecureRegistries>
                </configuration>
                <!--将jib与mvn构建的生命周期绑定 mvn package自动构造镜像-->
                <!--打包及推送命令 mvn  -DsendCredentialsOverHttp=true clean package-->
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>
                                build
                            </goal>
                        </goals>
                    </execution>
                </executions>

            </plugin>
        </plugins>
```

```
    </build>
```

container元素介绍

```
container配置:
这个标签主要配置目标容器相关的内容,比如:

appRoot -> 放置应用程序的根目录,用于war包项目
args -> 程序额外的启动参数.
environment -> 用于容器的环境变量
format -> 构建OCI规范的镜像
jvmFlags -> JVM参数
mainClass -> 程序启动类
ports -> 容器开放端口

详细资料请参考官网地址:
https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin#container-object
```

controller

controller/JibDemoController

```
@RestController
public class JibDemoController {
    @GetMapping("/")
    public String hello() {
        return "docker jib-maven-plugin jenkins hello!!!";
    }
}
```

构建镜像

```
执行命令：
mvn clean package -Dmaven.test.skip=true jib:build

执行命令后报错,错误的原因是由于 from image 配置的基础镜像需要认证信息必须要增加
 -DsendCredentialsOverHttp=true的参数。


再次执行命令：
mvn clean package -Dmaven.test.skip=true jib:build -
DsendCredentialsOverHttp=true
```

三种构建参数

对于一个已在pom.xml中配置了jib插件的java工程来说，下面是个标准的构建命令

```
mvn compile jib:dockerBuild
```

注意上面的dockerBuild参数，该参数的意思是将镜像存入当前的镜像仓库，这样的参数一共有三种，列表说明

参数名	作用
dockerBuild	将镜像存入当前镜像仓库，该仓库是当前docker客户端可以连接的docker daemon，一般是指本地镜像仓库
build	将镜像推送到远程仓库，仓库位置与镜像名字的前缀有关，一般是hub.docker.com，使用该参数时需要提前登录成功
buildTar	将镜像生成tar文件，保存在项目的target目录下，在任何docker环境执行docker load --input xxx.tar即可导入到本地镜像仓库

镜像的时间问题

在使用命令mvn compile jib:dockerBuild构建本地镜像时，会遇到创建时间不准的问题：如下所示，lagou/jenkins:1.0是刚刚使用jib插件构建的镜像，其生成时间(CREATED字段)显示的是50 years ago：

```
在jenkinsagent-154服务器拉取镜像
docker pull 192.168.198.155:5000/lagouedu/jenkinsdemo:1.0

查看镜像
docker images
```

上面显示的镜像生成时间显然是不对的，改正此问题的方法是修改pom.xml，在jib插件的container节点内增加creationTime节点，内容是maven.build.timestamp的时间，如下所示：

```
<container>
    <!--创建时间-->
    <creationTime>${maven.build.timestamp}</creationTime>
</container>
```

修改保存后再次构建，此时新的镜像的创建时间已经正确

```
删除jenkinsagent-154服务器上镜像
docker rmi -f 192.168.198.155:5000/lagouedu/jenkinsdemo:1.0

在harbor-155服务器上删除镜像像

在idea中再次构建镜像
mvn clean package -Dtest.skip=true jib:build -DsendCredentialsOverHttp=true

在jenkinsagent-154服务器拉取镜像
docker pull 192.168.198.155:5000/lagouedu/jenkinsdemo:1.0

查看镜像
docker images

运行容器
docker run -itd --name jenkinsdemo -p 8080:8080
192.168.198.155:5000/lagouedu/jenkinsdemo:1.0

测试容器：
http://192.168.198.154:8080/
```

多次构建后，积累的无用镜像

构建多次后，本地会遗留多个名为，tag也是的镜像。 这些都是上一次构建的结果，在经历了新一轮的构建后，其镜像名和tag被新镜像所有，所以自身只能显示名为，tag也是，清理这些镜像的命令是docker image prune，然后根据提示输入"y"，镜像即可被清理：

```
docker image prune
提示信息如下
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Deleted Images:

```

jenkins整合jib

gitlab服务器

1. gitlab-152服务器上创建jibdemo项目。
2. 使用idea开发工具将jibdemo项目上传gitlab服务器。

jenkins服务器

1. jenkinsmaster-153创建pipeline-test04任务

Jenkinsfile文件

编写步骤

1. 环境检测：检测jenkinsagent-154节点基础软件运行情况
2. 拉取代码：从gitlab-152服务器拉取jildemo项目
3. 编译构建：jenkinsagent-154执行maven命令；使用jib插件声明周期push镜像至harbor-155服务器
4. 删除容器：删除jenkinsagent-154服务器jibdemo容器
5. 删除镜像：删除jenkinsagent-154服务器jibdemo镜像
6. 登录harbor：docker登录harbor-155服务器
7. 拉取镜像：拉取jibdemo镜像
8. 运行容器：运行jibdemo容器

脚本骨架

```
pipeline {
   agent {
      label 'jenkinsagent-154'
   }
   stages {
       stage('检测环境') {
           steps {
               sh label: '', script: '''java -version
               mvn -v
               git version
               docker -v'''
           }
       }
       stage('拉取代码'){
            steps{
                echo 'gitlab拉取代码'
            }
       }
       stage('编译构建'){
           steps{
               echo '编译构建'
           }
       }
       stage('删除容器'){
          steps{
              echo '删除容器'
          }
       }
       stage('删除镜像'){
          steps{
             echo '删除镜像'
          }
       }
       stage('登录harbor'){
          steps{
            echo '登录harbor'
          }
       }
       stage('拉取镜像'){
```

```
            steps{
              echo '拉取镜像'
            }
        }
        stage('运行容器'){
            steps{
              echo '运行容器'
            }
        }
    }
}
```

测试pipeline任务

```
立即构建
```

拉取代码

```
        stage('拉取代码'){
             steps{
                 echo 'gitlab拉取代码'
                 checkout([$class: 'GitSCM', branches: [[name: '*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'c8634952-4993-4455-b164-35427823144f', url: 'ssh://git@192.168.198.152:222/lagou/jibdemo.git']]])
             }
        }
```

编译构建

```
        stage('编译构建'){
            steps{
                echo '编译构建'
                sh label: '', script: 'mvn clean package -Dmaven.test.skip=true jib:build -DsendCredentialsOverHttp=true'
            }
        }
```

删除容器

```
        stage('删除容器'){
           steps{
               sh label: '', script: '''echo \'检查容器是否存在\'
               containerid=`docker ps -a | grep -w jibdemo | awk \'{print $1}\'`
               if [ "$containerid" != "" ];then
                 echo '容器存在，停止容器'
                 docker stop $containerid
                 echo '删除容器'
                 docker rm $containerid
               fi'''
           }
        }
```

删除镜像

```
        stage('删除镜像'){
           steps{
              sh label: '', script: '''echo \'检查镜像是否存在\'
                 imageid=`docker images | grep jibdemo | awk \'{print $3}\'`
                 if [ "$imageid" != "" ];then

                   echo \'删除镜像\'
                   docker rmi -f  $imageid
                 fi'''
           }
        }
```

登录harbor

```
       stage('登录harbor'){
          steps{
            echo '登录harbor'
            sh label: '', script: 'docker login -u admin -p Harbor12345 192.168.198.155:5000'
          }
       }
```

拉取镜像

```
        stage('拉取镜像'){
          steps{
            echo '拉取镜像'
            sh label: '', script: 'docker pull 192.168.198.155:5000/lagouedu/jibdemo:1.0'
          }
       }
```

运行容器

```
        stage('运行容器'){
          steps{
            echo '运行容器'
            sh label: '', script: 'docker run -itd --name jibdemo -p
8080:8080 192.168.198.155:5000/lagouedu/jibdemo:1.0'
          }
       }
```

完整Jenkinsfile文件

```
pipeline {
   agent {
      label 'jenkinsagent-154'
   }
   stages {
       stage('检测环境') {
           steps {
               sh label: '', script: '''java -version
               mvn -v
               git version
               docker -v'''
           }
       }
       stage('拉取代码'){
            steps{
                echo 'gitlab拉取代码'
                checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: 'c8634952-4993-4455-b164-35427823144f',
url: 'ssh://git@192.168.198.152:222/lagou/jibdemo.git']]])
            }
       }
       stage('编译构建'){
           steps{
               echo '编译构建'
               sh label: '', script: 'mvn clean package -
Dmaven.test.skip=true jib:build -DsendCredentialsOverHttp=true'
           }
       }
       stage('删除容器'){
          steps{
              sh label: '', script: '''echo \'检查容器是否存在\'
              containerid=`docker ps -a | grep -w jibdemo | awk \'{print
$1}\'`
              if [ "$containerid" != "" ];then
                echo '容器存在，停止容器'
                docker stop $containerid
                echo '删除容器'
                docker rm $containerid
              fi'''
          }
```

```
        }
        stage('删除镜像'){
           steps{
              sh label: '', script: '''echo \'检查镜像是否存在\'
                 imageid=`docker images | grep jibdemo | awk \'{print $3}\'`
                 if [ "$imageid" != "" ];then

                   echo \'删除镜像\'
                   docker rmi -f  $imageid
                 fi'''
           }
        }
        stage('登录harbor'){
           steps{
             echo '登录harbor'
             sh label: '', script: 'docker login -u admin -p Harbor12345
192.168.198.155:5000'
           }
        }
        stage('拉取镜像'){
           steps{
             echo '拉取镜像'
             sh label: '', script: 'docker pull
192.168.198.155:5000/lagouedu/jibdemo:1.0'
           }
        }
        stage('运行容器'){
           steps{
             echo '运行容器'
             sh label: '', script: 'docker run -itd --name jibdemo -p
8080:8080 192.168.198.155:5000/lagouedu/jibdemo:1.0'
           }
        }
    }
}
```