

讲师(老司机)

# CI/CD、Devops

## 课程内容介绍

- CI/CD、Devops理论概述
- 软件生命周期概述
- 手工部署微服项目
- jenkins各种操作系统安装
- jenkins中文社区简介
- 自定义中文版jenkins镜像
- jenkins插件正确安装方式
- 容器化部署jenkins镜像
- jenkins分布式部署
- 自动风格项目介绍
- pipeline项目详解
- jenkins项目实战
- jenkins运维管理
- groovy语言入门
- jenkins共享库

## CI/CD、Devops概述

### 为什么需要CI/CD

英文版:

<https://dzone.com/articles/why-the-world-needs-cicd>

中文版:

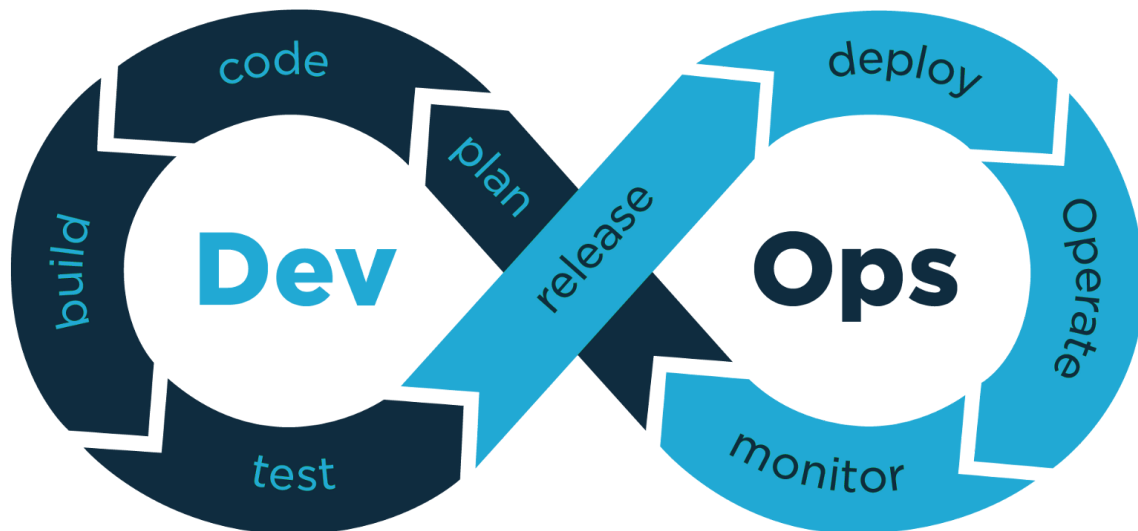
<https://www.kubernetes.org.cn/8640.html>

持续集成（CI）是一种**开发实践**，其中开发人员经常（最好每天几次）将代码集成到共享存储库中。然后可以通过自动构建和自动测试来验证每个集成。尽管自动化测试不是严格意义上的CI的一部分，但通常隐含了它。

定期集成的主要好处之一是，您可以快速检测到错误并更轻松地定位它们。由于引入的每个更改通常很小，因此可以快速查明引入缺陷的特定更改。

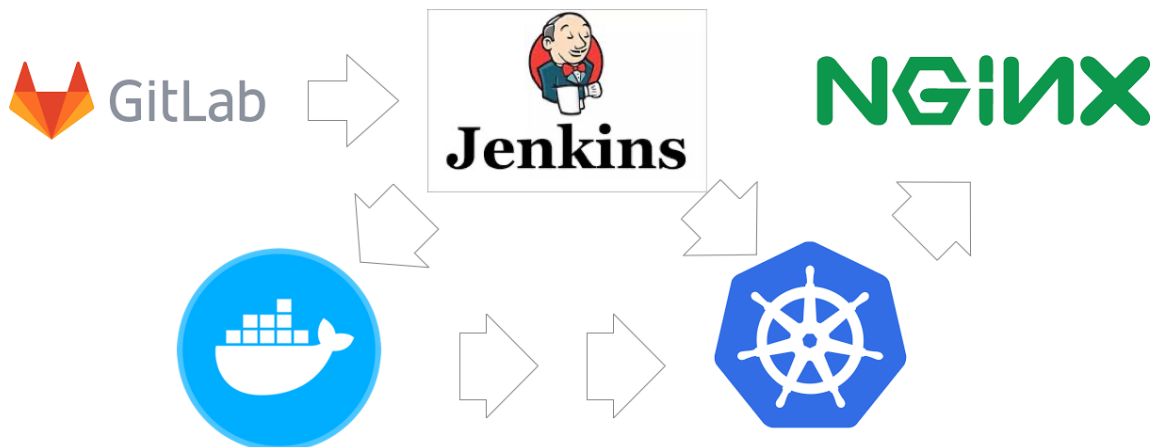
近年来，CI已成为软件开发的最佳实践，并遵循一系列关键原则。其中包括版本控制，构建自动化和自动化测试。

此外，持续部署和持续交付已成为**最佳实践**，可让您随时随地部署应用程序，甚至在每次引入新更改时甚至将主代码库自动推入生产环境。这使您的团队可以快速行动，同时保持可以自动检查的高质量标准。



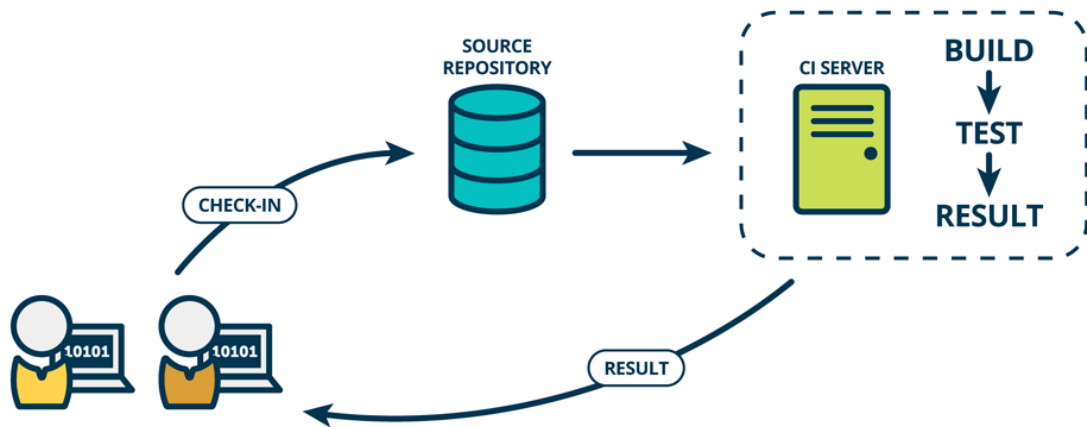
CI/CD应用场景：

- 开发人员将本地代码上传gitlab版本服务器
- jenkins通过webhook插件自动到gitlab服务器拉取最新代码
- 通过docker-maven-plugin插件自动编译代码
- 将自定义镜像上传docker私服仓库
- k8s集群自动拉取最新版本镜像
- 自动化部署整个项目
- 用户通过nginx负载均衡访问整个项目



## 持续集成（CI）

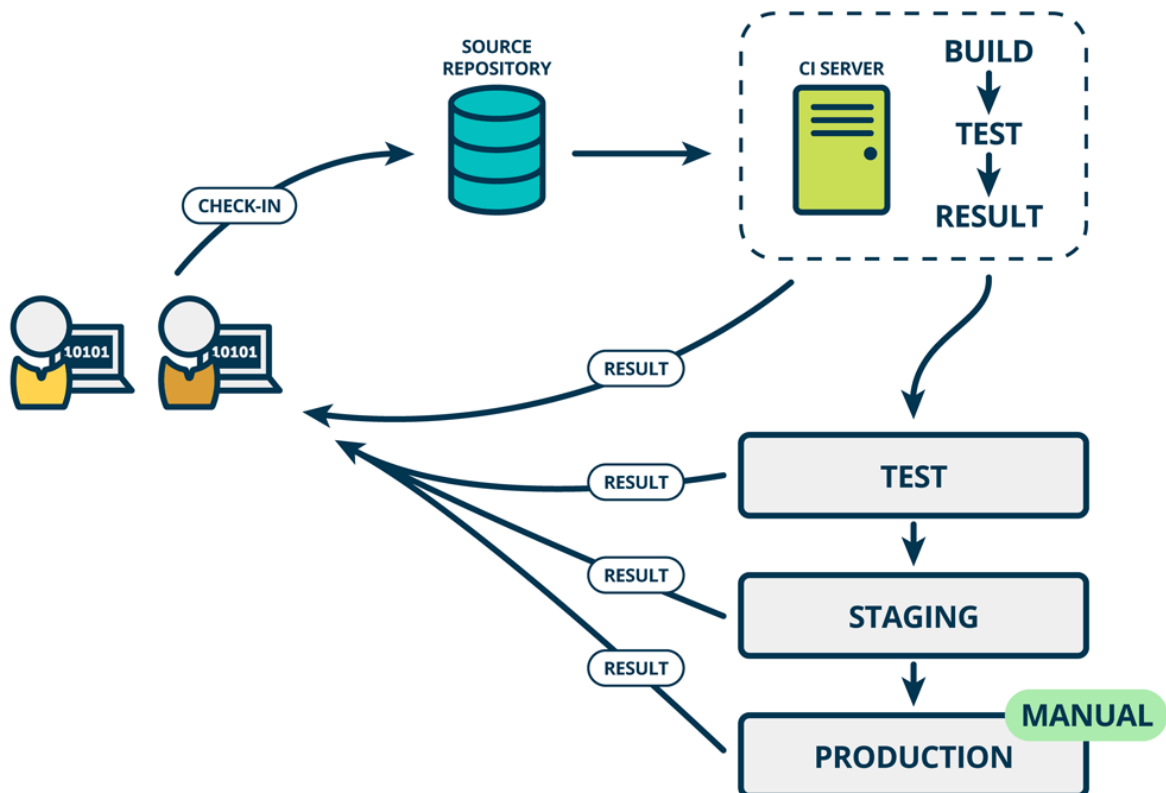
在传统的软件开发中，集成过程通常在每个人完成工作后的项目结束时进行。整合通常需要数周或数月的时间，可能会非常痛苦。持续集成是一种将集成阶段置于开发周期中较早的做法，因此，构建，测试和集成代码的时间安排更为规则。



开发人员通常使用称为CI Server的工具来进行构建和集成。CI要求自检代码。这是用于自我测试以确保其按预期工作的代码，这些测试通常称为单元测试。集成代码后，当所有单元测试通过时，将得到一个最新的代码版本。这表明他们已经验证了自己的更改已成功集成到一起，并且代码按测试期望的那样工作。

## 连续交付（CD）

持续交付意味着每次更改代码，集成并构建代码时，他们还将与生产非常相似的环境中自动测试该代码。我们将此部署到不同环境并在不同环境上进行测试的过程称为部署管道。部署管道通常具有开发环境，测试环境和过渡环境，但是这些阶段因团队，产品和组织而异。

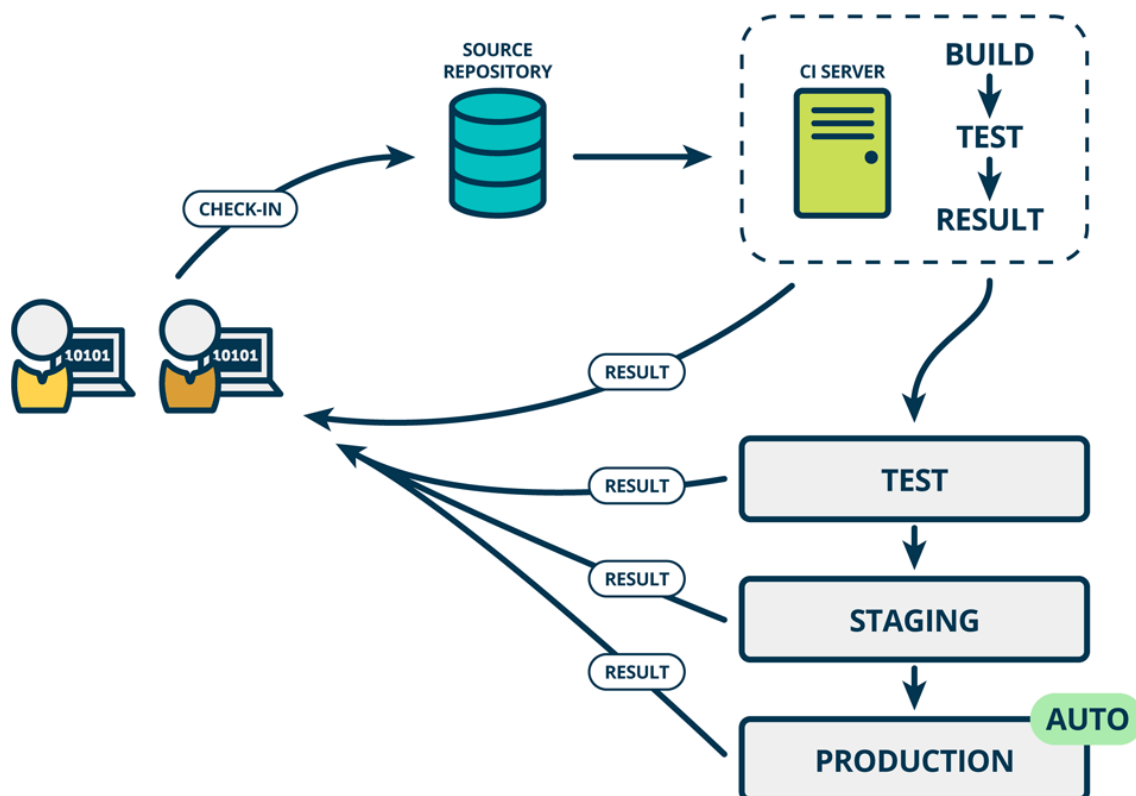


在每个不同的环境中，开发人员编写的代码都经过不同的测试。当代码在生产环境中部署时，它们将在生产环境中工作。至关重要的是，代码只提升到（上测试）在部署管道的下一个环境，如果出现故障，整个团队可以更轻松地了解问题可能出在哪里，并在代码进入生产环境之前予以解决。这个过程对我们这个行业的人来说非常强大。这意味着，如果代码的测试在所有环境中都通过了，团队负责人就会知道

开发人员的代码在投入生产时可能会按预期工作。一旦测试在所有环境中通过，团队负责人就可以立即决定最终用户是否通过测试。我们现在要在生产中使用这种绿色产品吗？是！因此，一旦开发人员完成构建，便可以立即为客户提供经过全面测试的全新工作软件。！

## 持续部署

在这种实践中，团队负责人所做的每一项更改都通过了所有测试阶段，并自动投入生产。要实现连续部署，团队负责人首先需要进行连续交付，因此在开始练习连续部署之前，先决定哪个对您更合适，持续交付都是为了增强整个业务的能力，因此至少您应该参与确定是否应该使用持续部署。



### 持续集成，持续部署和持续交付之间有什么区别？

- 持续集成 (Continuous integration)  
这种做法是将团队中不同开发人员的变更尽早集成到主线中，最好每天进行几次。这样可以确保各个开发人员处理的代码不会转移太多。当您将流程与自动化测试结合在一起时，持续集成可以使您的代码变得可靠。
- 持续交付 (Continuous delivery)  
保持代码库随时可部署的做法。除了确保您的应用程序通过自动化测试外，它还必须具有将其投入生产所需的所有配置。然后，许多团队会进行推送更改，以立即将自动化测试传递到测试或生产环境中，以确保快速的开发周期。
- 持续部署 (continuous deployment)  
与持续集成密切相关，是指保持您的应用程序可随时部署，甚至在最新版本通过所有自动化测试的情况下，甚至自动发布到测试或生产环境。

# CONTINUOUS DELIVERY



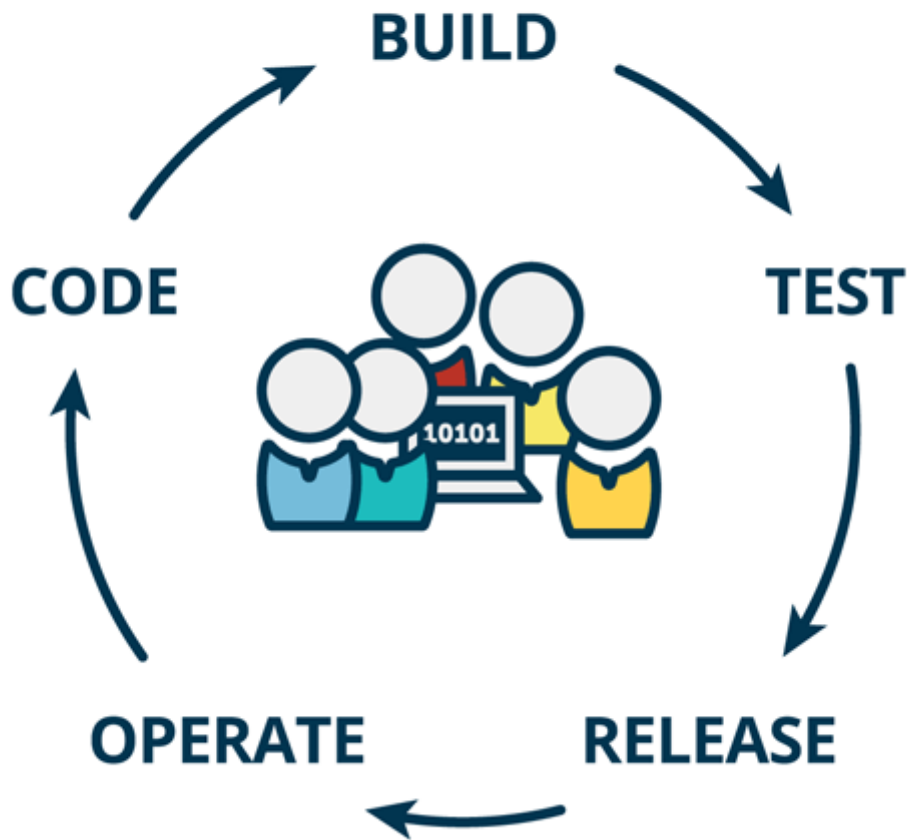
# CONTINUOUS DEPLOYMENT



## DevOps概述

“DevOps”一词来自“Development”和“Operations”的组合。DevOps是一种文化，可促进开发人员与其他技术专业人员之间的协作，通常称呼是运维。具体来说，是在软件交付和部署过程中进行通信和协作，目的是更快，更可靠地发布质量更好的软件。具有所谓DevOps文化的组织的共同特征是：自主的多技能团队，高水平的测试和发布自动化（连续交付）以及两者之间的共同目标多技术成员。

# NEW WAY:



DevOps文化通常与持续交付相关联，因为它们都旨在增强开发人员和运营团队之间的协作，并且都使用自动流程来更快，更频繁，更可靠地构建，测试和发布软件。这些都是像我们这样的人想要的东西。尽管开发团队经常看到流程改进的最直接好处，但CI，CD和DevOps对我们其他人来说却有很多好处。简而言之，我相信实践CD并拥护DevOps文化的组织将更频繁地向其客户提供更有价值，更可靠的软件。

## 常用术语

- 签到：将本地开发代码推送到公共源存储库的过程。
- CI服务器：用于构建和测试源代码的工具。CI服务器将告知开发人员最新的代码构建是否成功以及是否继续通过测试。
- 开发环境：开发人员在哪里创建，集成，构建和测试代码。
- 部署管道/管道：这是开发人员代码更改在完成并准备交付生产之前要经历的一组阶段。通常，这些将是“构建”，“单元测试”，“功能测试”，“性能测试”和“部署”。不同的自动化测试将在不同的阶段运行。一旦代码通过了整个部署管道，就可以将软件交付生产。
- 迭代开发：迭代开发是一次构建一点产品并对其进行完善直到完成的地方。该产品是迭代构建的，每次迭代都对相同的部分进行重新加工。预期并计划在不同迭代中的功能之间进行更改。您可以将CI，持续交付或持续部署与迭代开发结合使用
- 生产环境：这是部署或发布软件的地方。使用您的产品或网站的客户最有可能使用此环境。也称为“生产中”，“生产中”或“实时”。

- 单元测试：单元测试是代码中的自动化测试，用于测试低级的单个代码段，以确保它们可用并按预期工作。单元测试被认为是实践CI和CD的前提条件。

# Jenkins概述与安装

---

## 软件开发生命周期

软件开发生命周期又叫做SDLC（Software Development Life Cycle），它是集合了需求分析、设计、开发、测试和部署等过程的集合。

软件生命周期又称为软件生存周期或系统开发生命周期，是软件的产生直到报废的生命周期，周期内有问题定义、可行性分析、总体描述、系统设计、编码、调试和测试、验收与运行、维护升级到废弃等阶段，这种按时间分程的思想方法是软件工程中的一种思想原则，即按部就班、逐步推进，每个阶段都要有定义、工作、审查、形成文档以供交流或备查，以提高软件的质量。但随着新的面向对象的设计方法和技术的成熟，软件生命周期设计方法的指导意义正在逐步减少。

生命周期的每一个周期都有确定的任务，并产生一定规格的文档（资料），提交给下一个周期作为继续工作的依据。按照软件的生命周期，软件的开发不再只单单强调“编码”，而是概括了软件开发的全过程。软件工程要求每一周期工作的开始只能必须是建立在前一个周期结果“正确”前提上的延续；因此，每一周期都是按“活动-结果-审核-再活动-直至结果正确”循环往复进展的。

大致分为如下各个阶段：

- 需求分析

这是生命周期的第一阶段，根据项目需求，团队执行一个可行性计划的分析。项目需求可能是公司内部或者客户提出的。这阶段主要是对信息的收集，也有可能是对现有项目的改善和重新做一个新的项目。还要分析项目的预算多长，可以从哪方面受益及布局，这也是项目创建的目标。

- 设计

第二阶段就是设计阶段，系统架构和满意状态（就是要做成什么样子，有什么功能），和创建一个项目计划。计划可以使用图表，布局设计或者文者的方式呈现。

- 实现

第三阶段就是实现阶段，项目经理创建和分配工作给开者，开发者根据任务和在设计阶段定义的目标进行开发代码。依据项目的大小和复杂程度，可以需要数月或更长时间才能完成。

- 测试

测试人员进行代码测试，包括功能测试、代码测试、压力测试等。在软件设计完成后要经过严密的测试，以发现软件在整个设计过程中存在的问题并加以纠正。

- 运行及维护

最后阶段就是对产品不断的进化改进和维护阶段，根据用户的使用情况，可能需要对某功能进行修改，bug修复，功能增加等。

# 为什么需要jenkins

## 服务器规划

生成至少三台虚拟机

主机名	主机IP
harbor-155	192.168.198.155
jenkins-153	192.168.198.153
gitlab-152	192.168.198.152
jenkinsagent-151	192.168.198.151

## 微服项目

编写一个简单的springboo项目模拟开发环境要部署的微服项目。

### 创建项目

springboot/jenkinsdemo

### 控制器

```
@RestController
public class JenkinsTestController {

    @GetMapping("/")
    public String login() {
        return "hello jenkins!!!";
    }
}
```

### 打包插件

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



## 本地运行微服项目

```
cmd
mvn clean package -Dmaven.test.skip=true

java -jar jenkinsdemo-0.0.1-SNAPSHOT.jar

将jar上传到linux服务器
```

## 自定义jar包名称

在pom.xml文件中增加配置finalName。打包时可以自定义jar包名称。

```
<build>
  <finalName>jenkinsdemo</finalName>
  <plugins>
    ....
  </plugins>
</build>
```

## dockerfile

### 基础镜像

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

### 安装docker插件

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
```

### 制作镜像-dockerfile

#### *jenkins/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
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

## 生成测试镜像

```
docker build --rm -t lagou/jenkinsdemo:v1 --build-arg JAR_FILE=jenkinsdemo.jar .
```

## 测试、删除镜像

```
docker run -itd --name=jenkinsdemo -p 8080:8080 lagou/jenkinsdemo:v1

docker ps | grep jenkins

docker logs -f jenkinsdemo

http://192.168.198.154:8080

docker stop jenkinsdemo

docker rm jenkinsdemo
```

## harobor私服

### 推送镜像

登录私服

```
vi /etc/docker/daemon.json
```

```
"insecure-registries":["192.168.198.155:5000"]
```

```
systemctl daemon-reload
```

```
systemctl restart docker
```

并上传镜像

```
docker tag lagou/jenkinsdemo:v1 192.168.198.155:5000/lagouedu/jenkinsdemo:v1
```

```
docker push 192.168.198.155:5000/lagouedu/jenkinsdemo:v1
```

```
docker rmi -f lagou/jenkinsdemo:v1
```

## 部署项目

### K8S部署镜像

#### jenkins/jenkinsdemo-deployment.yml清单

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jenkinsdemo-deploy
  labels:
    app: jenkinsdemo-deploy
spec:
  replicas: 1
  template:
    metadata:
      name: jenkinsdemo-deploy
      labels:
        app: jenkinsdemo-deploy
    spec:
      containers:
        - name: jenkinsdemo-deploy
          image: 192.168.198.155:5000/lagouedu/jenkinsdemo:v1
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
          restartPolicy: Always
      selector:
        matchLabels:
          app: jenkinsdemo-deploy
---
apiVersion: v1
kind: Service
metadata:
  name: jenkinsdemo-svc
spec:
  selector:
    app: jenkinsdemo-deploy
```

```
ports:
  - port: 8888
    protocol: TCP
    targetPort: 8080
    nodePort: 30099
  type: NodePort
```

## 执行命令

```
kubectl apply -f jenkinsdemo-deployment.yml
kubectl get pods
```

进入容器，查看容器时间是否是东八区时间

```
kubectl exec -it jenkinsdemo-deploy-68785fcf7d-stjbz sh
date
exit
```

```
kubectl delete -f jenkinsdemo-deployment.yml
```

## jenkins安装

### jenkins官网

<https://www.jenkins.io/zh/>

官方文档

<https://www.jenkins.io/zh/doc/>

## window系统

### 自动部署

下载官网提供的msi文件即可

<https://www.jenkins.io/download/thank-you-downloading-windows-installer-stable/>

### 手动部署一

```
java -jar jenkins.war
```

## 手动部署二

1. 部署tomcat9
2. 在官网下载jenkins.war  
`http://mirrors.jenkins.io/war-stable/2.235.5/jenkins.war`
3. 将war包放入apache-tomcat-9.0.34\webapps\目录中
4. 启动tomcat  
cmd  
`cd apache-tomcat-9.0.34\bin\`  
执行 `startup.bat` 启动tomcat
5. 访问jenkins  
`http://localhost:8080/jenkins`

## linux系统

### 安装jenkins

1. 安装JDK8  
`yum install java-1.8.0-openjdk* -y`
2. yum方式安装  
`wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/redhat-stable/jenkins.repo`  
  
`rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io.key`  
  
`yum install jenkins`
3. 或者采用rpm方式安装：在清华大学镜像源下载jenkins的rpm文件：`jenkins-2.253-1.1.noarch.rpm`  
`https://mirrors.tuna.tsinghua.edu.cn/jenkins/redhat/`  
  
上传centos服务器进行安装：  
`rpm -ivh jenkins-2.190.3-1.1.noarch.rpm`

### 配置jenkins

1. 修改Jenkins配置  
`vi /etc/sysconfig/jenkins`  
  
修改内容如下：  
`JENKINS_USER="root"`  
`JENKINS_PORT="8888"`

```
2.启动Jenkins服务
systemctl start jenkins
systemctl enable jenkins

3.打开浏览器访问
http://192.168.198.153:8888
```

## 获取密码

```
获取并输入admin账户密码
cat /var/lib/jenkins/secrets/initialAdminPassword
```

## 容器化安装

### 基础镜像最新版

```
docker pull jenkins/jenkins:2.235.5-alpine 355MB

docker pull jenkins/jenkins:2.253-centos7 599MB
```

### 推荐使用镜像版本

```
docker pull jenkins/jenkins:2.204.5 617MB

docker pull jenkins/jenkins:2.204.5-alpine 224MB
```

## 运行镜像

```
docker run -itd --name jenkins -p 8080:8080 -p 50000:50000 -u root --
restart=always jenkins/jenkins:2.204.5-alpine

查看容器启动日志并找到jenkins初始化密码
docker logs -f jenkins
```

## 浏览器测试

```
http://192.168.198.153:8080
```

## 部署映射目录

给映射目录授权

```
mkdir -p /data/jenkins && chown -R 1000:1000 /data/jenkins
```

运行容器

```
docker run -itd --name jenkins -p 8080:8080 -p 50000:50000 -u root -e  
JAVA_OPTS=-Duser.timezone=Asia/Shanghai --restart=always -v  
/data/jenkins:/var/jenkins_home/ jenkins/jenkins:2.204.5-alpine
```

查看容器启动日志并找到jenkins初始化密码

```
docker logs -f jenkins
```

## 中文社区

### 官网地址

官网地址:

<https://bintray.com/jenkins-zh/generic/jenkins>

docker官网地址:

<https://hub.docker.com/r/jenkinszh/jenkins-zh>

github官网地址:

<https://github.com/jenkins-zh/jenkins-formulas>

## window系统部署

```
java -jar jenkins-zh.war
```

## 基础镜像

中文社区提供的镜像体积很庞大。

最新版本:

docker pull jenkinszh/jenkins-zh:2.235      718MB

推荐版本:

docker pull jenkinszh/jenkins-zh:2.204.5      682MB

## 安装jenkins

给映射目录授权

```
mkdir -p /data/jenkins && chown -R 1000:1000 /data/jenkins
```

```
docker run -itd --name jenkins -p 8080:8080 -p 50000:50000 -u root --restart=always jenkinszh/jenkins-zh:2.204.5
```

```
docker run -itd --name jenkins -p 8080:8080 -p 50000:50000 -u root -e JAVA_OPTS=-Duser.timezone=Asia/Shanghai --restart=always -v /data/jenkins:/var/jenkins_home jenkinszh/jenkins-zh:2.235
```

## 制作jenkins镜像

### 制作步骤

制作步骤：

1. 编写Dockerfile文件。
  2. 从下一步骤开始。所有文件都在jenkins中文社区寻找。但是不能在windows系统创建。因为系统文件格式不同。jenkins启动时会报错。
  3. 创建init.groovy文件。
  4. 创建hudson.model.UpdateCenter.xml
  5. 创建mirror-adapter.crt
  6. 制作镜像
  7. 启动容器，试运行镜像
  8. jenkins工作台操作，检验插件下载速度
  9. 停止容器
  10. 删除容器
- 备份镜像

### 基础镜像

```
docker pull jenkins/jenkins:2.204.5-alpine
```

## Dockerfile

在基础镜像中增加时区修正。安装常用软件。更新jenkins插件为中文社区地址

```
FROM jenkins/jenkins:2.204.5-alpine
#自定义jenkins镜像
# 作者信息
MAINTAINER war kubernet jenkins "admin@lagou.com"
# 修改源
USER root
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
ENV JENKINS_UC https://updates.jenkins-zh.cn
ENV JENKINS_UC_DOWNLOAD https://mirrors.tuna.tsinghua.edu.cn/jenkins
ENV JENKINS_OPTS="-
Dhudson.model.UpdateCenter.updateCenterUrl=https://updates.jenkins-zh.cn/update-
center.json"
ENV JENKINS_OPTS="-Djenkins.install.runSetupWizard=false"
COPY init.groovy /usr/share/jenkins/ref/init.groovy.d/init.groovy
COPY hudson.model.UpdateCenter.xml
/usr/share/jenkins/ref/hudson.model.UpdateCenter.xml
COPY mirror-adapter.crt /usr/share/jenkins/ref/mirror-adapter.crt

```

## init.groovy

在github的jenkins中文社区中找到文件，复制文件内容

```

import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.net.URL;
import hudson.init.InitMilestone;
import jenkins.model.Jenkins;

Thread.start {
    while(true) {
        Jenkins instance = Jenkins.getInstance();
        InitMilestone initLevel = instance.getInitLevel();
        Thread.sleep(1500L);
        println "Jenkins not ready when handle init config..."
        if (initLevel >= InitMilestone.PLUGINS_STARTED) {
            InputStream input = new FileInputStream("/usr/share/jenkins/ref/mirror-
adapter.crt")
            FileOutputStream out = new FileOutputStream(System.getenv("JENKINS_HOME")
+ "/war/WEB-INF/update-center-rootCAs/mirror-adapter.crt");
            byte[] buf = new byte[1024];
            int count = -1;

            while((count = input.read(buf)) > 0) {
                out.write(buf, 0, count);
            }
            println "Jenkins init ready..."
            break
        }
    }
}

```

## hudson.model.UpdateCenter.xml

在github的jenkins中文社区中找到文件，复制文件内容。更换插件安装地址为国内插件地址

```
<?xml version='1.1' encoding='UTF-8'?>
<sites>
  <site>
    <id>default</id>
    <url>https://updates.jenkins-zh.cn/update-center.json</url>
  </site>
</sites>
```

## mirror-adapter.crt

在github的jenkins中文社区中找到文件，复制文件内容。jenkins官方配置2对秘钥，一对是jenkins官网使用。这一对是jenkins中文社区使用。更新秘钥后。下载地址更新为国内地址。

```
-----BEGIN CERTIFICATE-----
MIICCTCCAdoCCQD/jZ7AgrzJKTANBgkqhkiG9w0BAQsFADB9MQswCQYDVQQGEWJD
TjELMAkGA1UECAwCR0QxCzAJBgNVBACMA1NaMQ4wDAYDVQQKDAV2aWhvbzEMMAoG
A1UECwwDZGV2MREwDwYDVQQDDAhkZW1vLmNvbTEjMCEGCSqGSIb3DQEJARYUYWRt
aw5AamVua2lucy16aC5jb20wHhcNMjxMTA5MTA0MDA5WhcNMjIxMTA4MTA0MDA5
WjB9MQswCQYDVQQGEWJDjTjELMAkGA1UECAwCR0QxCzAJBgNVBACMA1NaMQ4wDAYD
VQQKDAV2aWhvbzEMMAoGA1UECwwDZGV2MREwDwYDVQQDDAhkZW1vLmNvbTEjMCEG
CSqGSIb3DQEJARYUYWRtaW5AamVua2lucy16aC5jb20wgZ8wDQYJKoZIhvcNAQEB
BQADgY0AMIGJAoGBAN+6jn8rCIjvkQ0Q7ZbJLk4IdcHor2WdskoQMh1br0goyb4g
RX+CorjDRjDm6mj2Oohq1rtRxLGyxBnXFeQGU7wwjQHyfKDghtP51G/6721XFtzB
KXukHBYHjtzrDxAutKTdolyBCuIDDGJmRk+LavIBY3/Lxh6f0ZQSeCSjYiyxAgMB
AAEwDQYJKoZIhvcNAQELBQADgYEAD92126PoJcb19GojK2L3pyOQjeeDm/vV9e3R
EgwGmoIQz1ubM0mjxpCz1J73nesoAcuplTEps/46L7yoMjptCA3TU9FZAHNQ8dbz
a0vm4CF9841/Fik8tsLtwCT6ivkAi01XGwhX0FK7FaAyU0nNeo/EPvDwzTim4XDK
9j1WgPE=
-----END CERTIFICATE-----
```

## 制作镜像

```
docker build --rm -t lagou/jenkins:2.204.5-alpine .
```

## 启动容器

创建映射目录

```
mkdir -p /data/jenkins && chown -R 1000:1000 /data/jenkins
```

运行容器

```
docker run -itd --name jenkins -p 8080:8080 -p 50000:50000 -u root -e  
JAVA_OPTS=-Duser.timezone=Asia/Shanghai --restart=always -v  
/data/jenkins:/var/jenkins_home/ lagou/jenkins:2.204.6-alpine
```

查看容器启动日志并找到jenkins初始化密码

```
docker logs -f jenkins
```

## 测试镜像

```
http://192.168.198.153:8080
```

## 安装插件

安装jenkins默认插件，观察下载速度。

## 删除镜像

停止容器

```
docker stop jenkins
```

删除容器

```
docker rm jenkins
```

备份镜像

```
docker save lagou/jenkins:2.204.5-alpine -o lagou.jenkins:2.204.5-alpine.tar
```

或者将镜像上传harbor私服

## jenkins插件离线安装

Jenkins 社区的网络基础设施都是架设在国外的服务器上，而且，并没有在国内有 CDN 或者负载均衡的配置。对所有的 Jenkins 用户而言，1500+的插件可以帮助他们解决很多问题。然而，我相信，对于国内的很多用户来说，可能有过一些不太愉快的经历——插件下载速度很慢，甚至会超时。难道遇到这种情况下，我们就只能等吗？

程序员，作为天生懒惰的人，总是希望能通过手中的键盘来解决各种个样的问题。凭什么？下载一个插件，我还苦苦地等待来自美国的数据包呢？数数你手里的 Jenkins 都安装了多少个插件。30个算少的吧。经过一番搜索，发现果然已经有前人帮忙把大树种好了。让我们一起感谢“清华大学开源软件镜像站”提供的镜像服务

<https://mirrors.tuna.tsinghua.edu.cn/jenkins/>

## Jenkins环境搭建

### 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?](https://www.oracle.com/webapps/redirect/signon?nexturl=https://download.oracle.com/otn/java/jdk/8u261-b12/a4634525489241b9a9e1aa73d9e118e6/jdk-8u261-linux-x64.tar.gz)

[nexturl=https://download.oracle.com/otn/java/jdk/8u261-](https://download.oracle.com/otn/java/jdk/8u261-b12/a4634525489241b9a9e1aa73d9e118e6/jdk-8u261-linux-x64.tar.gz)

[b12/a4634525489241b9a9e1aa73d9e118e6/jdk-8u261-linux-x64.tar.gz](https://download.oracle.com/otn/java/jdk/8u261-b12/a4634525489241b9a9e1aa73d9e118e6/jdk-8u261-linux-x64.tar.gz)

```
tar -zxvf 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 -zxvf 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

```
生成密钥
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vmware快照。如果出错了。可以快速恢复快照版本信息。

### 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/e16/>

在线下载安装包：

`wget` [https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/e16/gitlab-ce-12.7.6-ce.0.e16.x86\\_64.rpm](https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/e16/gitlab-ce-12.7.6-ce.0.e16.x86_64.rpm)

安装：

```
rpm -i gitlab-ce-12.7.6-ce.0.e16.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

```
gitlabctl reconfigure  
gitlabctl 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
```

```
gitlabctl reconfigure
gitlabctl restart
gitlabctl 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。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可接受交互式输入，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

```
gitlab->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

gitlab-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. gitlab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>

# Jenkins项目实战

## 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，ENTRYPOINT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, 可能由源代码产生一个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, 由源代码产生的是一个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

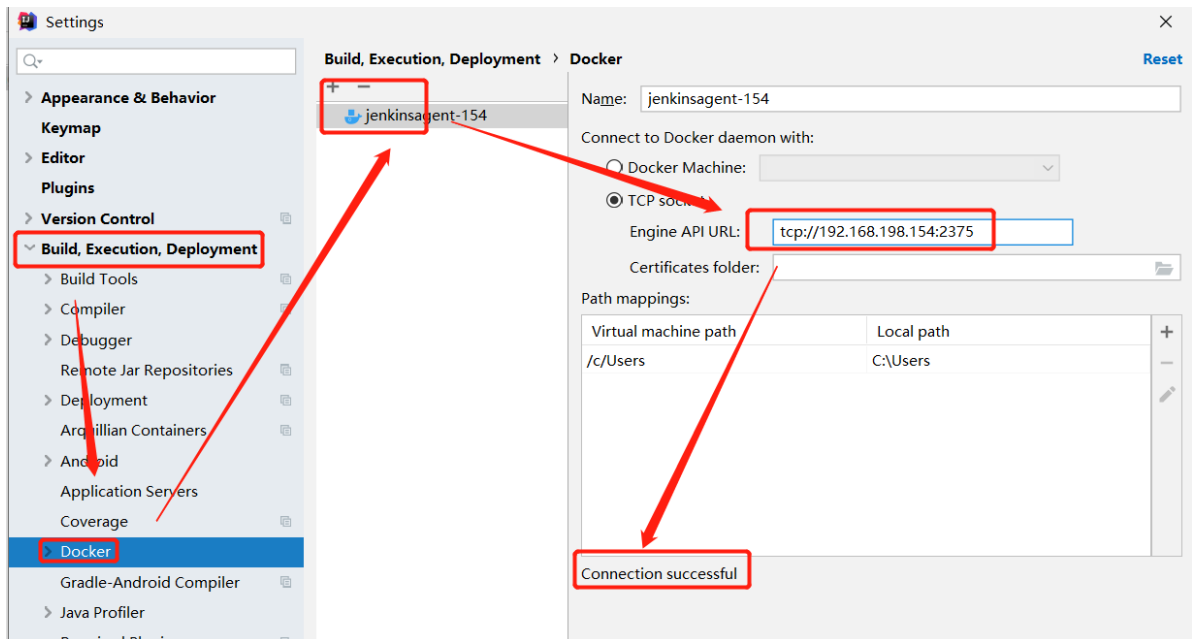
配置内容如下:

name:jenkinsagent-154

TCP Socket:

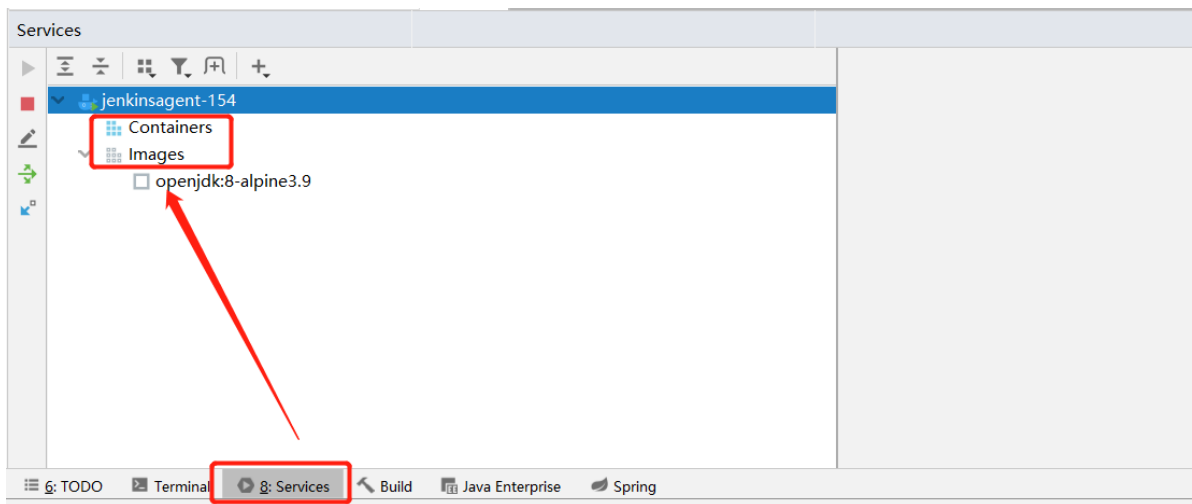
Engine API URL:tcp://192.168.198.154:2375

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

```

@RestController
public class JenkinsDemoController {

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

```

## 打包部署

```
idea在terminal窗口中运行如下命令  
mvn clean package -Dmaven.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

```
<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Dockerfile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>
          </auth>
        </from>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```

        <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

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jibdemo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'
    }
}
}







```

```
}  
}
```

# Jenkins运维管理



## 凭据管理

 **凭据**

类型	提供者	存储 ↓	域	唯一标识	名称
	 Jenkins	全局	58673430-367a-4983-b156-88c668abe813	<a href="#">root (jenkinsagent-154-SSH)</a>	
	 Jenkins	全局	b26bd84e-e0cb-4b90-8469-1c2a46213466	<a href="#">root/***** (gitlab-154-http)</a>	
	 Jenkins	全局	c8634952-4993-4455-b164-35427823144f	<a href="#">root (gitlab-154-SSH)</a>	

图标: [小](#) [中](#) [大](#)

Stores scoped to **Jenkins**

提供者	存储 ↓	域
 Jenkins		 全局

## 插件管理

可更新 可选插件 **已安装** 高级

启用

名称 ↓

版本

上一个安装的版本

卸载

<input checked="" type="checkbox"/>	<a href="#">Apache HttpComponents Client 4.x API Plugin</a> Bundles Apache HttpComponents Client 4.x and allows it to be used by Jenkins plugins.	<a href="#">4.5.10-2.0</a>		<a href="#">卸载</a>
<input checked="" type="checkbox"/>	<a href="#">bouncycastle API Plugin</a> This plugin provides an stable API to Bouncy Castle related tasks.	<a href="#">2.18</a>		<a href="#">卸载</a>
<input checked="" type="checkbox"/>	<a href="#">Branch API Plugin</a> This plugin provides an API for multiple branch based projects.	<a href="#">2.5.9</a>		<a href="#">卸载</a>
<input checked="" type="checkbox"/>	<a href="#">Build Timeout</a> This plugin allows builds to be automatically terminated after the specified amount of time has elapsed.	<a href="#">1.19.1</a>		<a href="#">卸载</a>
<input checked="" type="checkbox"/>	<a href="#">Command Agent Launcher Plugin</a> Allows agents to be launched using a specified command.	<a href="#">1.4</a>		<a href="#">卸载</a>
<input checked="" type="checkbox"/>	<a href="#">Credentials Binding Plugin</a> Allows credentials to be bound to environment variables for use from miscellaneous build steps.	<a href="#">1.23</a>		<a href="#">卸载</a>
<input checked="" type="checkbox"/>	<a href="#">Credentials Plugin</a> This plugin allows you to store credentials in Jenkins.	<a href="#">2.3.11</a>		<a href="#">卸载</a>
<input checked="" type="checkbox"/>	<a href="#">Display URL API</a>	<a href="#">2.2.2</a>		<a href="#">卸载</a>

## 用户管理

### 创建用户

1. devmanager
2. testmanager

 查看版本以及证书信息。

 **管理旧数据**  
从旧的、早期版本的插件中清理配置文件。

 **管理用户**  
创建、删除或修改 Jenkins 用户

 **准备关机**  
停止执行新的构建任务以安全关闭计算机。

## 用户权限管理

### 安装Role-based Authorization Strategy插件

利用Role-based Authorization Strategy 插件来管理jenkins用户权限

过滤:

名称	版本
<a href="#">CloudBees AWS Credentials</a>	1.21
Allows storing Amazon IAM credentials within the Jenkins Credentials API. Store Amazon IAM access keys (AWSAccessKeyId and AWSSecretKey) within the Jenkins Credentials API. Also support IAM Roles and IAM MFA token.	
<b>Role-based Authorization Strategy</b>	2.16
Enables user authorization using a Role-Based strategy. Roles can be defined globally or for particular jobs or nodes selected by regular expressions.	

直接安装 下载后重启安装 5分56秒之前获取了更新信息 立即获取

安装成功后重新启动jenkins。这两个用户创建成功后是没有任何权限的。登录jenkins也不能做任何事情。必须要给用户分配权限。

### 开启权限全局安全配置

授权策略切换为"Role-Based Strategy"，点击"保存"按钮

☐ Unix user/group database

授权策略

☒ **Role-Based Strategy**

☐ 任何用户可以做任何事(没有任何限制)

☐ 安全矩阵

☐ 登录用户可以做任何事

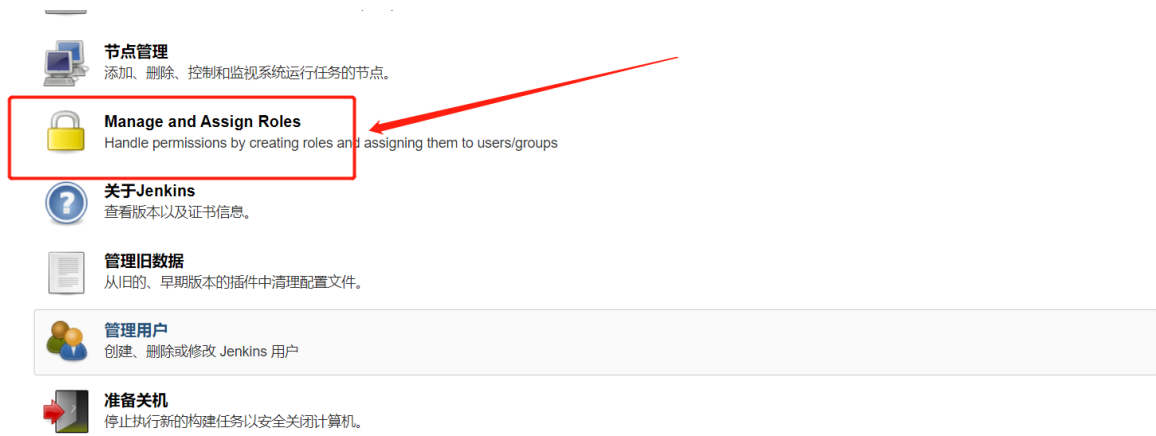
☐ 遗留模式

☐ 项目矩阵授权策略

器

### 创建角色

1. baserole: 用于登录jenkins
2. devrole: 管理所有lagoudev.\*开头的任务
3. testrole: 管理所有lagoutest.\*开头的任务



## jenkins默认角色

1. Global roles（全局角色）：管理员等高级用户可以创建基于全局的角色
2. Item roles（项目角色）：针对某个或者某些项目的角色
3. Node roles（节点角色）：节点相关的权限

## groovy入门

### 简介

Groovy 是运行在 JVM 中的另外一种语言，我们可以用 Groovy 在 Java 平台上进行编程，使用方式基本与使用 Java 代码的方式相同，所以如果你熟悉 Java 代码的话基本上不用花很多精力就可以掌握 Groovy 了，它的语法与 Java 语言的语法很相似，而且完成同样的功能基本上所需要的 Groovy 代码量会比 Java 的代码量少。

### 官网地址

```
http://groovy-lang.org/download.html
```

### 配置groovy

```
GROOVY_HOME:D:\groovy-3.0.4  
PATH: %GROOVY_HOME%\bin;
```

### 测试groovy环境

```
cmd  
groovy -v || groovy --version
```

## idea集成groovy

与创建JDK步骤类似。在创建工程时候配置groovy。

src/com/laogou/hello/

```
package com.lagou.hello

class helloGroovy {
    public static void main(String[] args) {
        def name="laosiji";
        println("groovy hello!! ," + name)
        println('单引号, 中文、分号测试');
    }
}
```

### 总结

1. 从输出结果可以看出 Groovy 里面支持单引号和双引号两种方式，注释支持//和/\*\*/两种方式，而且不以分号“;”结尾也可以，但是我们还是推荐都带上分号保持代码的一致性。
2. 标识符被用来定义变量，函数或其他用户定义的变量。标识符以字母、美元或下划线开头，不能以数字开头。

## 数据类型

除了字符串之外，Groovy 也支持有符号整数、浮点数、字符等。

```
class Example1 {
    public static void main(String[] args) {
        String str = "Hello"; // 字符串
        int x = 5; // 整数
        long y = 100L; // 长整型
        float a = 10.56f; // 32位浮点数
        double b = 10.5e40; // 64位浮点数
        char c = 'A'; // 字符
        Boolean l = true; // 布尔值，可以是true或false。
        println(str);
        println(x);
        println(y);
        println(a);
        println(b);
        println(c);
        println(l);
    }
}
```

## 打印变量

用 def 关键字来定义变量，当然也可以用一个确定的数据类型来声明一个变量，我们可以用下面的几种方式来打印变量



```

class Example2 {
    public static void main(String[] args) {
        // 初始化两个变量
        int x = 5;
        int y = 6;

        // 打印变量值
        println("x = " + x + " and y = " + y);
        println("x = ${x} and y = ${y}");
        println('x = ${x} and y = ${y}');
    }
}

```

## 注意事项

从这里我们可以看出 Groovy 在单引号的字符串里面是不支持插值的，这点非常重要，很多同学在使用 Pipeline 脚本的时候经常会混淆。

除此之外，还支持三引号：Groovy 里面三引号支持双引号和单引号两种方式，但是单引号同样不支持插值，要记住。

```

class Example3 {
    public static void main(String[] args) {
        // 初始化两个变量
        int x = 5;
        int y = 6;

        println """
        x = ${x}
        x = ${y}
        """

        println '''
        x = ${x}
        x = ${y}
        '''
    }
}

```

## 函数

Groovy 中的函数是使用返回类型或使用 def 关键字定义的，函数可以接收任意数量的参数，定义参数时，不必显式定义类型，可以添加修饰符，如 public，private 和 protected，默认情况下，如果未提供可见性修饰符，则该方法为 public。

```

class Example4 {
    static def PrintHello() {
        println("This is a print hello function in groovy");
    }

    static int sum(int a, int b, int c = 10) {
        int d = a+b+c;
    }
}

```

```

        return d;
    }

    public static void main(String[] args) {
        PrintHello();
        println(sum(5, 50));
    }
}

```

## 条件语句

在我们日常工作中条件判断语句是必不可少的，即使在 Jenkins Pipeline 脚本中也会经常遇到，Groovy 里面的条件语句和其他语言基本一致，使用 if/else 判断

```

class Example5 {
    public static void main(String[] args) {
        // 初始化变量值
        int a = 2

        // 条件判断
        if (a < 100) {
            // 如果a<100打印下面这句话
            println("The value is less than 100");
        } else {
            // 如果a>=100打印下面这句话
            println("The value is greater than 100");
        }
    }
}

```

## 循环语句

除了条件判断语句之外，循环语句也是非常重要的，Groovy 中可以使用三种方式来进行循环：`while`、`for`语句、`for-in`语句

```

class Example6 {
    public static void main(String[] args) {
        int count = 0;
        println("while循环语句: ");
        while(count<5) {
            println(count);
            count++;
        }

        println("for循环语句: ");
        for(int i=0;i<5;i++) {
            println(i);
        }

        println("for-in循环语句: ");
        int[] array = [0,1,2,3];
        for(int i in array) {

```

```
        println(i);
    }

    println("for-in循环范围: ");
    for(int i in 1..5) {
        println(i);
    }
}
}
```

除了上面这些最基本的特性外，Groovy 还支持很多其他的特性，比如异常处理、面向对象设计、正则表达式、泛型、闭包等等，由于我们这里只是为了让大家对 Jenkins Pipeline 的脚本有一个基本的认识，更深层次的用法很少会涉及到，大家如果感兴趣的可以去查阅官方文档了解更多信息。

## jenkins共享库入门

### 简介

实际生产环境中，运维工程师经常使用 Jenkins Pipeline 一定会遇到多个不同流水线中有大量重复代码的情况，很多时候为了方便我们都是直接复制粘贴到不同的Jenkinsfile文件中，但是长期下去这些代码的维护就会越来越麻烦。为了解决这个问题，Jenkins 中提供了共享库的概念来解决重复代码的问题，我们只需要将公共部分提取出来，然后就可以在所有的 Pipeline 中引用这些共享库下面的代码了。

### 什么是共享库

共享库（shared library）是一些**独立的 Groovy 脚本的集合**，我们可以在运行 Pipeline 的时候去获取这些共享库代码。使用共享库最好的方式同样是把代码使用 Git 仓库进行托管，这样我们就可以进行版本化管理了。当然我们也需要一些 Groovy 语言的基础，不过并不需要多深入，基本的语法概念掌握即可，可以查看前面我们的Groovy 简明教程。

使用共享库一般只需要3个步骤即可：

- 首先创建 Groovy 脚本，添加到 Git 仓库中
- 然后在 Jenkins 中配置将共享库添加到 Jenkins 中来
- 最后，在我们的流水线中导入需要使用的共享库：Jenkins老版本中使用 `@Library('your-shared-library')`，这样就可以使用共享库中的代码。新版本中 `Library 'your-shared-library'`，这样就可以使用共享库中的代码。

### 共享库目录

#### 官网示例

```
(root)
+- src                                # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy # for org.foo.Bar class
+- vars
|   +- foo.groovy         # for global 'foo' variable
|   +- foo.txt            # help for 'foo' variable
+- resources              # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json   # static helper data for org.foo.Bar
```

## sayHello.groovy

在工程中新建vars目录。

idea设置: project structure->module->source, 将vars目录标记为Sources目录。如果不标记为Sources目录, vars下不能创建\*.groovy文件。

在目录vars/sayHello.groovy新建groovy文件, 文件内容如下:

```
def call(String name = "laosiji") {
    println("hello $name")
}
```

## 配置共享库

jenkins工作台->系统管理->系统配置-> Global Pipeline Libraries

### Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library Name	Default version	Load implicitly	Allow default version to be overridden	Include @Library changes in job recent changes	Retrieval method	Source Code Management	Item repository	Credentials	Behavior	Repository related	Discover branches	Actions
groovyhello	master	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Modern SCM	Git	ssh://git@192.168.198.152:222/lagou/groovyhello.git	root (gitlab-154-SSH)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	保存 应用 删除

## pipeline任务

```
//引入SharedLibrary库
library "groovyhello"
pipeline {
    agent {
        label 'jenkinsagent-154'
    }

    stages {
        stage('Hello') {
            steps {
                echo 'Hello world'
                //通过groovy脚本名称直接调用
                sayHello "lagou"
            }
        }
    }
}
```

## 测试pipeline任务

点击->立即构建

## src目录测试

src/com.lagou.hello.ansible.groovy

```
package com.lagou.hello

def printMessage(){
    println("hello laosiji")
}
```

## Jenkinsfile文件

```
@Library("groovyhello")_

def build = new com.lagou.hello.ansible()
pipeline {
    agent {
        label 'jenkinsagent-154'
    }

    stages {
        stage('Hello') {
            steps {
                echo 'Hello world'
            }
        }
    }
}
```

```
        sayHello "lagou"  
        script{  
            build.printMessage()  
        }  
    }  
}  
}
```