《Jenkins持续集成入门到精通》

1、持续集成及Jenkins介绍

2、Jenkins安装和持续集成环境配置

3、Jenkins构建Maven项目

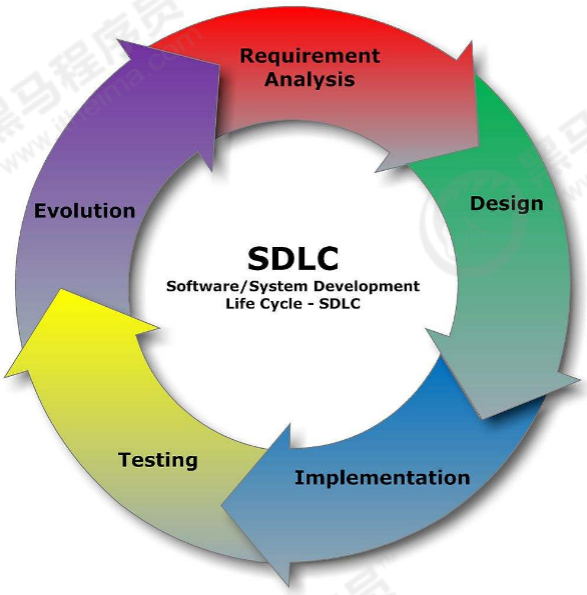
4、Jenkins+Docker+SpringCloud微服务持续集成

5、基于Kubernetes/K8S构建Jenkins微服务持续集成平台

1、持续集成及Jenkins介绍

软件开发生命周期

软件开发生命周期又叫做**SDLC**（Software Development Life Cycle），它是集合了计划、开发、测试和部署过程的集合。如下图所示 ：



- 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。

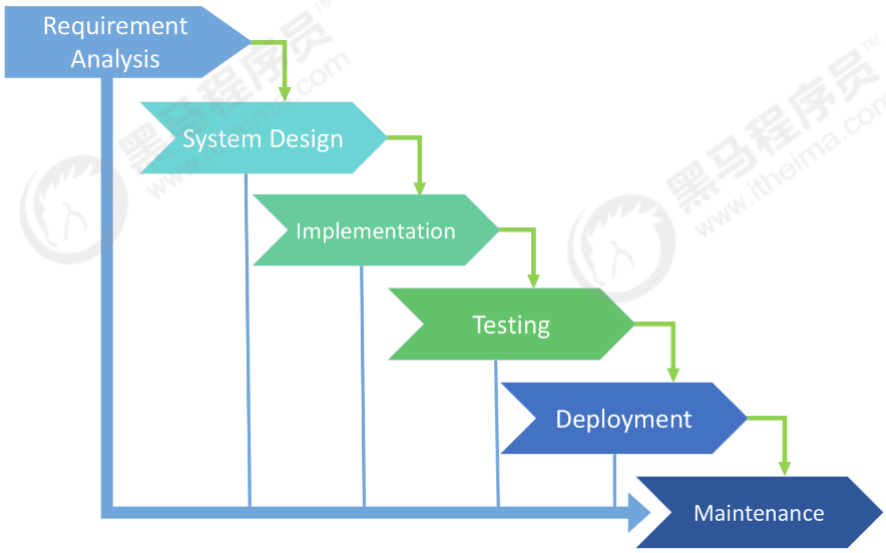
- 进化

最后进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。

软件开发瀑布模型

瀑布模型是最著名和最常使用的软件开发模型。瀑布模型就是一系列的软件开发过程。它是由制造业繁衍出来的。一个高度化的结构流程在一个方向上流动，有点像生产线一样。在瀑布模型创建之初，没有其它开发的模型，有很多东西全靠开发人员去猜测，去开发。这样的模型仅适用于那些简单的软件开发， 但是已经不适合现在的开发了。

下图对软件开发模型的一个阐述。



优势	劣势
简单易用和理解	各个阶段的划分完全固定，阶段之间产生大量的文档，极大地增加了工作量。
当前一阶段完成后，您只需要去关注后续阶段。	由于开发模型是线性的，用户只有等到整个过程的末期才能见到开发成果，从而增加了开发风险。
为项目提供了按阶段划分的检查节点	瀑布模型的突出缺点是不适应用户需求的变化。

软件的敏捷开发

什么是敏捷开发？

敏捷开发（Agile Development）的核心是迭代开发（Iterative Development）与 增量开发（Incremental Development）。

===何为迭代开发？===

对于大型软件项目，传统的开发方式是采用一个大周期（比如一年）进行开发，整个过程就是一次"大开发"；迭代开发的方式则不一样，它将开发过程拆分成多个小周期，即一次"大开发"变成多次"小开发"，每次小开发都是同样的流程，所以看上去就好像重复在做同样的步骤。

举例来说，SpaceX 公司想造一个大推力火箭，将人类送到火星。但是，它不是一开始就造大火箭，而是先造一个最简陋的小火箭 Falcon 1。结果，第一次发射就爆炸了，直到第四次发射，才成功进入轨道。然后，开发了中型火箭 Falcon 9，九年中发射了70次。最后，才开发 Falcon 重型火箭。如果 SpaceX 不采用迭代开发，它可能直到现在还无法上天。

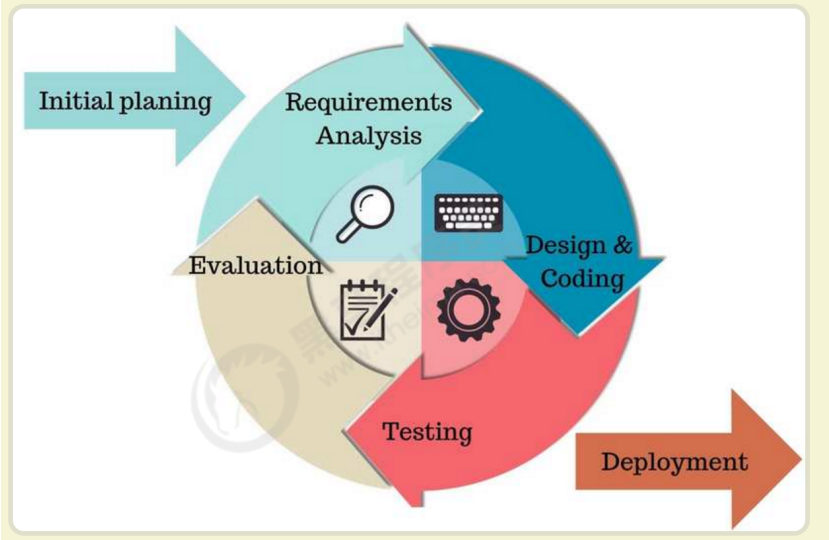
===何为增量开发？===

软件的每个版本，都会新增一个用户可以感知的完整功能。也就是说，按照新增功能来划分迭代。

举例来说，房产公司开发一个10栋楼的小区。如果采用增量开发的模式，该公司第一个迭代就是交付一号楼，第二个迭代交付二号楼……每个迭代都是完成一栋完整的楼。而不是第一个迭代挖好10栋楼的地基，第二个迭代建好每栋楼的骨架，第三个迭代架设屋顶……

敏捷开发如何迭代？

虽然敏捷开发将软件开发分成多个迭代，但是也要求，每次迭代都是一个完整的软件开发周期，必须按照软件工程的方法论，进行正规的流程管理。



敏捷开发有什么好处？

==早期交付==

敏捷开发的第一个好处，就是早期交付，从而大大降低成本。 还是以上一节的房产公司为例，如果按照传统的"瀑布开发模式"，先挖10栋楼的地基、再盖骨架、然后架设屋顶，每个阶段都等到前一个阶段完成后开始，可能需要两年才能一次性交付10栋楼。也就是说，如果不考虑预售，该项目必须等到两年后才能回款。 敏捷开发是六个月后交付一号楼，后面每两个月交付一栋楼。因此，半年就能回款10%，后面每个月都会有现金流，资金压力就大大减轻了。

==降低风险===

敏捷开发的第二个好处是，及时了解市场需求，降低产品不适用的风险。 请想一想，哪一种情况损失比较小：10栋楼都造好以后，才发现卖不出去，还是造好第一栋楼，就发现卖不出去，从而改进或停建后面9栋楼？

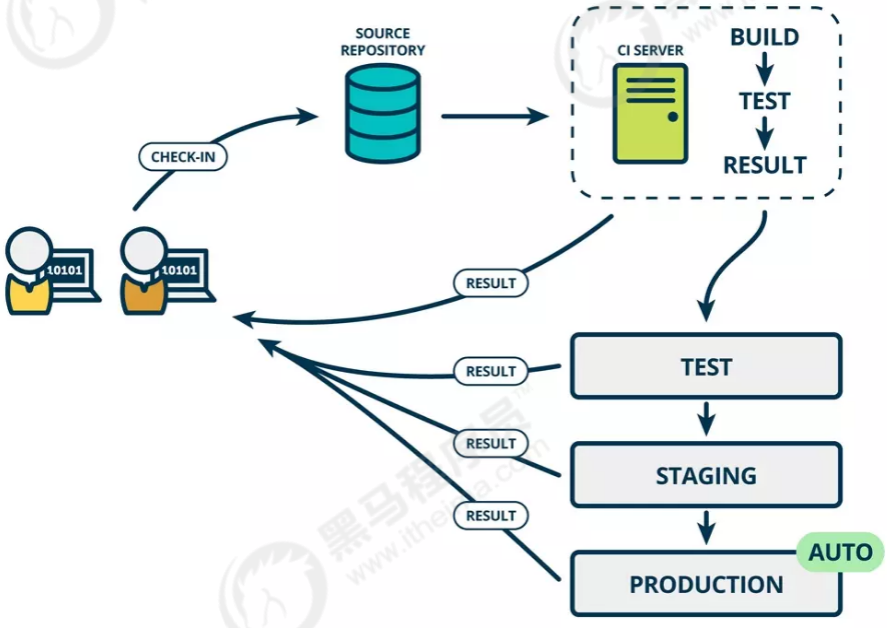
什么是持续集成

持续集成（ Continuous integration ， 简称 CI ）指的是，频繁地（一天多次）将代码集成到主干。

持续集成的目的，就是让产品可以快速迭代，同时还能保持高质量。它的核心措施是，代码集成到主干之前，必须通过自动化测试。只要有一个测试用例失败，就不能集成。

通过持续集成， 团队可以快速的从一个功能到另一个功能，简而言之，敏捷软件开发很大一部分都要归功于持续集成。

=== 持续集成的流程===



根据持续集成的设计，代码从提交到生产，整个过程有以下几步。

- 提交

流程的第一步，是开发者向代码仓库提交代码。所有后面的步骤都始于本地代码的一次提交（ commit ）。

- 测试（第一轮）

代码仓库对commit操作配置了钩子（hook），只要提交代码或者合并进主干，就会跑自动化测试。

- 构建

通过第一轮测试，代码就可以合并进主干，就算可以交付了。

交付后，就先进行构建（build），再进入第二轮测试。所谓构建，指的是将源码转换为可以运行的实际代码，比如安装依赖，配置各种资源（样式表、JS脚本、图片）等等。

- 测试（第二轮）

构建完成，就要进行第二轮测试。如果第一轮已经涵盖了所有测试内容，第二轮可以省略，当然，这时构建步骤也要移到第一轮测试前面。

- 部署

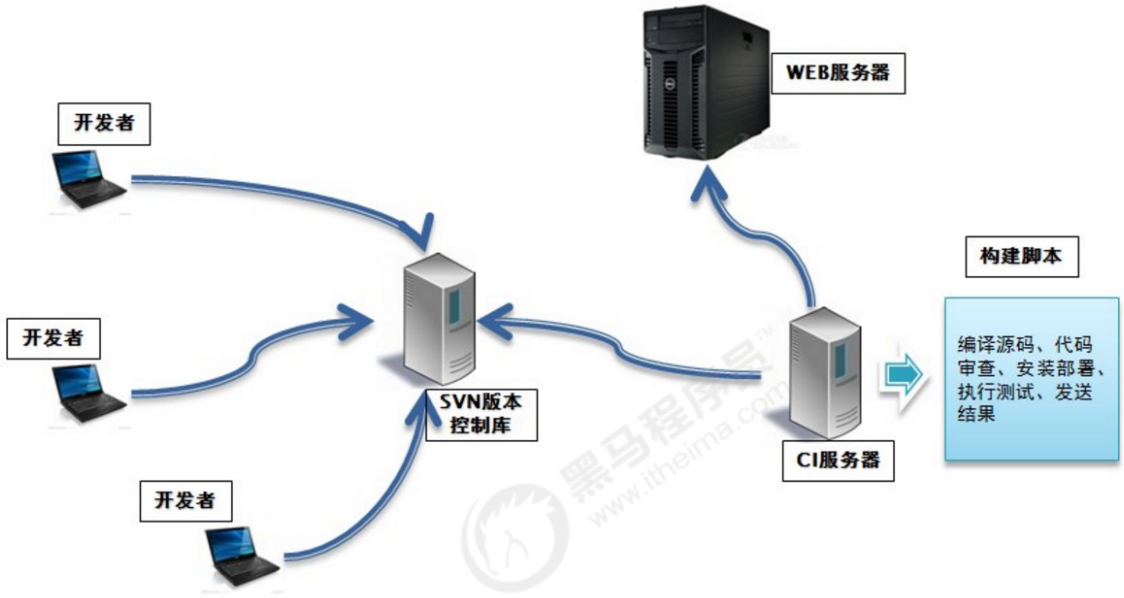
过了第二轮测试，当前代码就是一个可以直接部署的版本（artifact）。将这个版本的所有文件打包（tar filename.tar * ）存档，发到生产服务器。

- 回滚

一旦当前版本发生问题，就要回滚到上一个版本的构建结果。最简单的做法就是修改一下符号链接，指向上一个版本的目录。

持续集成的组成要素

- 一个自动构建过程， 从检出代码、 编译构建、 运行测试、 结果记录、 测试统计等都是自动完成的， 无需人工干预。
- 一个代码存储库，即需要版本控制软件来保障代码的可维护性，同时作为构建过程的素材库，一般使用SVN或Git。
- 一个持续集成服务器， Jenkins 就是一个配置简单和使用方便的持续集成服务器。



持续集成的好处

1、降低风险，由于持续集成不断去构建，编译和测试，可以很早期发现问题，所以修复的代价就少；

2、对系统健康持续检查，减少发布风险带来的问题；

3、减少重复性工作；

4、持续部署，提供可部署单元包；

5、持续交付可供使用的版本；

6、增强团队信心；

Jenkins介绍

Jenkins 是一款流行的开源持续集成（Continuous Integration）工具，广泛用于项目开发，具有自动化构建、测试和部署等功能。官网： http://jenkins-ci.org/。

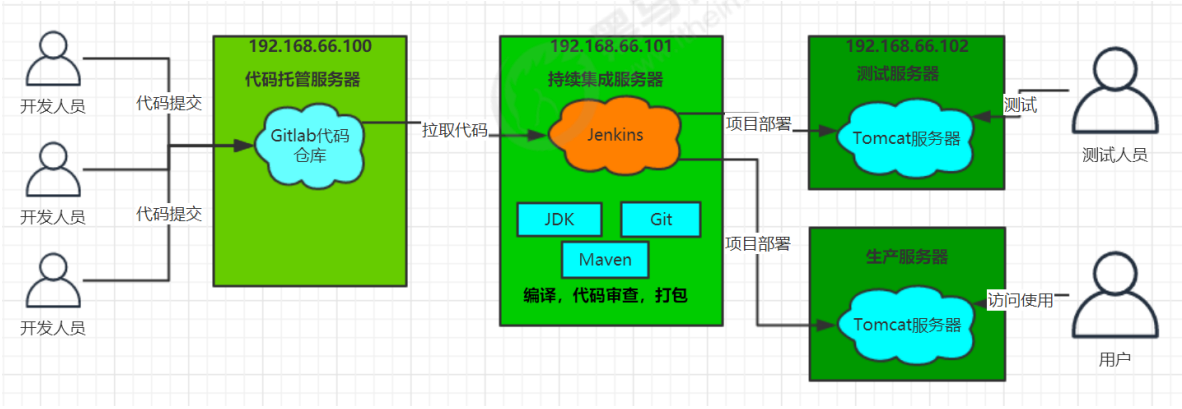
Jenkins的特征：

- 开源的Java语言开发持续集成工具，支持持续集成，持续部署。
- 易于安装部署配置：可通过yum安装,或下载war包以及通过docker容器等快速实现安装部署，可方便web界面配置管理。
- 消息通知及测试报告：集成RSS/E-mail通过RSS发布构建结果或当构建完成时通过e-mail通知，生成JUnit/TestNG测试报告。

- 分布式构建：支持Jenkins能够让多台计算机一起构建/测试。
- 文件识别：Jenkins能够跟踪哪次构建生成哪些jar，哪次构建使用哪个版本的jar等。
- 丰富的插件支持：支持扩展插件，你可以开发适合自己团队使用的工具，如git，svn，maven，docker等。

2、Jenkins安装和持续集成环境配置

持续集成流程说明



1）首先，开发人员每天进行代码提交，提交到Git仓库

2）然后，Jenkins作为持续集成工具，使用Git工具到Git仓库拉取代码到集成服务器，再配合JDK，Maven等软件完成代码编译，代码测试与审查，测试，打包等工作，在这个过程中每一步出错，都重新再执行一次整个流程。

3）最后，Jenkins把生成的jar或war包分发到测试服务器或者生产服务器，测试人员或用户就可以访问应用。

服务器列表

本课程虚拟机统一采用CentOS7。

名称	IP地址	安装的软件
代码托管服务器	192.168.66.100	Gitlab-12.4.2
持续集成服务器	192.168.66.101	Jenkins-2.190.3，JDK1.8，Maven3.6.2，Git，SonarQube
应用测试服务器	192.168.66.102	JDK1.8，Tomcat8.5

Gitlab代码托管服务器安装

Gitlab简介

官网： https://about.gitlab.com/

GitLab 是一个用于仓库管理系统的开源项目，使用Git作为代码管理工具，并在此基础上搭建起来的web服务。

GitLab和GitHub一样属于第三方基于Git开发的作品，免费且开源（基于MIT协议），与Github类似，可以注册用户，任意提交你的代码，添加SSHKey等等。不同的是，**GitLab是可以部署到自己的服务器上，数据库等一切信息都掌握在自己手上，适合团队内部协作开发**，你总不可能把团队内部的智慧总放在别人的服务器上吧？简单来说可把GitLab看作个人版的GitHub。

Gitlab安装

1. 安装相关依赖

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

2. 启动ssh服务&设置为开机启动

   > systemctl enable sshd && sudo systemctl start sshd

3. 设置postfix开机自启，并启动，postfix支持gitlab发信功能

   > systemctl enable postfix && systemctl start postfix

4. 开放ssh以及http服务，然后重新加载防火墙列表

   > firewall-cmd --add-service=ssh --permanent
   >
   > firewall-cmd --add-service=http --permanent
   >
   > firewall-cmd --reload

   如果关闭防火墙就不需要做以上配置

5. 下载gitlab包，并且安装

   > 在线下载安装包：
   >
   > wget https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el6/gitlab-ce-12.4.2-ce.0.el6.x86_64.rpm
   >
   > 安装：
   >
   > rpm -i gitlab-ce-12.4.2-ce.0.el6.x86_64.rpm

6. 修改gitlab配置

   > vi /etc/gitlab/gitlab.rb

修改gitlab访问地址和端口，默认为80，我们改为82

external_url 'http://192.168.66.100:82'

nginx['listen_port'] = 82

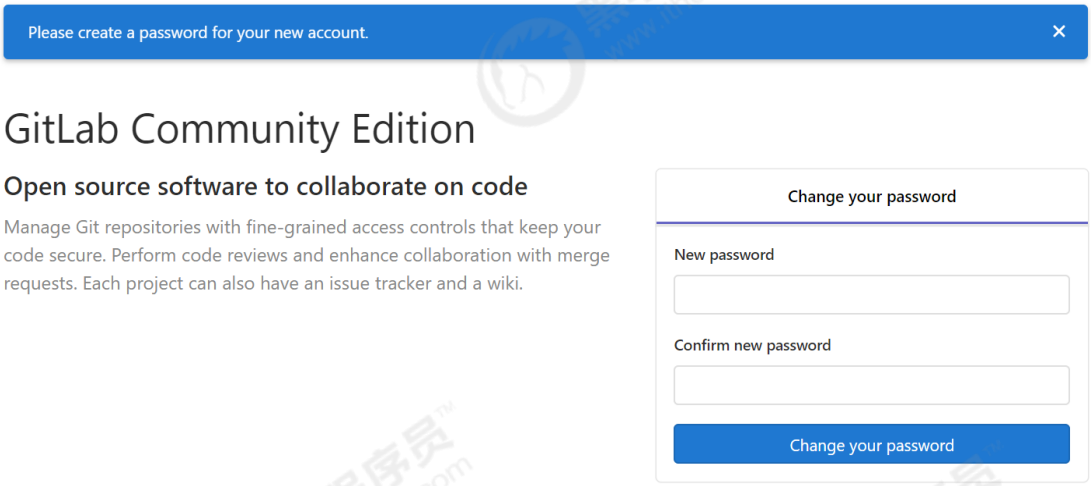
7. 重载配置及启动gitlab

> gitlab-ctl reconfigure
>
> gitlab-ctl restart

8. 把端口添加到防火墙

> firewall-cmd --zone=public --add-port=82/tcp --permanent
>
> firewall-cmd --reload

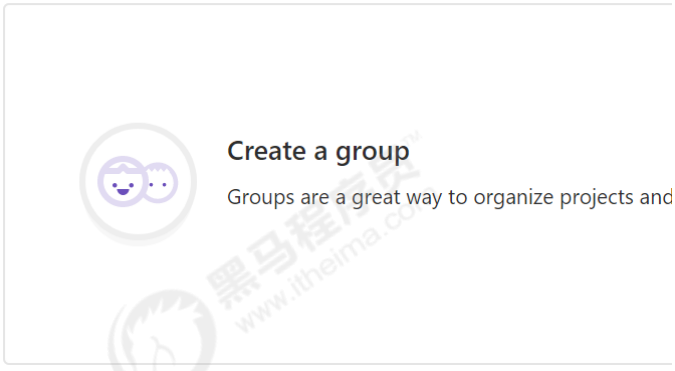
启动成功后，看到以下修改管理员root密码的页面，修改密码后，然后登录即可

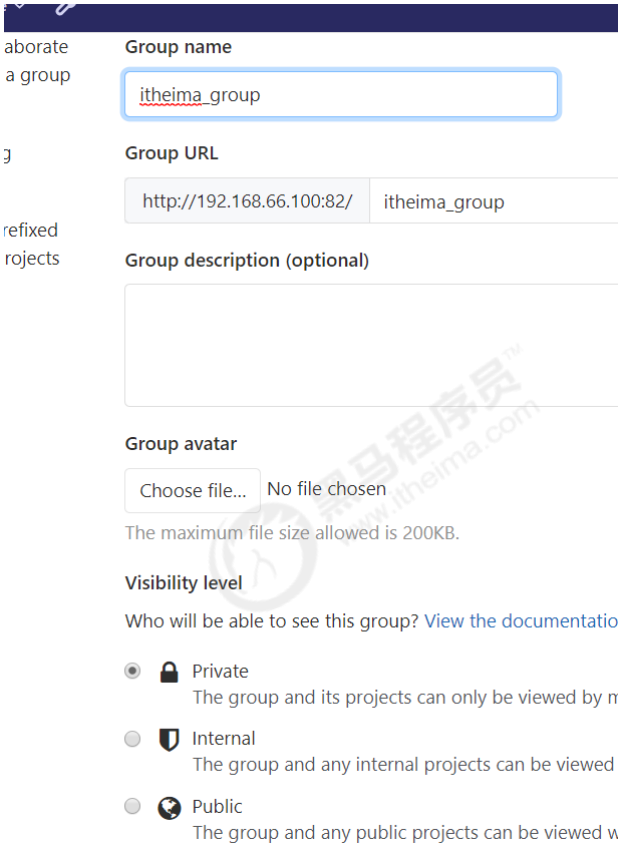


Gitlab添加组、创建用户、创建项目

1）创建组

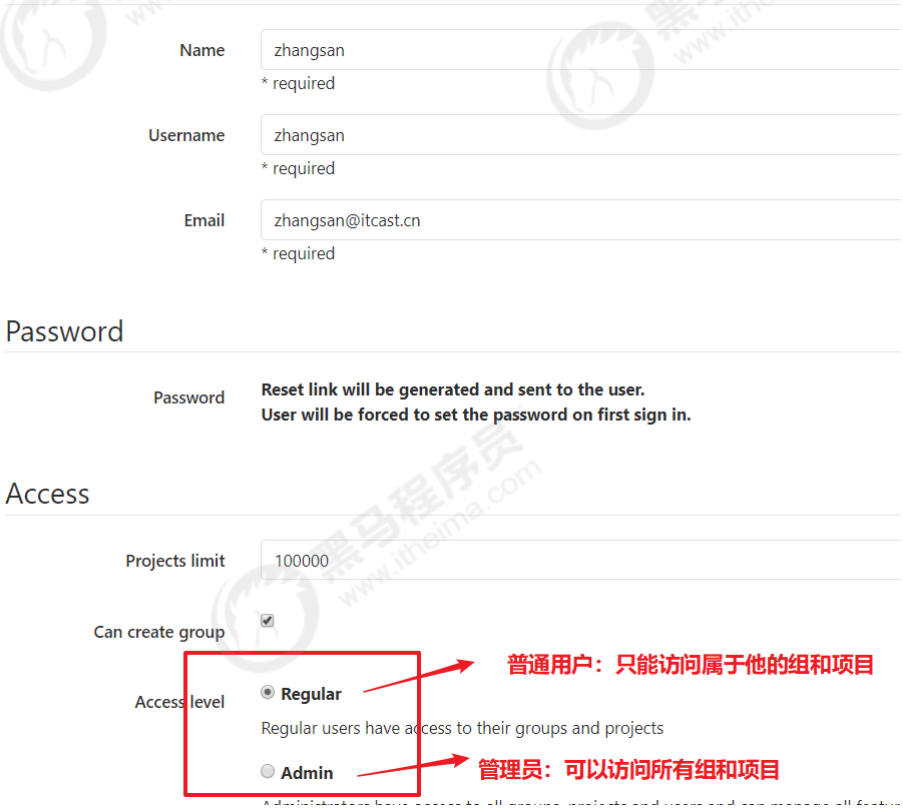
使用管理员 root 创建组，一个组里面可以有多个项目分支，可以将开发添加到组里面进行设置权限，不同的组就是公司不同的开发项目或者服务模块，不同的组添加不同的开发即可实现对开发设置权限的管理



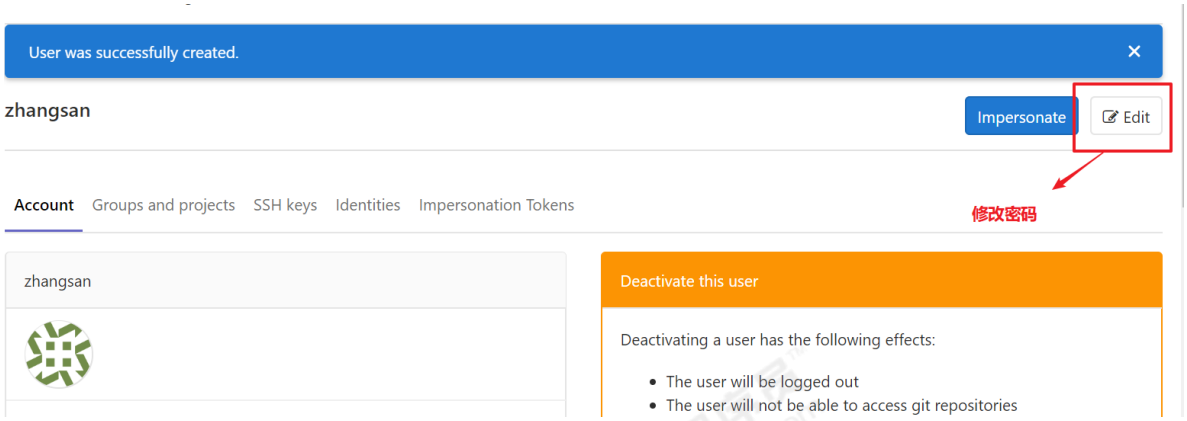


2）创建用户

创建用户的时候，可以选择Regular或Admin类型。

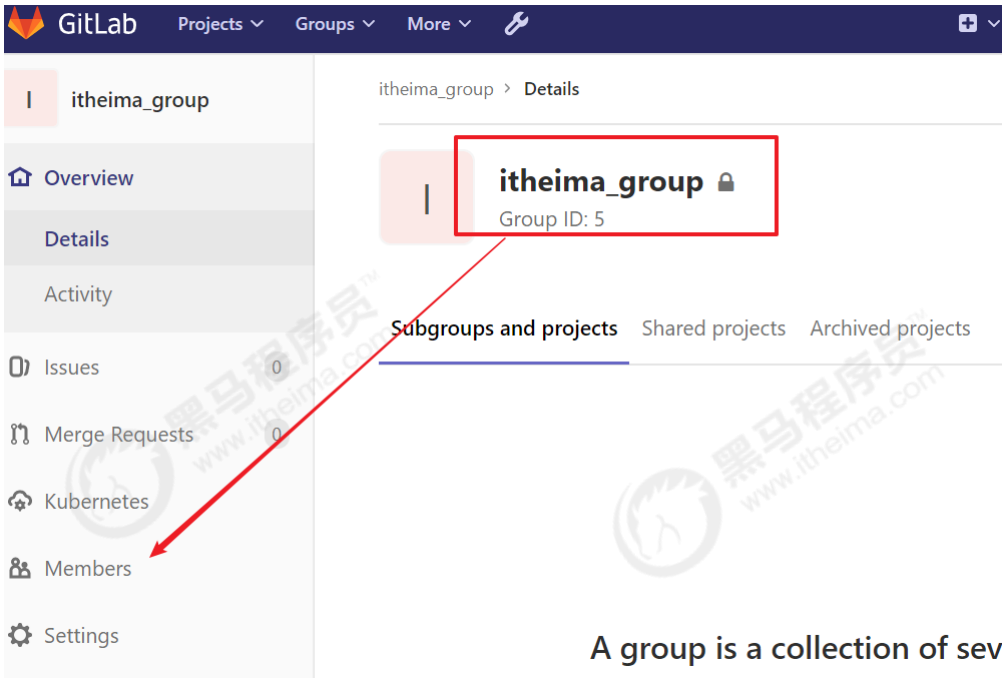


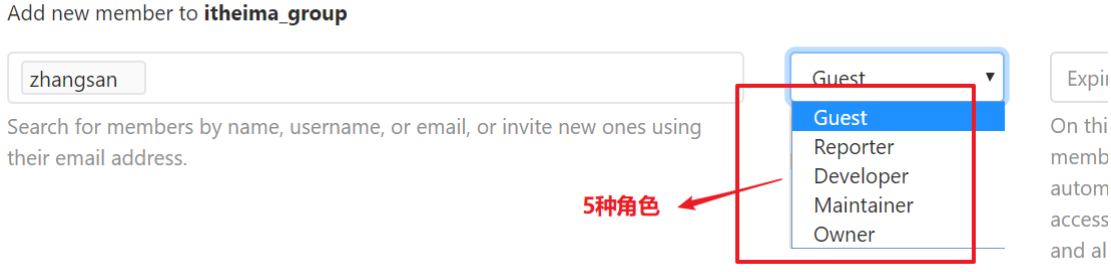
创建完用户后，立即修改密码



3）将用户添加到组中

选择某个用户组，进行Members管理组的成员



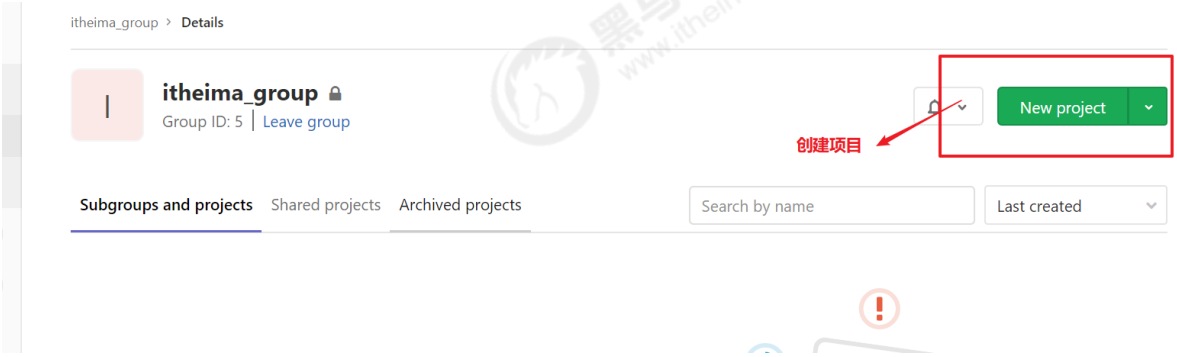


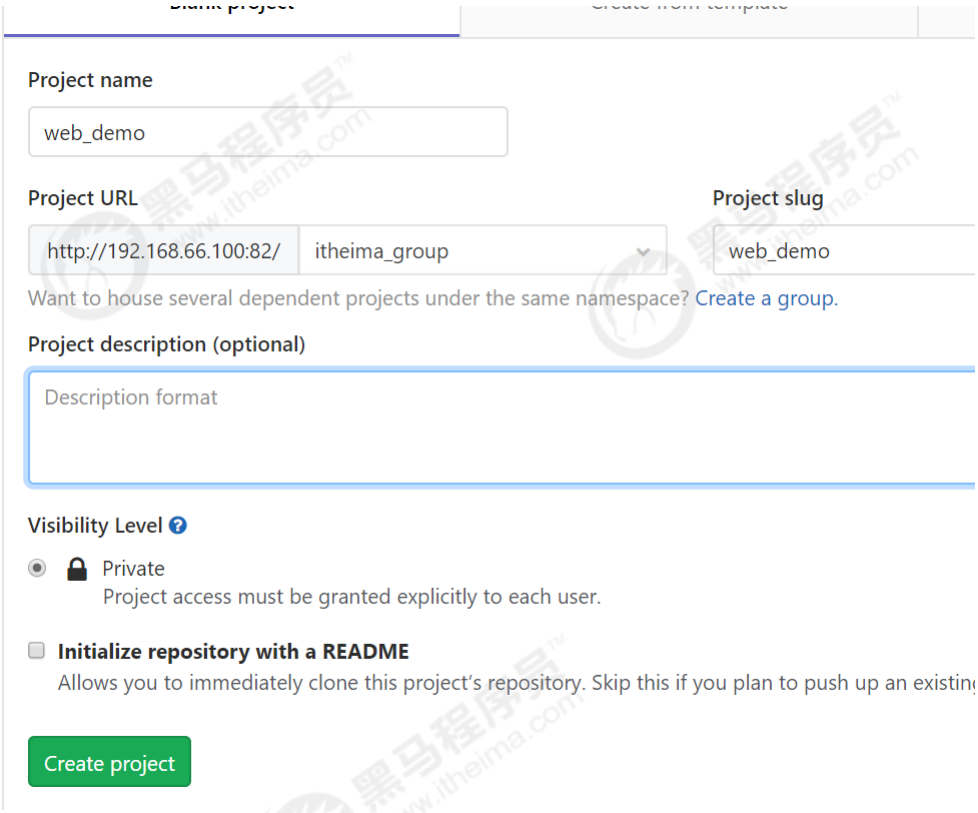
Gitlab用户在组里面有5种不同权限：

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

4）在用户组中创建项目

以刚才创建的新用户身份登录到Gitlab，然后在用户组中创建新的项目



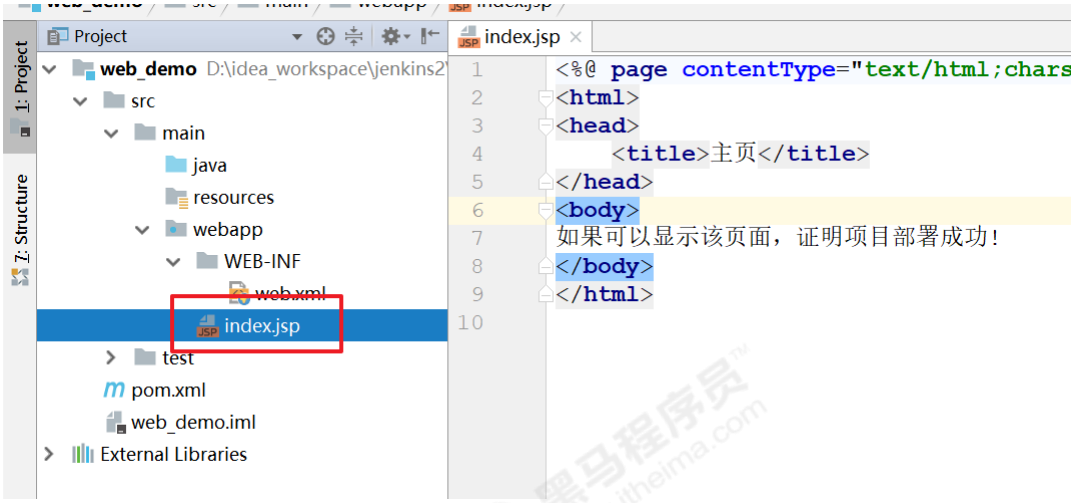


源码上传到Gitlab仓库

下面来到IDEA开发工具，我们已经准备好一个简单的Web应用准备到集成部署。

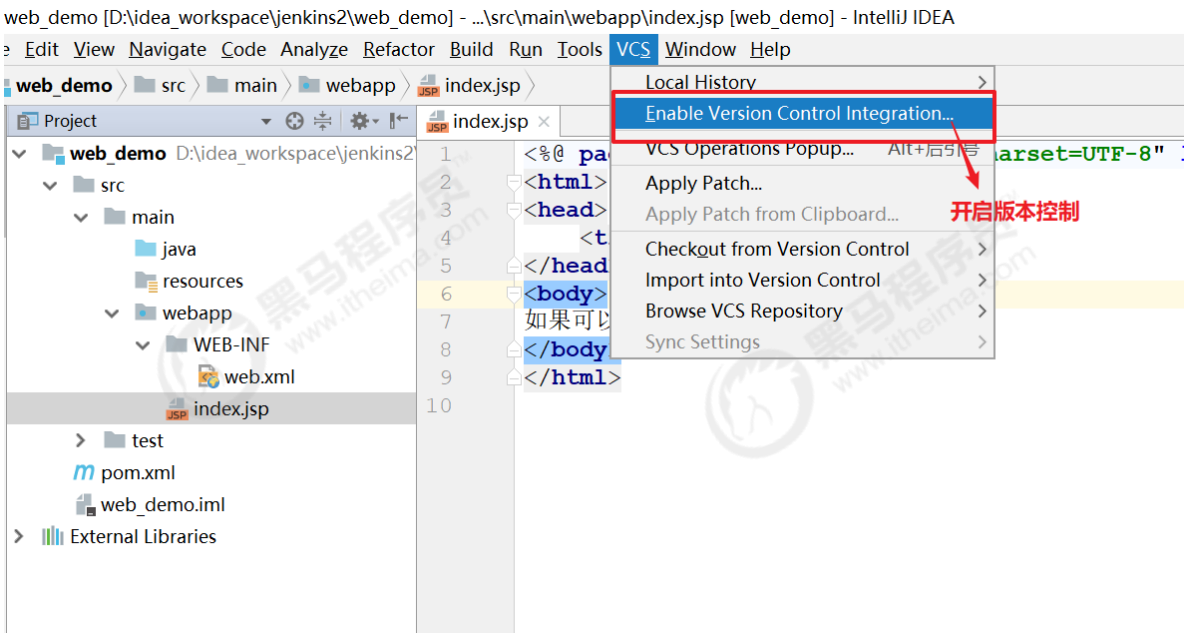
我们要把源码上传到Gitlab的项目仓库中。

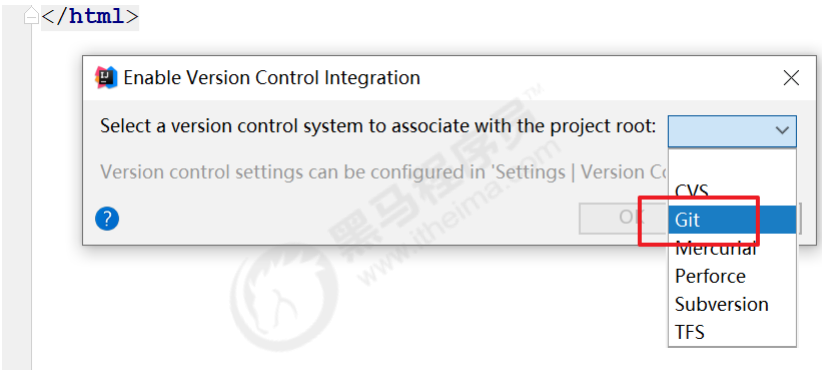
1）项目结构说明



我们建立了一个非常简单的web应用，只有一个index.jsp页面，如果部署好，可以访问该页面就成功啦！

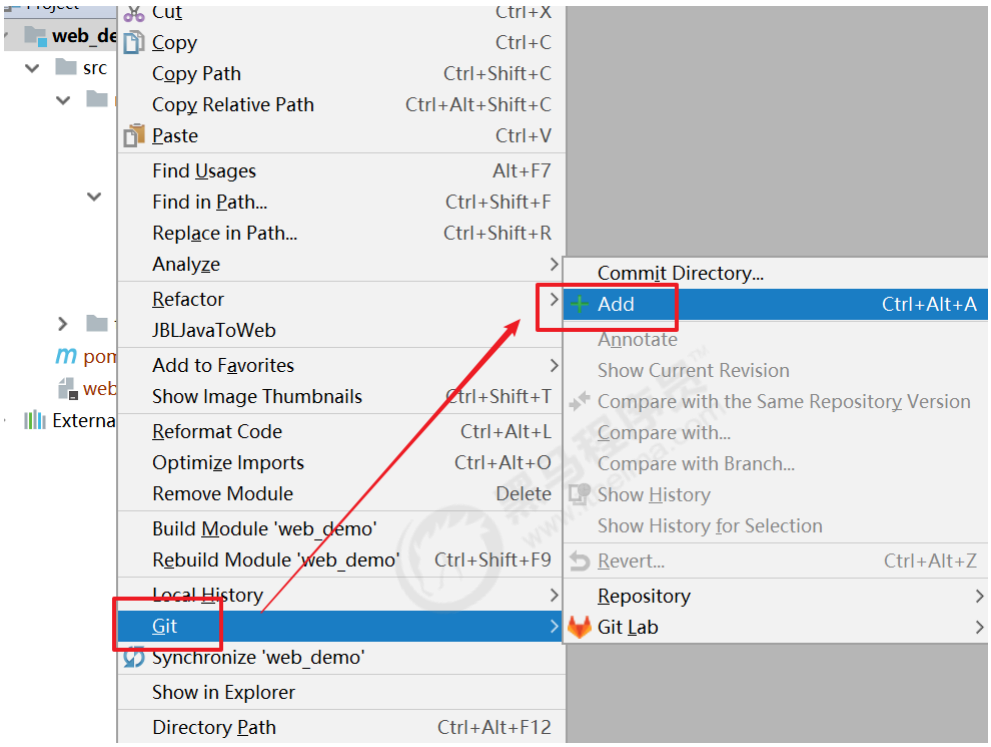
2）开启版本控制



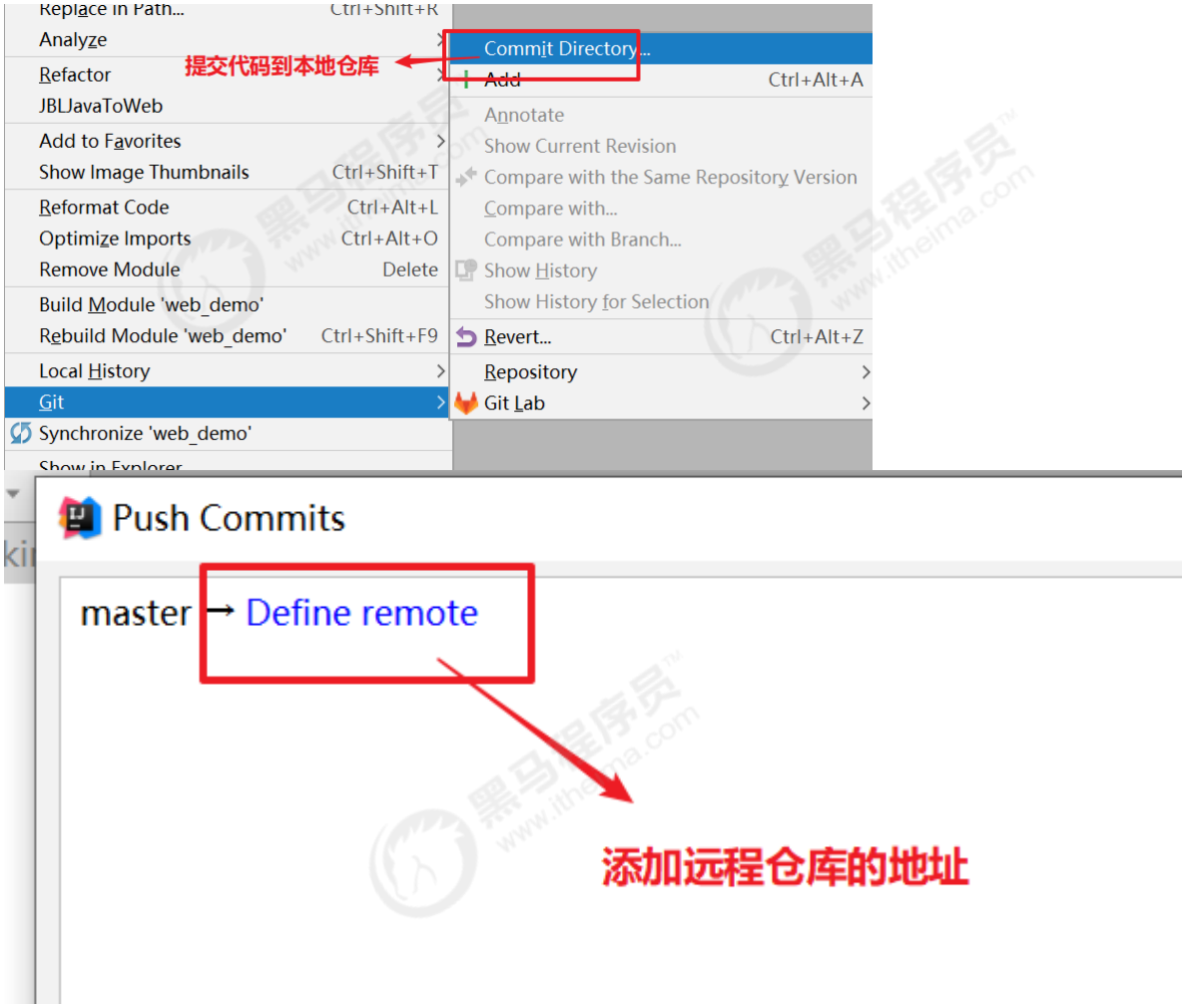


2）提交代码到本地仓库

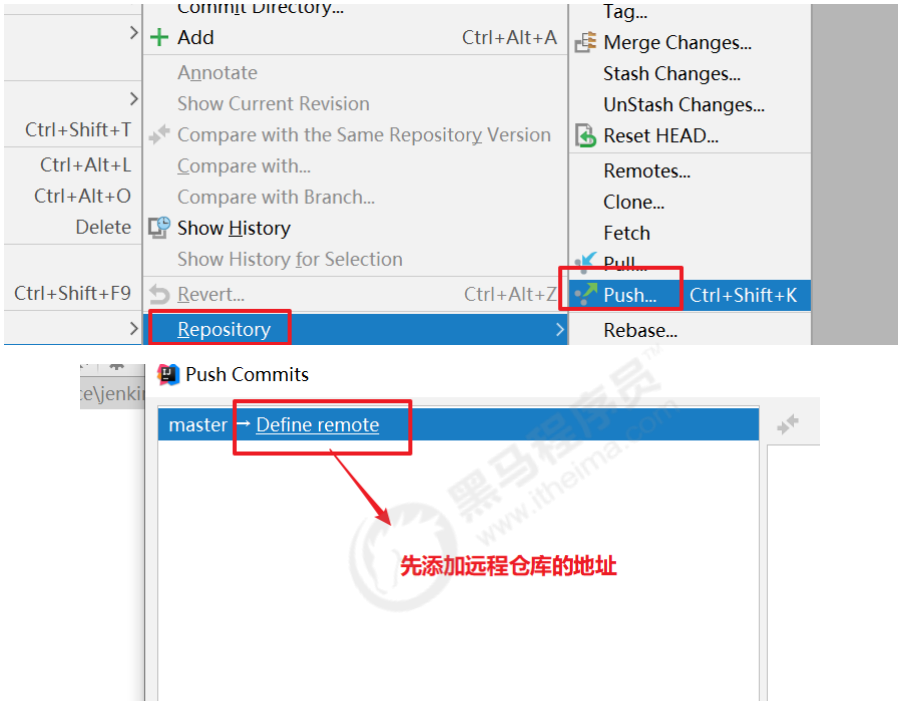
先Add到缓存区



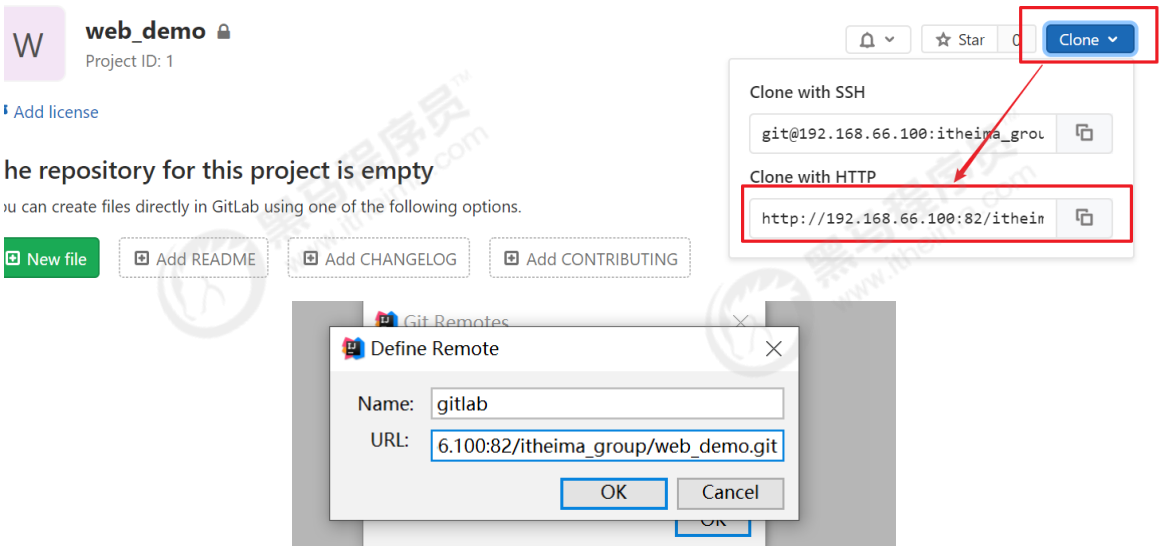
再Commit到本地仓库



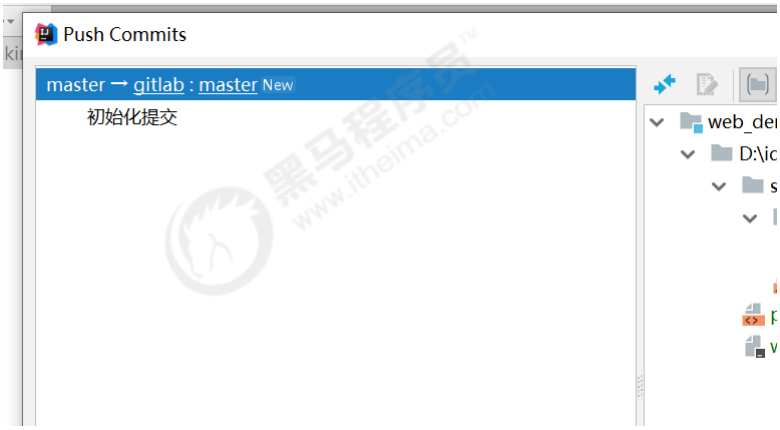
3）推送到Gitlab项目仓库中



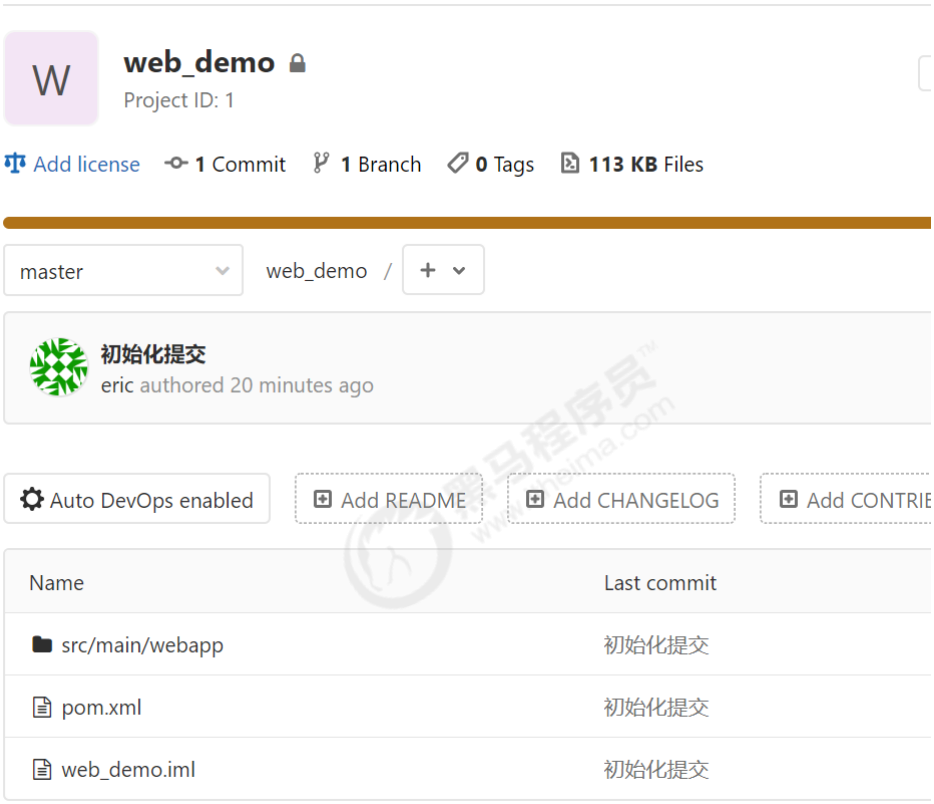
这时都Gitlab的项目中拷贝url地址



输入gitlab的用户名和密码，然后就可以把代码推送到远程仓库啦



刷新gitlab项目



持续集成环境(1)-Jenkins安装

1）安装JDK

Jenkins需要依赖JDK，所以先安装JDK1.8

> yum install java-1.8.0-openjdk* -y

安装目录为：/usr/lib/jvm

2）获取jenkins安装包

下载页面：https://jenkins.io/zh/download/

安装文件：jenkins-2.190.3-1.1.noarch.rpm

3）把安装包上传到192.168.66.101服务器，进行安装

> rpm -ivh jenkins-2.190.3-1.1.noarch.rpm

4）修改Jenkins配置

> vi /etc/syscofig/jenkins

修改内容如下：

> JENKINS_USER="root"
>
> JENKINS_PORT="8888"

5）启动Jenkins

> systemctl start jenkins

6）打开浏览器访问

http://192.168.66.101:8888

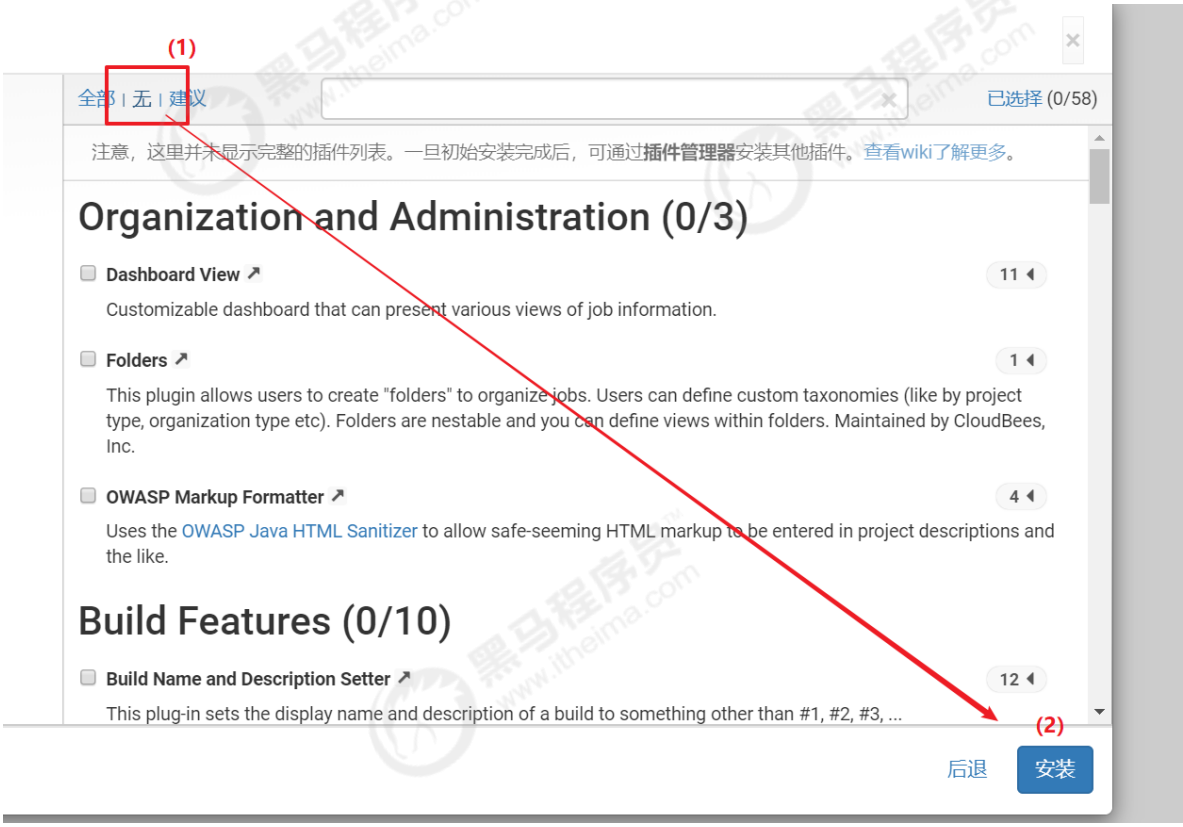
注意：本服务器把防火墙关闭了，如果开启防火墙，需要在防火墙添加端口

7）获取并输入admin账户密码

> cat /var/lib/jenkins/secrets/initialAdminPassword

8）跳过插件安装

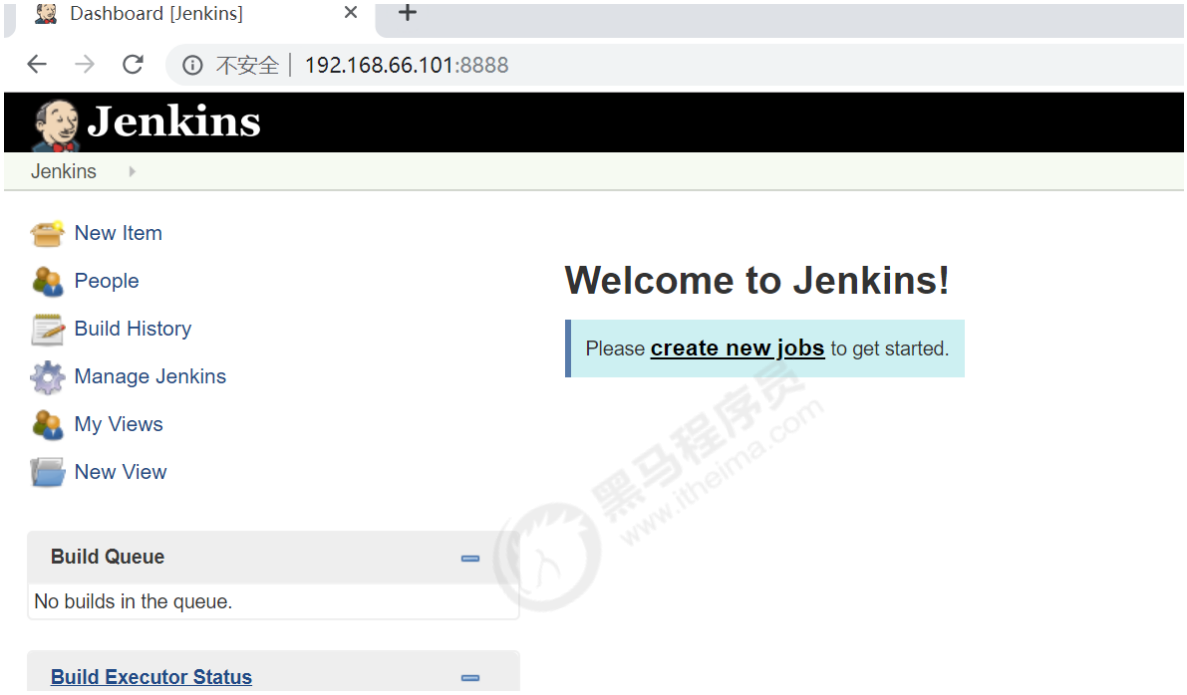
因为Jenkins插件需要连接默认官网下载，速度非常慢，而且经过会失败，所以我们暂时先跳过插件安装



9）添加一个管理员账户，并进入Jenkins后台

保存并完成

开始使用Jenkins



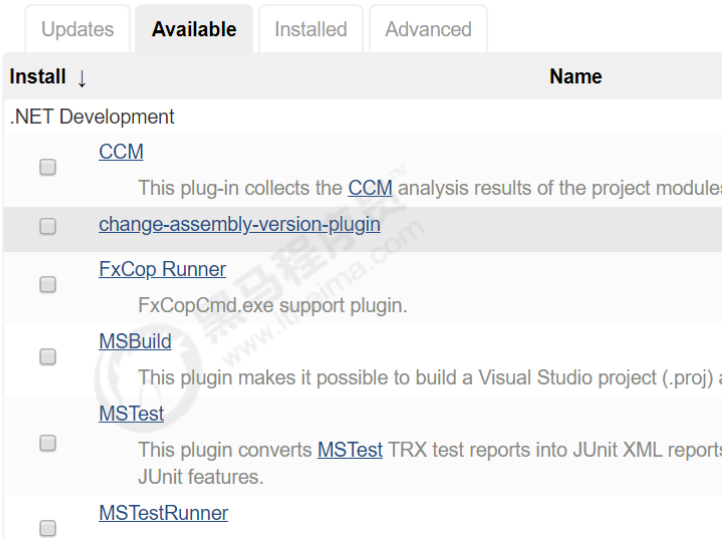
持续集成环境(2)-Jenkins插件管理

Jenkins本身不提供很多功能，我们可以通过使用插件来满足我们的使用。例如从Gitlab拉取代码，使用Maven构建项目等功能需要依靠插件完成。接下来演示如何下载插件。

修改Jenkins插件下载地址

Jenkins国外官方插件地址下载速度非常慢，所以可以修改为国内插件地址：

Jenkins->Manage Jenkins->Manage Plugins，点击Available



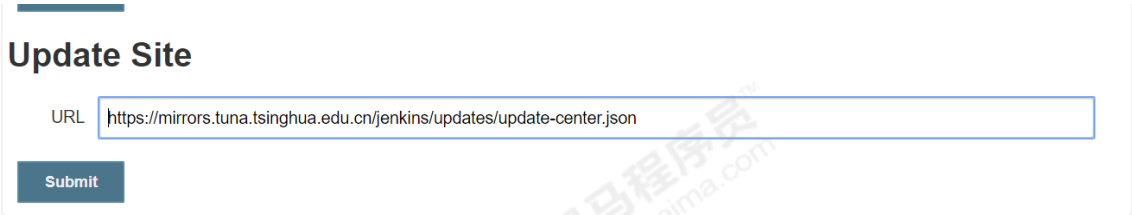
这样做是为了把Jenkins官方的插件列表下载到本地，接着修改地址文件，替换为国内插件地址

```
cd /var/lib/jenkins/updates
```

> sed -i 's/http:\/\/updates.jenkins-ci.org\/download/https:\/\/mirrors.tuna.tsinghua.edu.cn\/jenkins/g' default.json && sed -i 's/http:\/\/www.google.com/https:\/\/www.baidu.com/g' default.json

最后，Manage Plugins点击Advanced，把Update Site改为国内插件下载地址

> https://mirrors.tuna.tsinghua.edu.cn/jenkins/updates/update-center.json

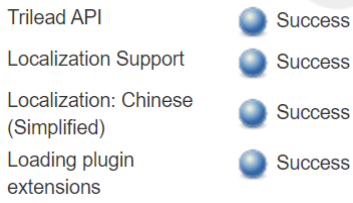


Sumbit后，在浏览器输入： http://192.168.66.101:8888/restart ，重启Jenkins。

下载中文汉化插件

Jenkins->Manage Jenkins->Manage Plugins，点击Available，搜索"Chinese"

完成后如下图：



重启Jenkins后，就看到Jenkins汉化了！（PS：但可能部分菜单汉化会失败）

持续集成环境(3)-Jenkins用户权限管理

我们可以利用Role-based Authorization Strategy 插件来管理Jenkins用户权限

安装Role-based Authorization Strategy插件

开启权限全局安全配置

授权策略切换为"Role-Based Strategy"，保存

创建角色

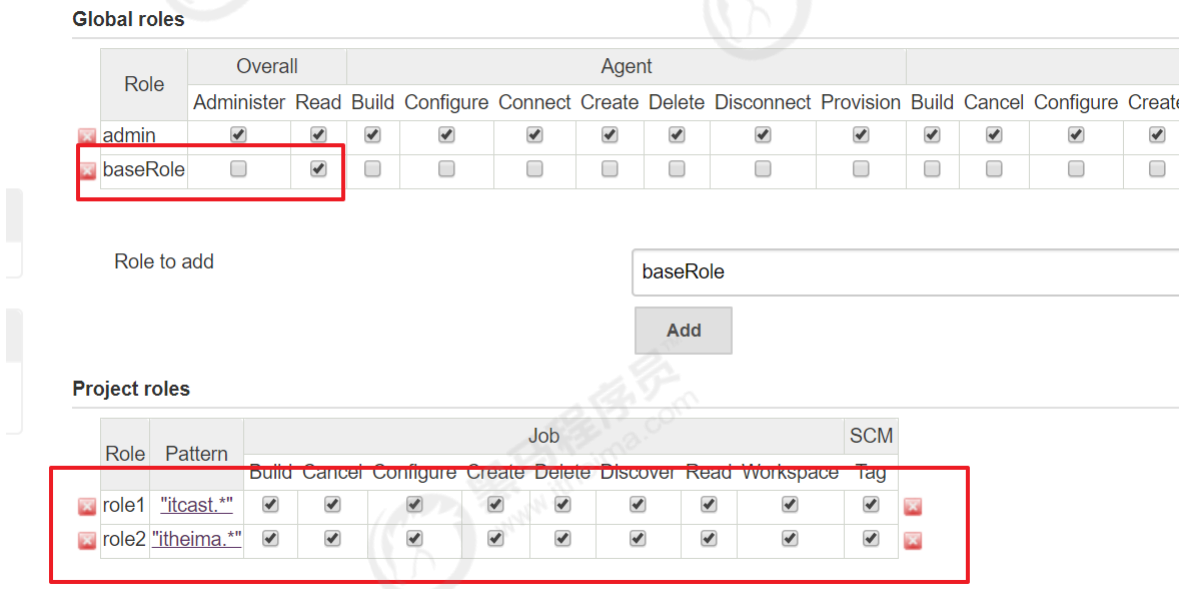
在系统管理页面进入 Manage and Assign Roles

点击"Manage Roles"

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 Project roles（项目角色）：针对某个或者某些项目的角色 Slave roles（奴隶角色）：节点相关的权限

我们添加以下三个角色：

- baseRole：该角色为全局角色。这个角色需要绑定Overall下面的Read权限，是为了给所有用户绑定最基本的Jenkins访问权限。注意：如果不给后续用户绑定这个角色，会报错误：用户名 is missing the Overall/Read permission
- role1：该角色为项目角色。使用正则表达式绑定"itcast.*"，意思是只能操作itcast开头的项目。
- role2：该角色也为项目角色。绑定"itheima.*"，意思是只能操作itheima开头的项目。



保存。

创建用户

在系统管理页面进入 Manage Users

分别创建两个用户：jack和eric

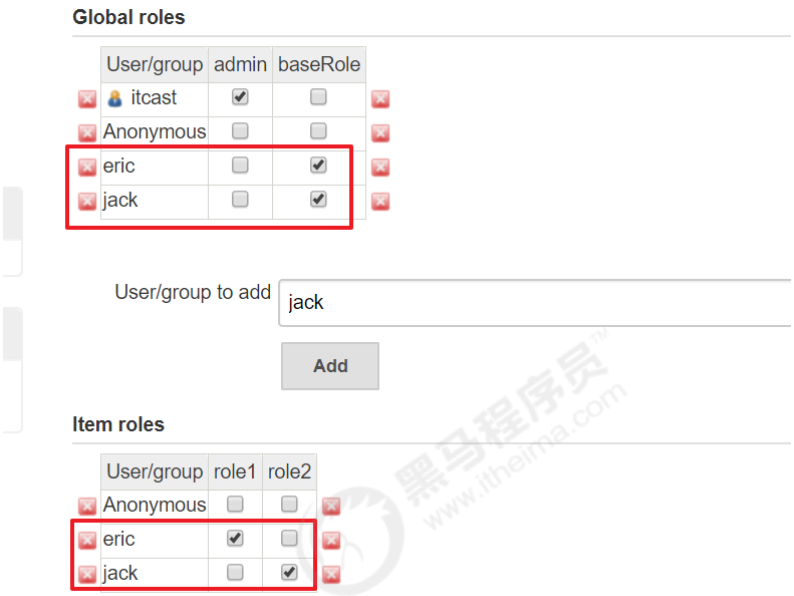
User ID	名称
eric	eric
itcast	itcast
jack	jack

给用户分配角色

系统管理页面进入Manage and Assign Roles，点击Assign Roles

绑定规则如下：

- eric用户分别绑定baseRole和role1角色
- jack用户分别绑定baseRole和role2角色



保存。

创建项目测试权限

以itcast管理员账户创建两个项目，分别为itcast01和itheima01

结果为：

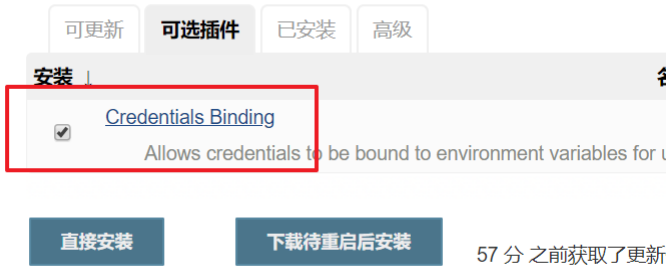
- eric用户登录，只能看到itcast01项目
- jack用户登录，只能看到itheima01项目

持续集成环境(4)-Jenkins凭证管理

凭据可以用来存储需要密文保护的数据库密码、Gitlab密码信息、Docker私有仓库密码等，以便Jenkins可以和这些第三方的应用进行交互。

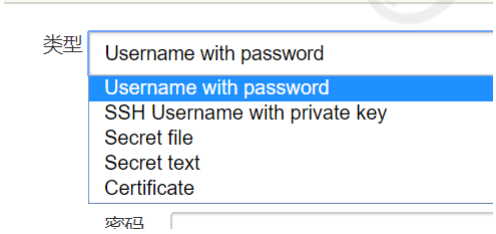
安装Credentials Binding插件

要在Jenkins使用凭证管理功能，需要安装Credentials Binding插件



安装插件后，左边多了"凭证"菜单，在这里管理所有凭证

可以添加的凭证有5种：



- Username with password：用户名和密码
- SSH Username with private key： 使用SSH用户和密钥
- Secret file：需要保密的文本文件，使用时Jenkins会将文件复制到一个临时目录中，再将文件路径设置到一个变量中，等构建结束后，所复制的Secret file就会被删除。
- Secret text：需要保存的一个加密的文本串，如钉钉机器人或Github的api token
- Certificate：通过上传证书文件的方式

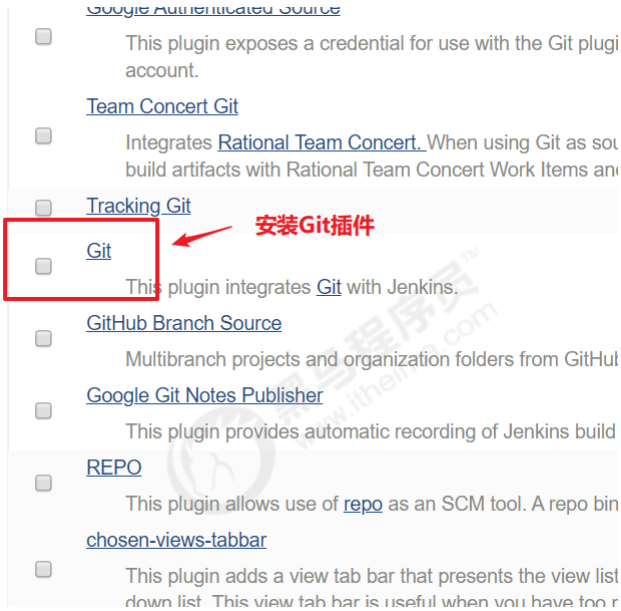
常用的凭证类型有：Username with password（用户密码）和SSH Username with private key（SSH密钥）

接下来以使用Git工具到Gitlab拉取项目源码为例，演示Jenkins的如何管理Gitlab的凭证。

安装Git插件和Git工具

为了让Jenkins支持从Gitlab拉取源码，需要安装Git插件以及在CentOS7上安装Git工具。

Git插件安装：



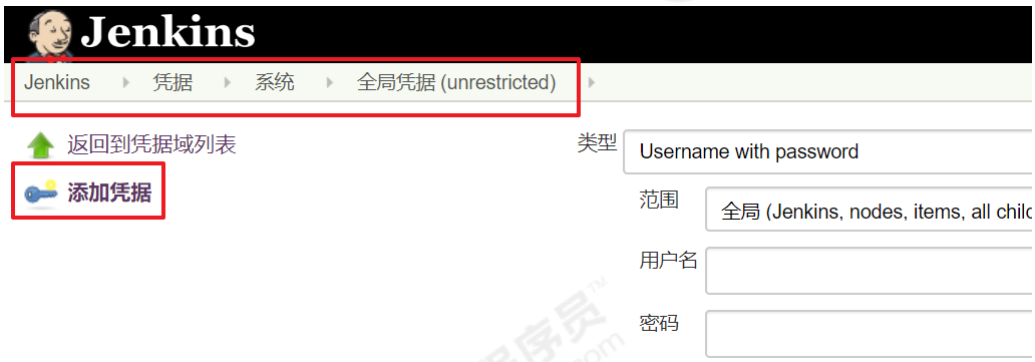
CentOS7上安装Git工具：

> yum install git -y 安装
>
> git --version 安装后查看版本

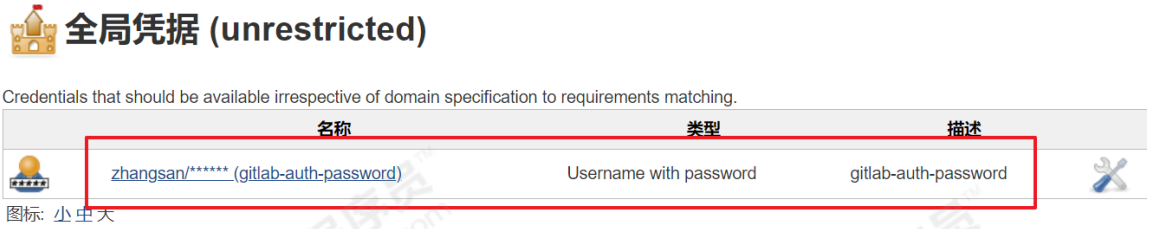
用户密码类型

1）创建凭证

Jenkins->凭证->系统->全局凭证->添加凭证



选择"Username with password"，输入Gitlab的用户名和密码，点击"确定"。

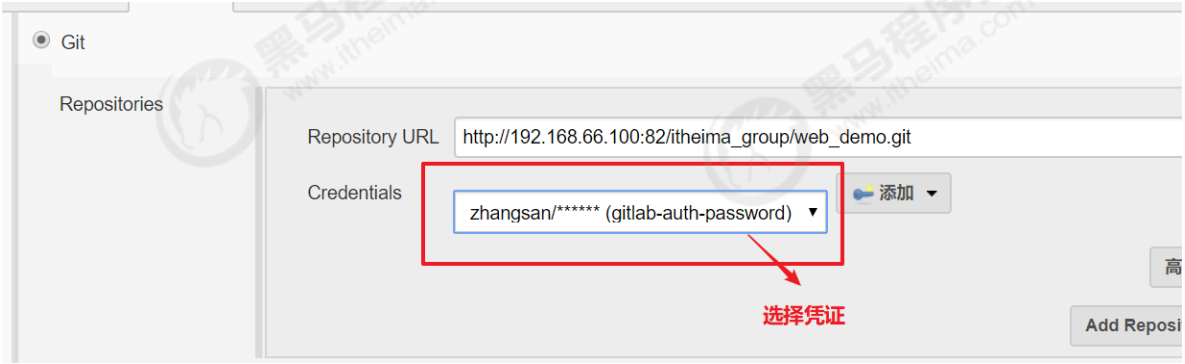


2）测试凭证是否可用

创建一个FreeStyle项目：新建Item->FreeStyle Project->确定

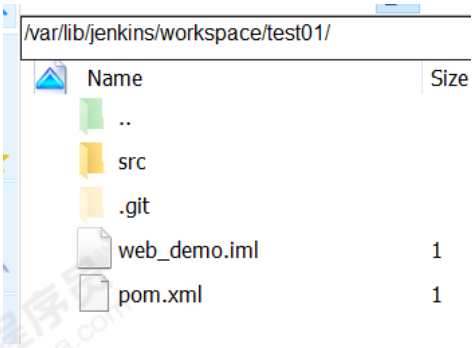
找到"源码管理"->"Git"，在Repository URL复制Gitlab中的项目URL

这时会报错说无法连接仓库！在Credentials选择刚刚添加的凭证就不报错啦



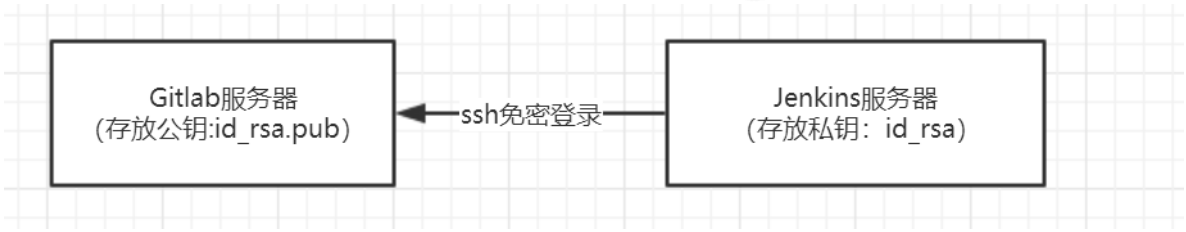
保存配置后，点击构建"Build Now" 开始构建项目

查看**/var/lib/jenkins/workspace/**目录，发现已经从Gitlab成功拉取了代码到enkins中。



SSH密钥类型

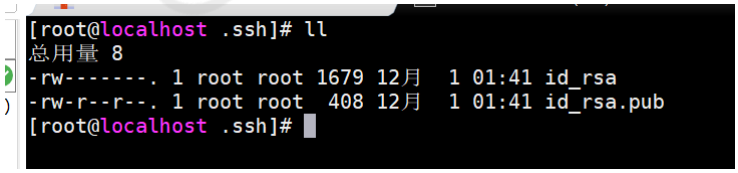
SSH免密登录示意图



1）使用root用户生成公钥和私钥

```
ssh-keygen -t rsa
```

在/root/.ssh/目录保存了公钥和使用



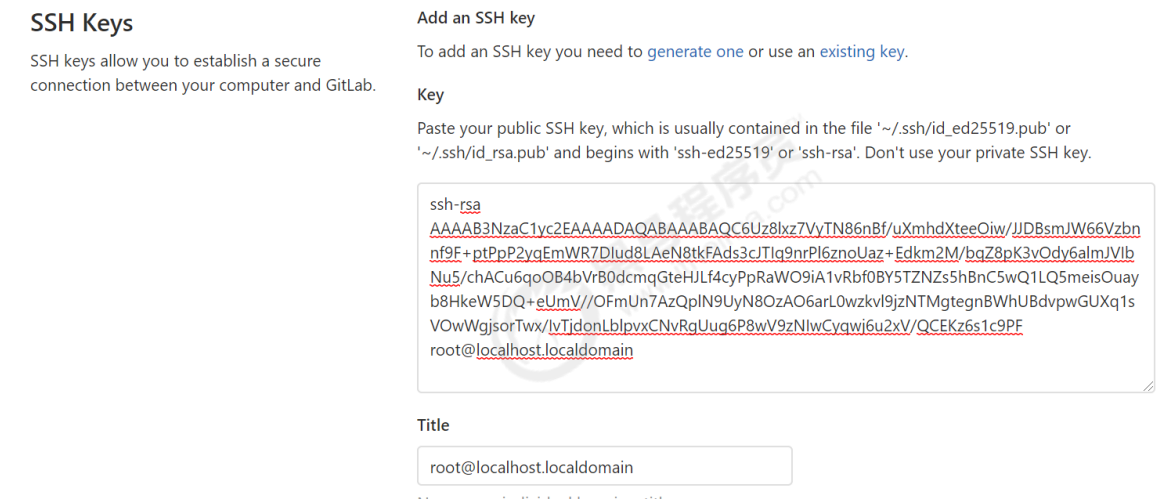
id_rsa：私钥文件

id_rsa.pub：公钥文件

2）把生成的公钥放在Gitlab中

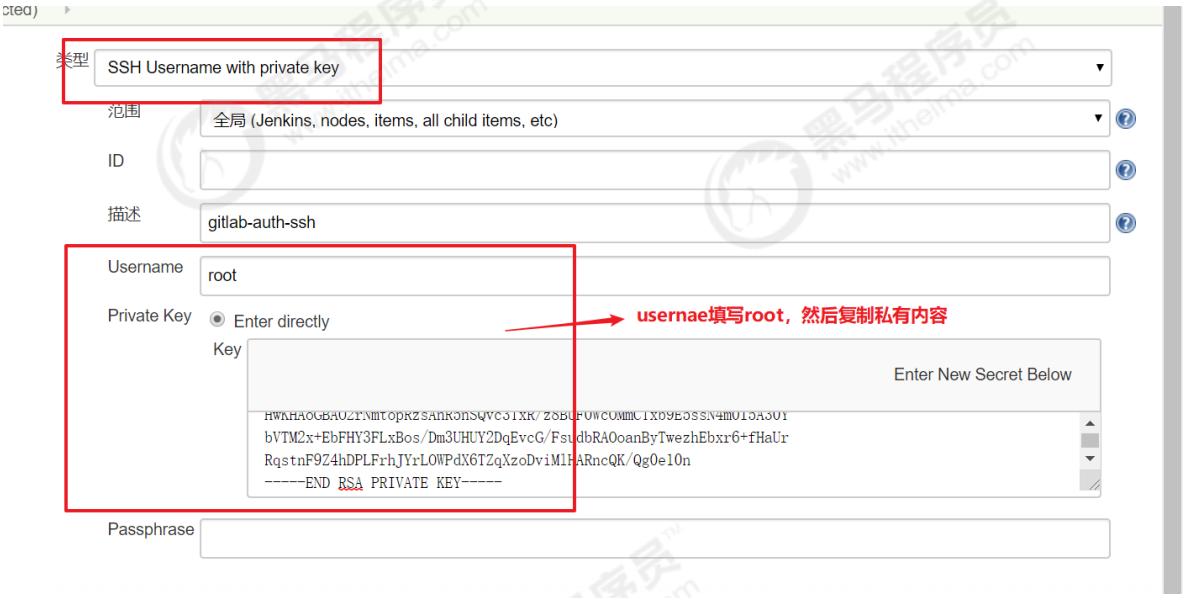
以root账户登录->点击头像->Settings->SSH Keys

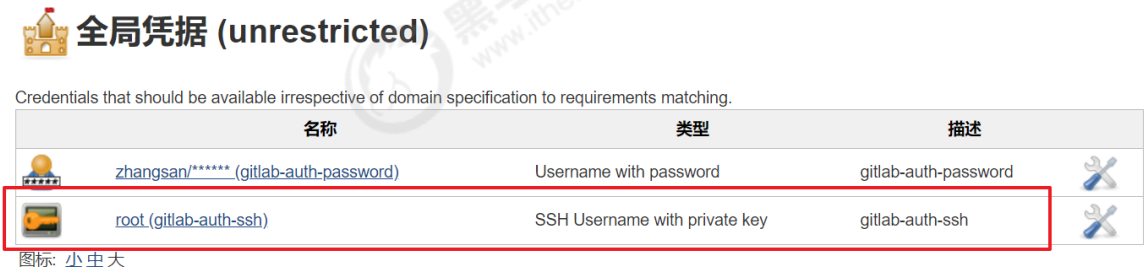
复制刚才id_rsa.pub文件的内容到这里，点击"Add Key"



3）在Jenkins中添加凭证，配置私钥

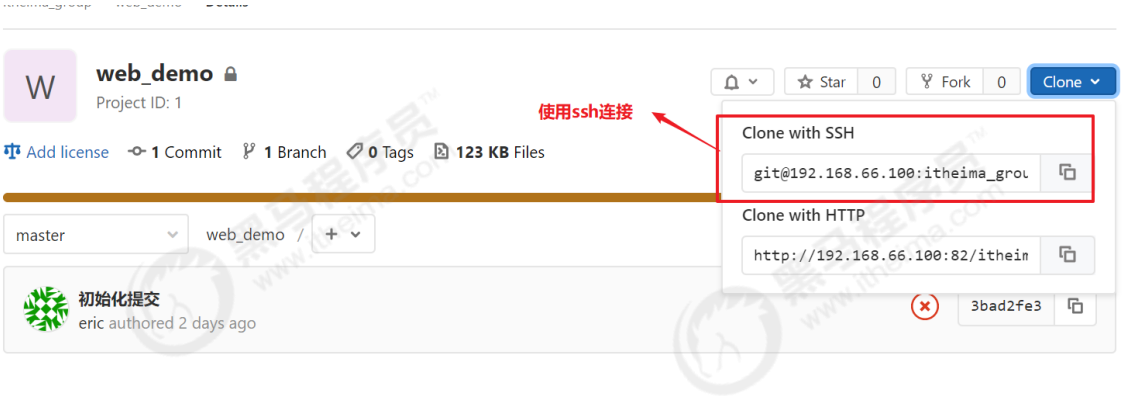
在Jenkins添加一个新的凭证，类型为"SSH Username with private key"，把刚才生成私有文件内容复制过来



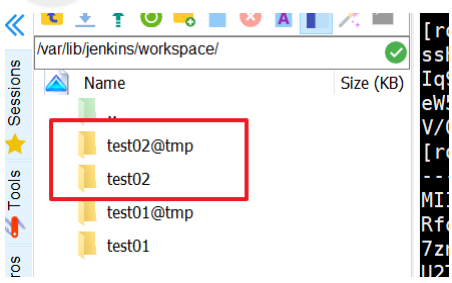


4）测试凭证是否可用

新建"test02"项目->源码管理->Git，这次要使用Gitlab的SSH连接，并且选择SSH凭证



同样尝试构建项目，如果代码可以正常拉取，代表凭证配置成功！



持续集成环境(5)-Maven安装和配置

在Jenkins集成服务器上，我们需要安装Maven来编译和打包项目。

安装Maven

先上传Maven软件到192.168.66.101

> tar -xzf apache-maven-3.6.2-bin.tar.gz 解压
>
> mkdir -p /opt/maven 创建目录
>
> mv apache-maven-3.6.2/* /opt/maven 移动文件

配置环境变量

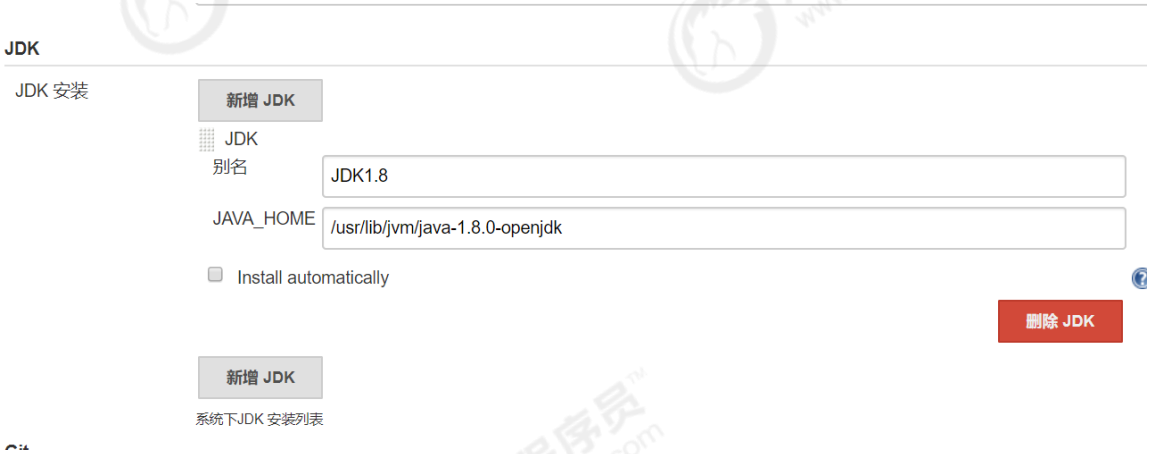
> vi /etc/profile

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk
export MAVEN_HOME=/opt/maven
export PATH=$PATH:$JAVA_HOME/bin:$MAVEN_HOME/bin
```

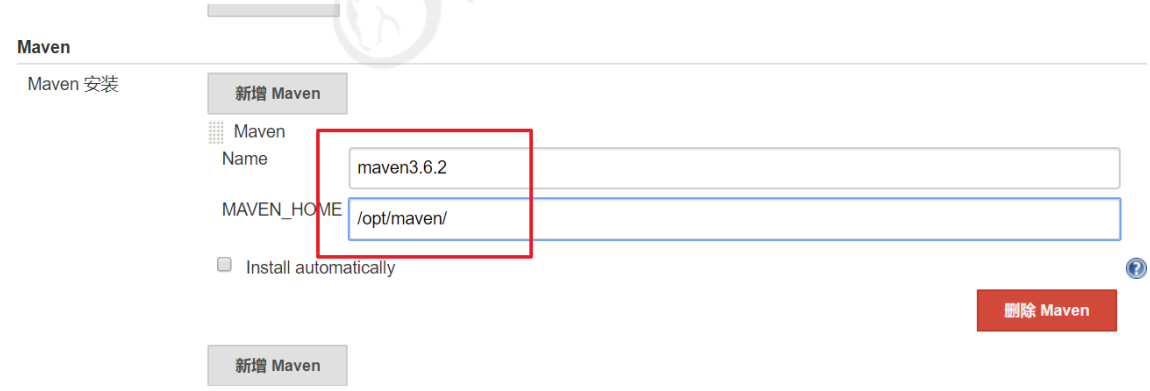
> source /etc/profile 配置生效
>
> mvn -v 查找Maven版本

全局工具配置关联JDK和Maven

Jenkins->Global Tool Configuration->JDK->新增JDK，配置如下：



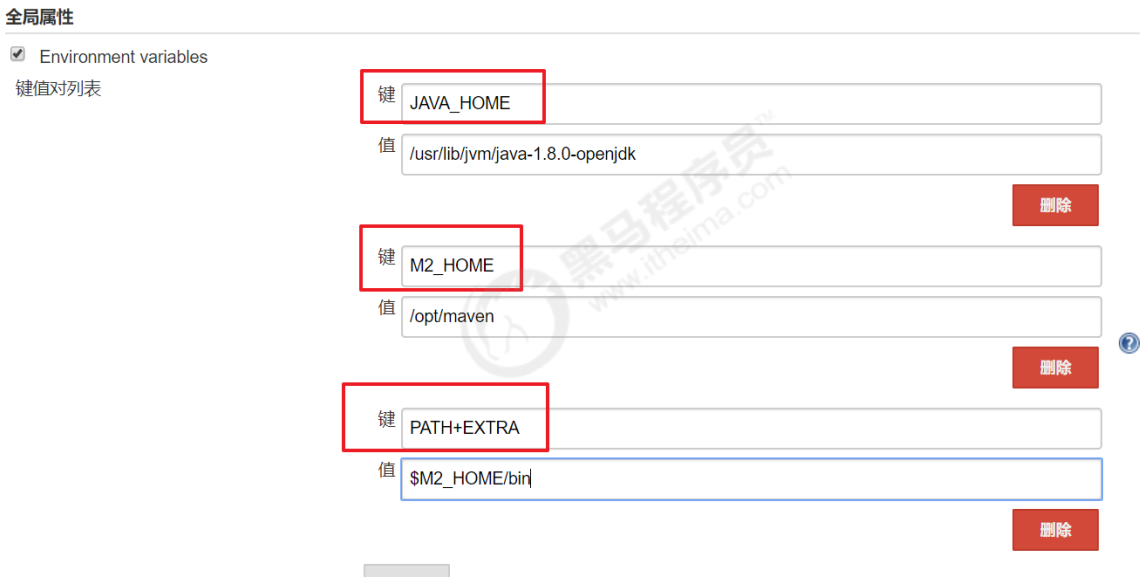
Jenkins->Global Tool Configuration->Maven->新增Maven，配置如下：



添加Jenkins全局变量

Manage Jenkins->Configure System->Global Properties ，添加三个全局变量

JAVA_HOME、M2_HOME、PATH+EXTRA



修改Maven的settings.xml

> mkdir /root/repo 创建本地仓库目录
>
> vi /opt/maven/conf/settings.xml

本地仓库改为：/root/repo/

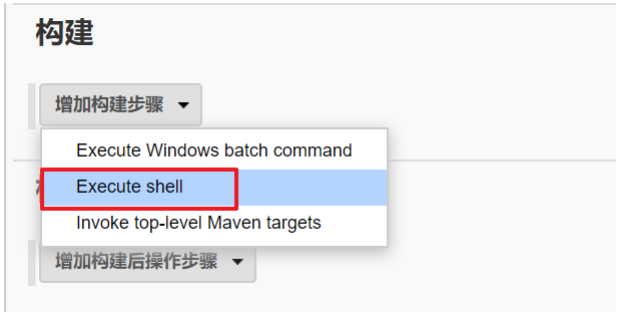
添加阿里云私服地址：

> alimaven aliyun maven http://maven.aliyun.com/nexus/content/groups/public/ central

测试Maven是否配置成功

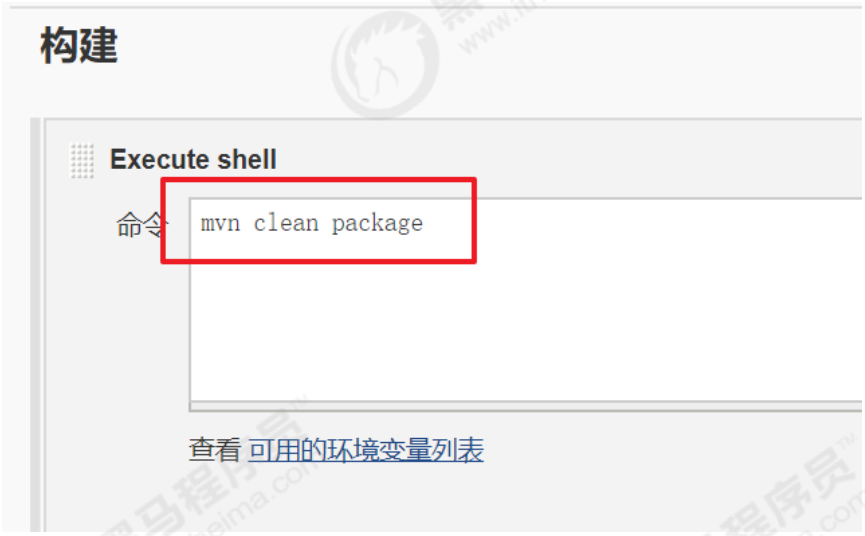
使用之前的gitlab密码测试项目，修改配置

构建->增加构建步骤->Execute Shell



输入

> mvn clean package



再次构建，如果可以把项目打成war包，代表maven环境配置成功啦！

```
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ web_demo ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-war-plugin:2.2:war (default-war) @ web_demo ---
[INFO] Packaging webapp
[INFO] Assembling webapp [web_demo] in [/var/lib/jenkins/workspace/test02/target/web_demo-1.0-SNAPSHOT]
[INFO] Processing war project
[INFO] Copying webapp resources [/var/lib/jenkins/workspace/test02/src/main/webapp]
[INFO] Webapp assembled in [61 msecs]
[INFO] Building war: /var/lib/jenkins/workspace/test02/target/web_demo-1.0-SNAPSHOT.war
[INFO] WEB-INF/web.xml already added, skipping
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.499 s
[INFO] Finished at: 2019-12-01T21:51:20+08:00
[INFO] ------------------------------------------------------------------------
Finished: SUCCESS
```

持续集成环境(6)-Tomcat安装和配置

安装Tomcat8.5

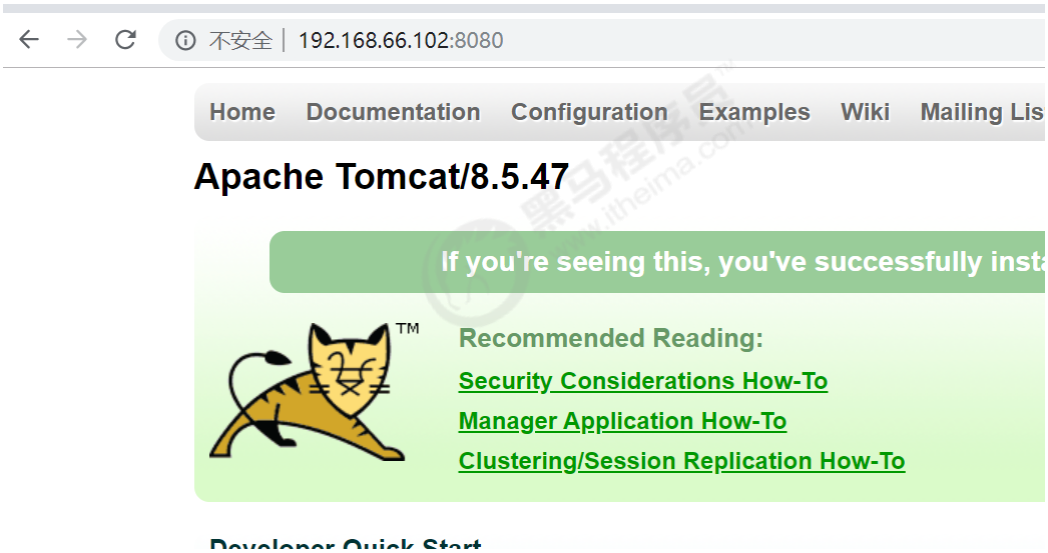
把Tomcat压缩包上传到192.168.66.102服务器

> yum install java-1.8.0-openjdk* -y 安装JDK（已完成）
>
> tar -xzf apache-tomcat-8.5.47.tar.gz 解压
>
> mkdir -p /opt/tomcat 创建目录

> mv /root/apache-tomcat-8.5.47/* /opt/tomcat 移动文件
>
> /opt/tomcat/bin/startup.sh 启动tomcat

注意：服务器已经关闭了防火墙，所以可以直接访问Tomcat啦

地址为：http://192.168.66.102/8080



配置Tomcat用户角色权限

默认情况下Tomcat是没有配置用户角色权限的

但是，后续Jenkins部署项目到Tomcat服务器，需要用到Tomcat的用户，所以修改tomcat以下配置，添加用户及权限

> vi /opt/tomcat/conf/tomcat-users.xml

内容如下：

```
<tomcat-users>
    <role rolename="tomcat"/>
    <role rolename="role1"/>
    <role rolename="manager-script"/>
    <role rolename="manager-gui"/>
    <role rolename="manager-status"/>
    <role rolename="admin-gui"/>
    <role rolename="admin-script"/>
    <user username="tomcat" password="tomcat" roles="manager-gui,manager-
script,tomcat,admin-gui,admin-script"/>
</tomcat-users>
```

用户和密码都是：tomcat

注意：为了能够刚才配置的用户登录到Tomcat，还需要修改以下配置

> vi /opt/tomcat/webapps/manager/META-INF/context.xml

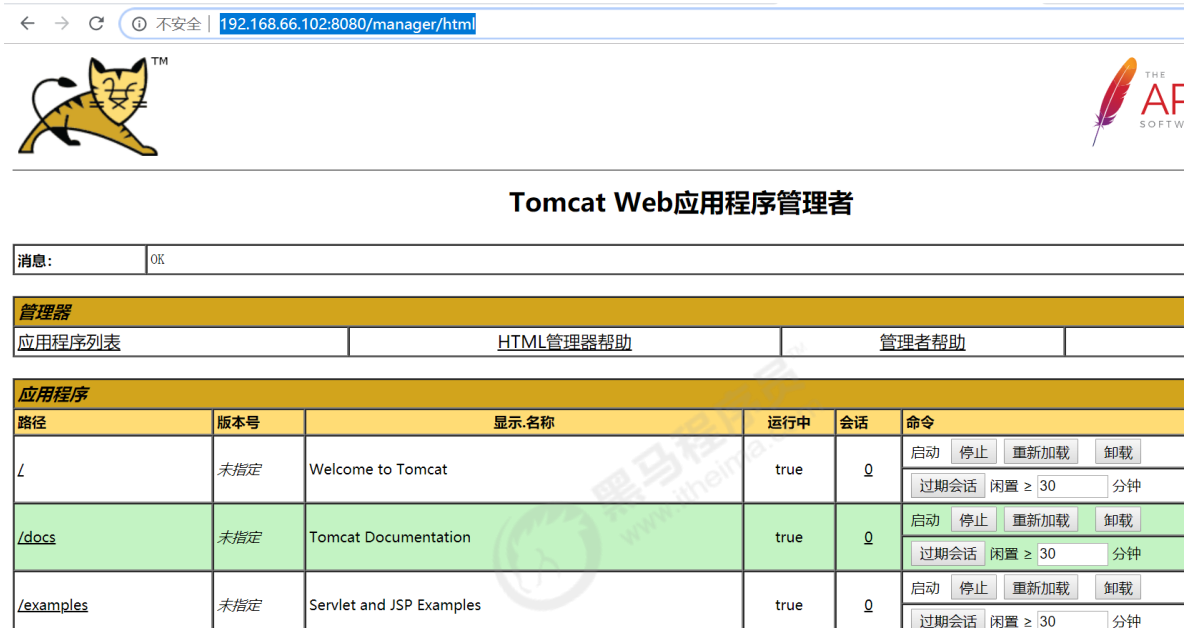
```
<!--
 <Valve className="org.apache.catalina.valves.RemoteAddrValve"
        allow="127\.\d+\.\d+\.\d+|::1|0:0:0:0:0:0:0:1" />
 -->
```

把上面这行注释掉即可！

重启Tomcat，访问测试

> /opt/tomcat/bin/shutdown.sh 停止
>
> /opt/tomcat/bin/startup.sh 启动

访问： http://192.168.66.102:8080/manager/html ，输入tomcat和tomcat，看到以下页面代表成功啦



3、Jenkins构建Maven项目

Jenkins项目构建类型(1)-Jenkins构建的项目类型介绍

Jenkins中自动构建项目的类型有很多，常用的有以下三种：

- 自由风格软件项目（FreeStyle Project）
- Maven项目（Maven Project）
- 流水线项目（Pipeline Project）

每种类型的构建其实都可以完成一样的构建过程与结果，只是在操作方式、灵活度等方面有所区别，在实际开发中可以根据自己的需求和习惯来选择。（PS：个人推荐使用流水线类型，因为灵活度非常高）

Jenkins项目构建类型(2)-自由风格项目构建

下面演示创建一个自由风格项目来完成项目的集成过程：

> 拉取代码->编译->打包->部署

拉取代码

1）创建项目

2）配置源码管理，从gitlab拉取代码

编译打包

构建->添加构建步骤->Executor Shell

```
echo "开始编译和打包"
mvn clean package
echo "编译和打包结束"
```

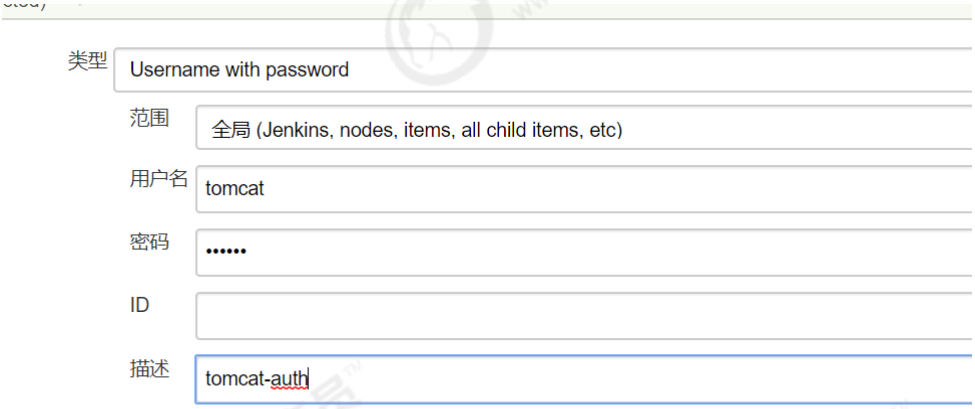
部署

把项目部署到远程的Tomcat里面

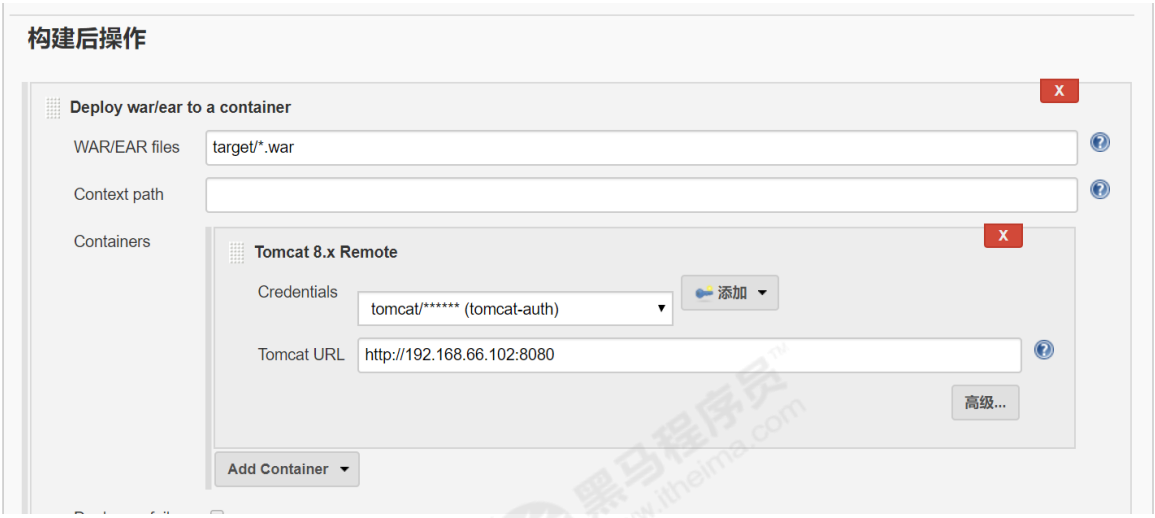
1）安装 Deploy to container插件

Jenkins本身无法实现远程部署到Tomcat的功能，需要安装Deploy to container插件实现

2）添加Tomcat用户凭证



3）添加构建后操作

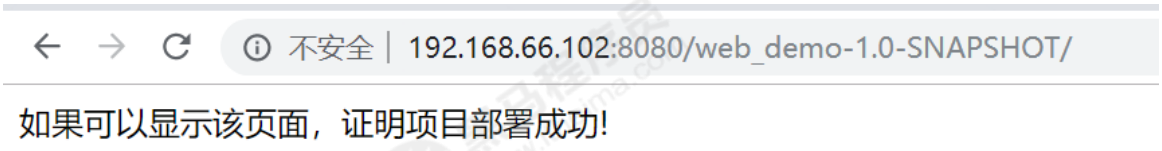


点击"Build Now"，开始构建过程

```
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ web_demo ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ web_demo ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-war-plugin:2.2:war (default-war) @ web_demo ---
[INFO] Packaging webapp
[INFO] Assembling webapp [web_demo] in [/var/lib/jenkins/workspace/web_demo_freestyle/target/web_demo-1.0-SNAPSHOT]
[INFO] Processing war project
[INFO] Copying webapp resources [/var/lib/jenkins/workspace/web_demo_freestyle/src/main/webapp]
[INFO] Webapp assembled in [64 msecs]
[INFO] Building war: /var/lib/jenkins/workspace/web_demo_freestyle/target/web_demo-1.0-SNAPSHOT.war
[INFO] WEB-INF/web.xml already added, skipping
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.721 s
[INFO] Finished at: 2019-12-01T23:45:36+08:00
[INFO] ------------------------------------------------------------------------
+ echo 编译和打包结束
编译和打包结束
[DeployPublisher][INFO] Attempting to deploy 1 war file(s)
[DeployPublisher][INFO] Deploying /var/lib/jenkins/workspace/web_demo_freestyle/target/web_demo-1.0-SNAPSHOT.war to container Tomcat
8.x Remote with context null
  [/var/lib/jenkins/workspace/web_demo_freestyle/target/web_demo-1.0-SNAPSHOT.war] is not deployed. Doing a fresh deployment.
  Deploying [/var/lib/jenkins/workspace/web_demo_freestyle/target/web_demo-1.0-SNAPSHOT.war]
Finished: SUCCESS
```

4）部署成功后，访问项目

http://192.168.66.102:8080/web_demo-1.0-SNAPSHOT/



如果可以显示该页面，证明项目部署成功!

演示改动代码后的持续集成

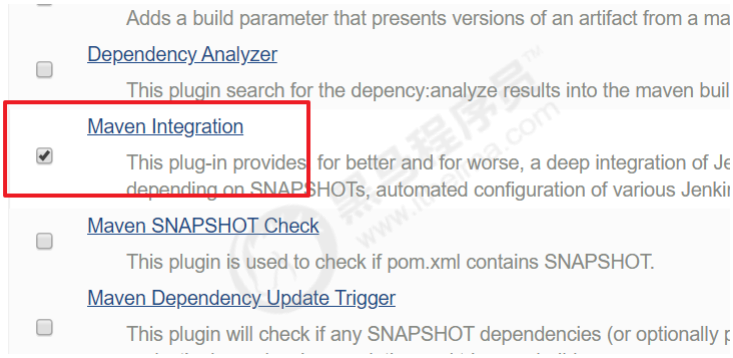
1）IDEA中源码修改并提交到gitlab

2）在Jenkins中项目重新构建

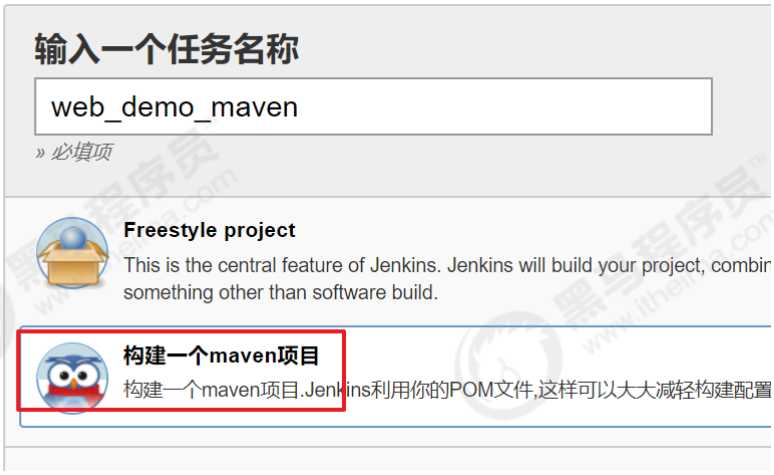
3）访问Tomcat

Jenkins项目构建类型(3)-Maven项目构建

1）安装Maven Integration插件



2）创建Maven项目



3）配置项目

拉取代码和远程部署的过程和自由风格项目一样，只是"构建"部分不同

Jenkins项目构建类型(4)-Pipeline流水线项目构建(*)

Pipeline简介

1）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。

2）使用Pipeline有以下好处（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。

3）如何创建 Jenkins Pipeline呢？

- 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Pipeline插件

Manage Jenkins->Manage Plugins->可选插件

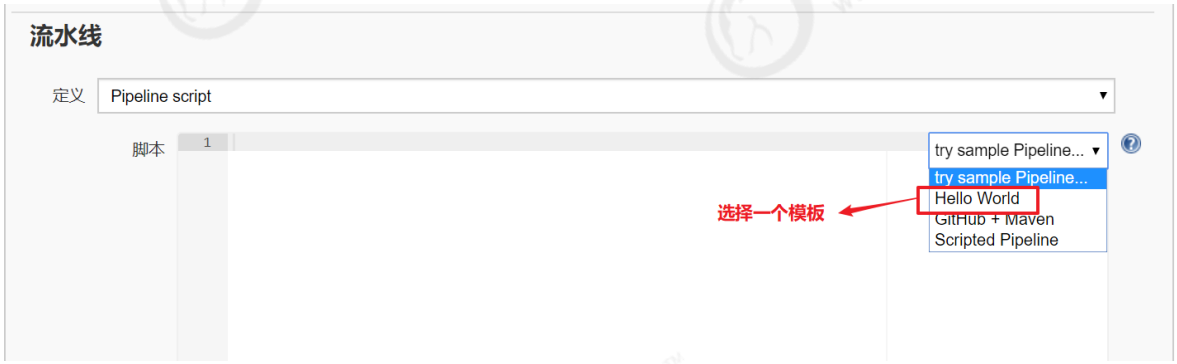
安装插件后，创建项目的时候多了“流水线”类型

Pipeline语法快速入门

1）Declarative声明式-Pipeline

创建项目

流水线->选择HelloWorld模板



生成内容如下：

```
pipeline {
    agent any

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

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

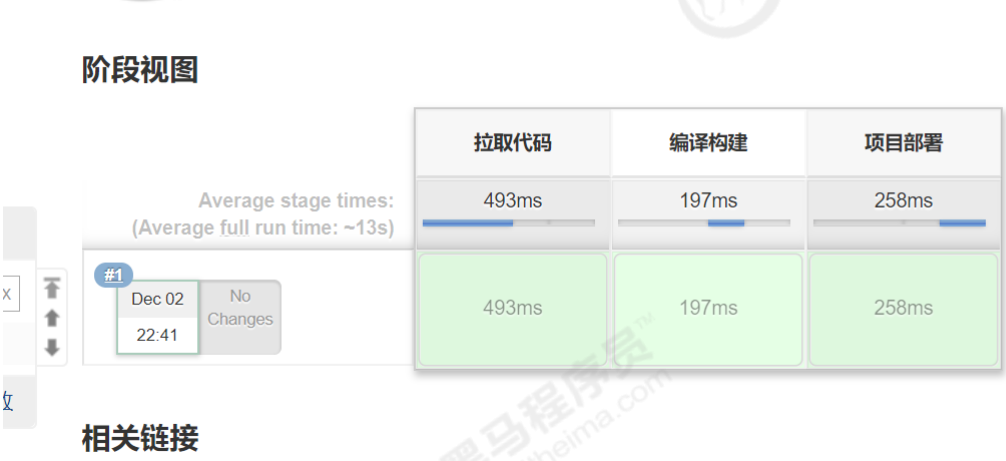
stage：代表流水线中的某个阶段，可能出现n个。一般分为拉取代码，编译构建，部署等阶段。

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。

编写一个简单声明式Pipeline：

```
pipeline {
    agent any

    stages {
        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'
            }
        }
    }
}
```

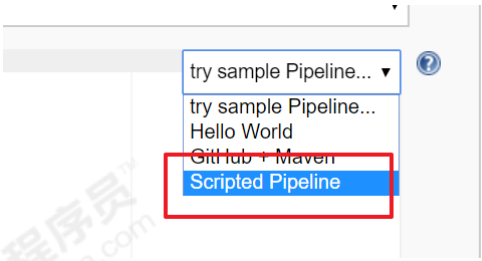
点击构建，可以看到整个构建过程



2) Scripted Pipeline脚本式-Pipeline

创建项目

这次选择"Scripted Pipeline"




```
node {
    def mvnHome
    stage('Preparation') { // for display purposes

    }
    stage('Build') {

    }
    stage('Results') {

    }
}
```

- Node：节点，一个 Node 就是一个 Jenkins 节点，Master 或者 Agent，是执行 Step 的具体运行环境，后续讲到Jenkins的Master-Slave架构的时候用到。
- Stage：阶段，一个 Pipeline 可以划分为若干个 Stage，每个 Stage 代表一组操作，比如：Build、Test、Deploy，Stage 是一个逻辑分组的概念。
- Step：步骤，Step 是最基本的操作单元，可以是打印一句话，也可以是构建一个 Docker 镜像，由各类 Jenkins 插件提供，比如命令：sh 'make'，就相当于我们平时 shell 终端中执行 make 命令一样。

编写一个简单的脚本式Pipeline

```
node {
   def mvnHome
   stage('拉取代码') { // for display purposes
      echo '拉取代码'
   }
   stage('编译构建') {
      echo '编译构建'
   }
   stage('项目部署') {
      echo '项目部署'
   }
}
```

构建结果和声明式一样！

拉取代码

```
pipeline {
   agent any

   stages {
      stage('拉取代码') {
         steps {
            checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '68f2087f-a034-4d39-a9ff-1f776dd3dfa8', url:
'git@192.168.66.100:itheima_group/web_demo.git']]])
         }
      }
   }
}
```

编译打包

```
pipeline {
   agent any

   stages {
      stage('拉取代码') {
         steps {
            checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '68f2087f-a034-4d39-a9ff-1f776dd3dfa8', url:
'git@192.168.66.100:itheima_group/web_demo.git']]])
         }
      }
      stage('编译构建') {
         steps {
            sh label: '', script: 'mvn clean package'
         }
      }
   }
}
```

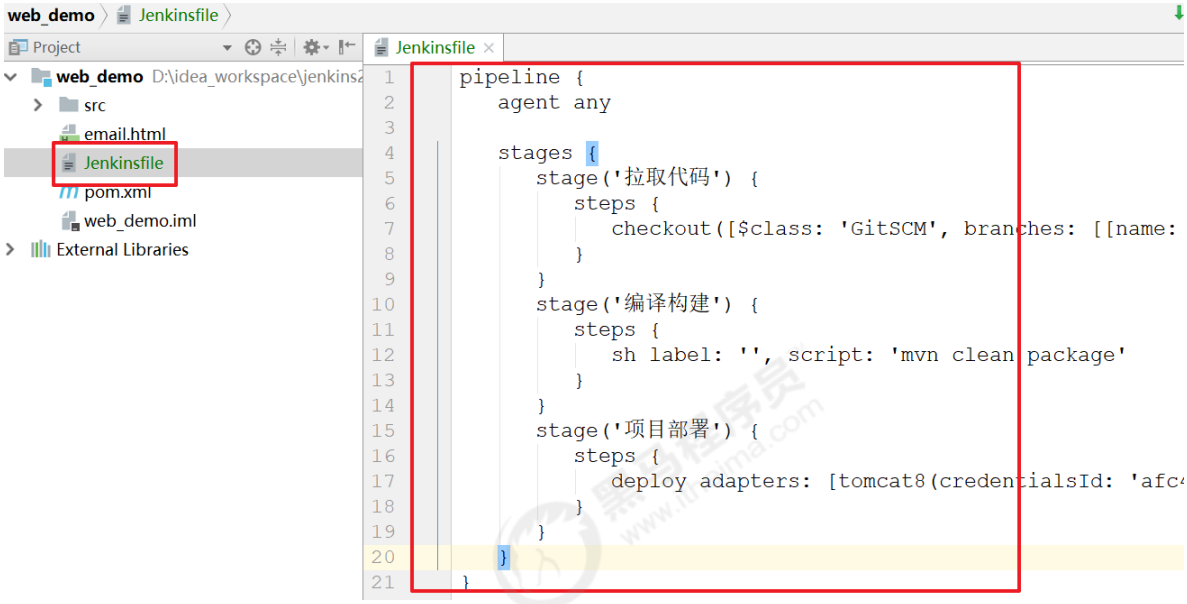
部署

```
pipeline {
   agent any

   stages {
      stage('拉取代码') {
         steps {
            checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '68f2087f-a034-4d39-a9ff-1f776dd3dfa8', url:
'git@192.168.66.100:itheima_group/web_demo.git']]])
         }
      }
      stage('编译构建') {
         steps {
            sh label: '', script: 'mvn clean package'
         }
      }
      stage('项目部署') {
         steps {
            deploy adapters: [tomcat8(credentialsId: 'afc43e5e-4a4e-4de6-984f-
b1d5a254e434', path: '', url: 'http://192.168.66.102:8080')], contextPath: null,
war: 'target/*.war'
         }
      }
   }
}
```

Pipeline Script from SCM

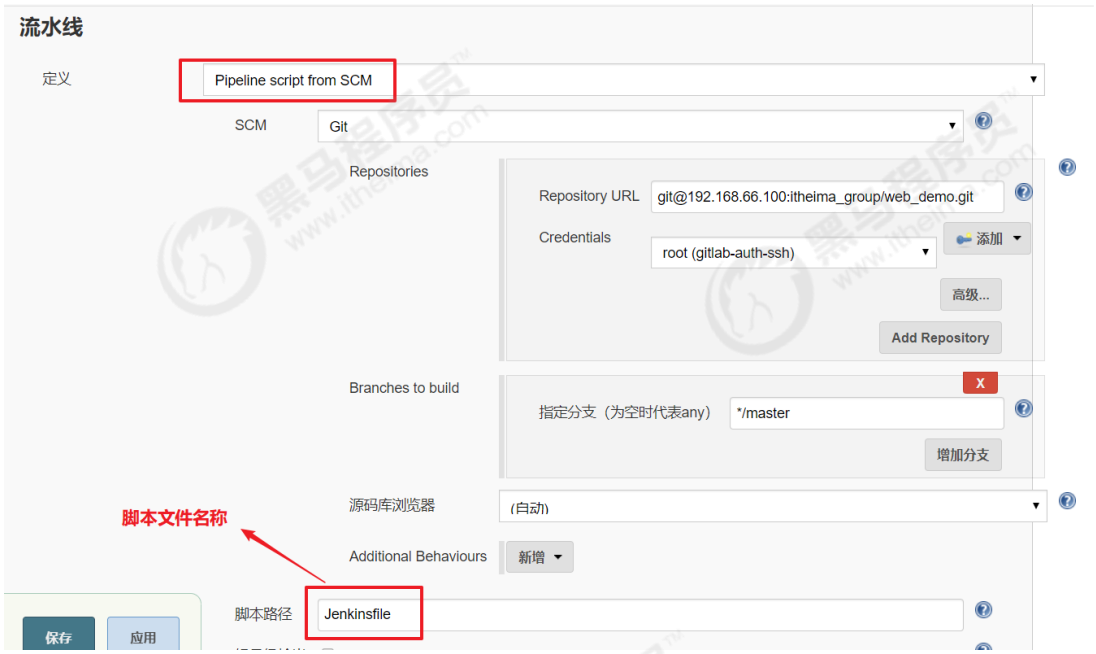
刚才我们都是直接在Jenkins的UI界面编写Pipeline代码，这样不方便脚本维护，建议把Pipeline脚本放在项目中（一起进行版本控制）

1）在项目根目录建立Jenkinsfile文件，把内容复制到该文件中



把Jenkinsfile上传到Gitlab

2）在项目中引用该文件



Jenkins项目构建细节(1)-常用的构建触发器

Jenkins内置4种构建触发器：

- 触发远程构建
- 其他工程构建后触发（Build after other projects are build）
- 定时构建（Build periodically）
- 轮询SCM（Poll SCM）

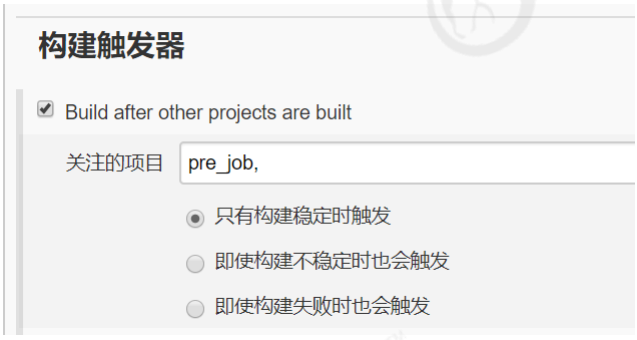
触发远程构建

触发构建url：http://192.168.66.101:8888/job/web_demo_pipeline/build?token=6666

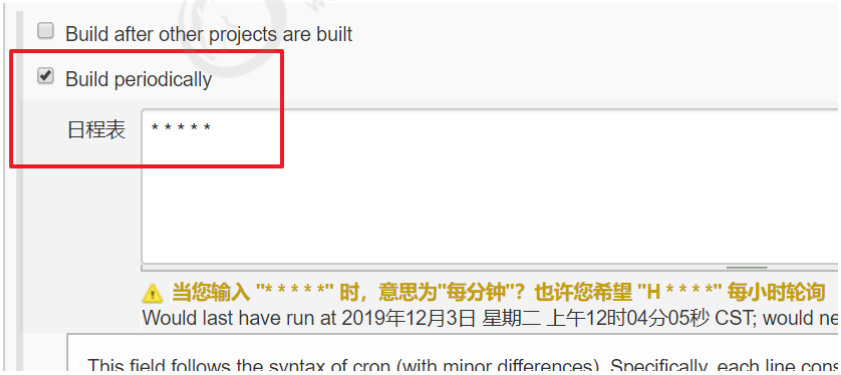
其他工程构建后触发

1）创建pre_job流水线工程

2）配置需要触发的工程



定时构建



定时字符串从左往右分别为： 分 时 日 月 周

一些定时表达式的例子：

> 每30分钟构建一次：H代表形参 H/30 * * * * 10:02 10:32
>
> 每2个小时构建一次: H H/2 * * *
>
> 每天的8点，12点，22点，一天构建3次： (多个时间点中间用逗号隔开) 0 8,12,22 * * *
>
> 每天中午12点定时构建一次 H 12 * * *
>
> 每天下午18点定时构建一次 H 18 * * *
>
> 在每个小时的前半个小时内的每10分钟 H(0-29)/10 * * * *
>
> 每两小时一次，每个工作日上午9点到下午5点(也许是上午10:38，下午12:38，下午2:38，下午4:38) H H(9-16)/2 * * 1-5

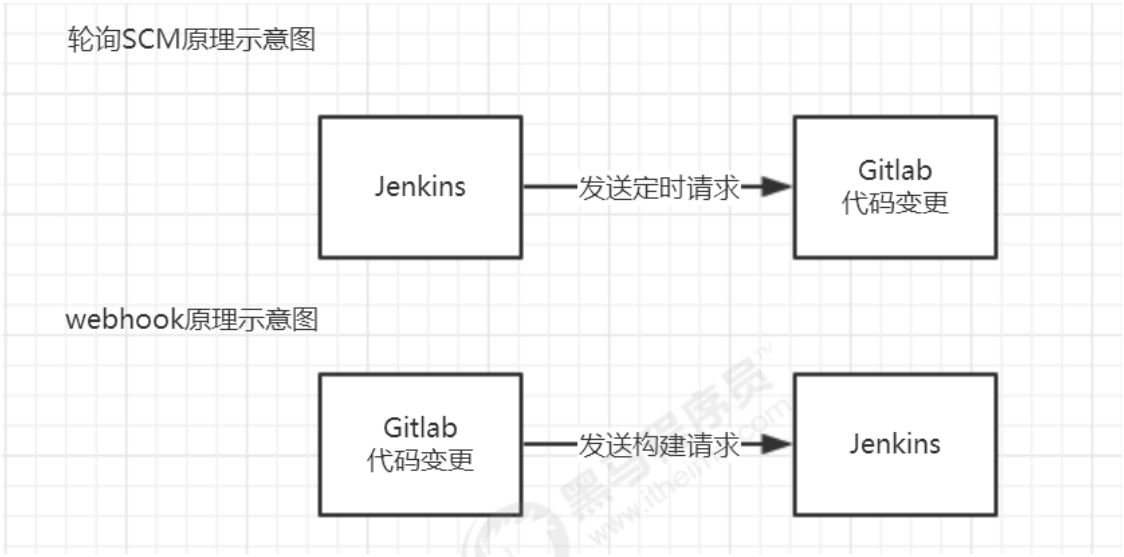
轮询SCM

轮询SCM，是指定时扫描本地代码仓库的代码是否有变更，如果代码有变更就触发项目构建。

注意：这次构建触发器，Jenkins会定时扫描本地整个项目的代码，增大系统的开销，不建议使用。

Jenkins项目构建细节(2)-Git hook自动触发构建(*)

刚才我们看到在Jenkins的内置构建触发器中，轮询SCM可以实现Gitlab代码更新，项目自动构建，但是该方案的性能不佳。那有没有更好的方案呢？ 有的。就是利用Gitlab的webhook实现代码push到仓库，立即触发项目自动构建。

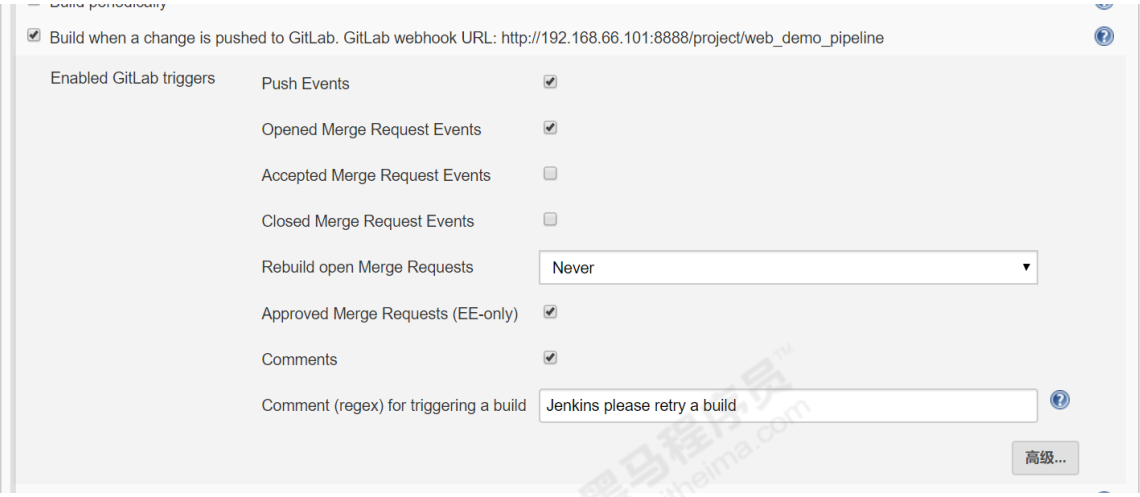


安装Gitlab Hook插件

需要安装两个插件：

Gitlab Hook和GitLab

Jenkins设置自动构建



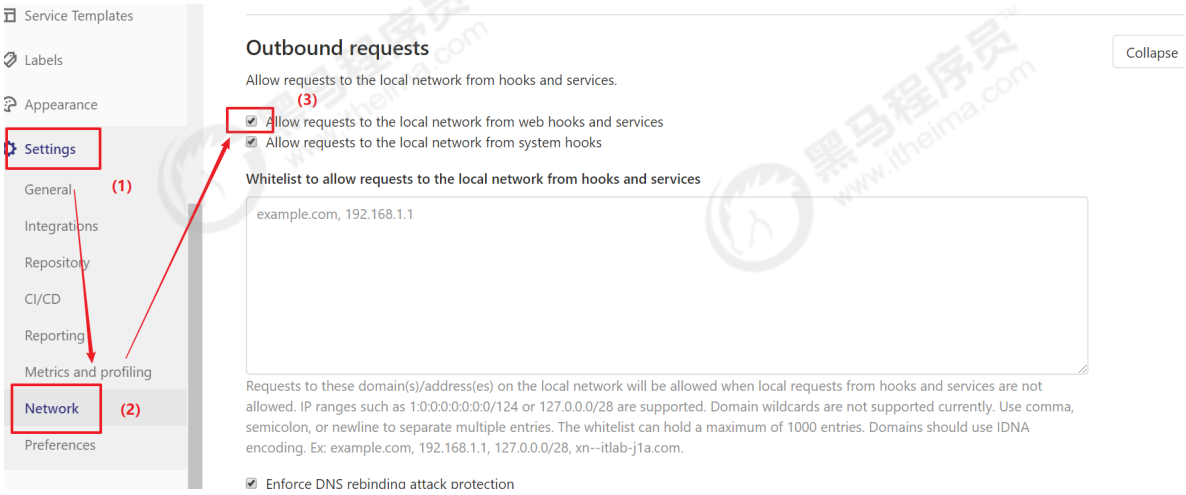
等会需要把生成的webhook URL配置到Gitlab中。

Gitlab配置webhook

1）开启webhook功能

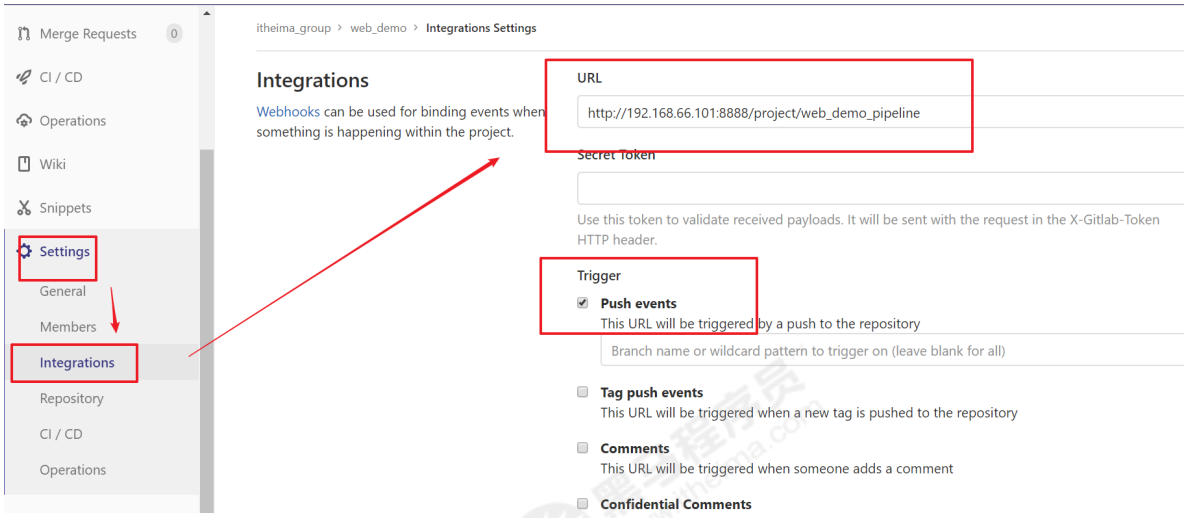
使用root账户登录到后台，点击Admin Area -> Settings -> Network

勾选"Allow requests to the local network from web hooks and services"



2）在项目添加webhook

点击项目->Settings->Integrations



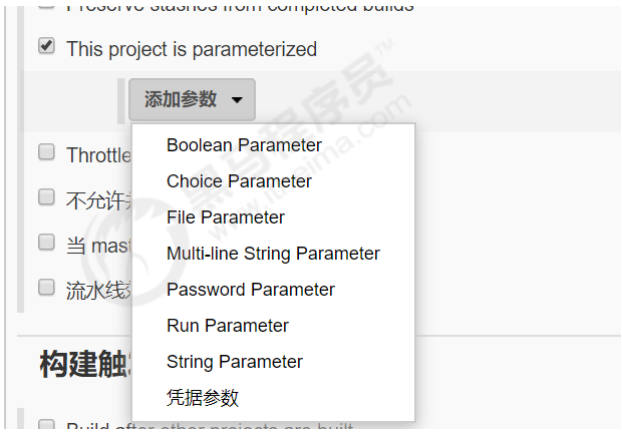
注意：以下设置必须完成，否则会报错！

Manage Jenkins->Configure System

Jenkins项目构建细节(3)-Jenkins的参数化构建

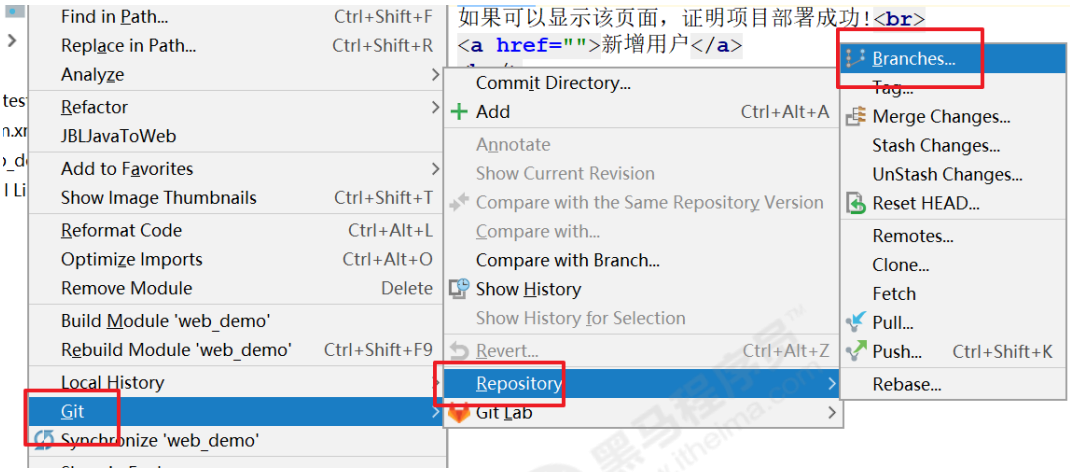
有时在项目构建的过程中，我们需要根据用户的输入动态传入一些参数，从而影响整个构建结果，这时我们可以使用参数化构建。

Jenkins支持非常丰富的参数类型



接下来演示通过输入gitlab项目的分支名称来部署不同分支项目。

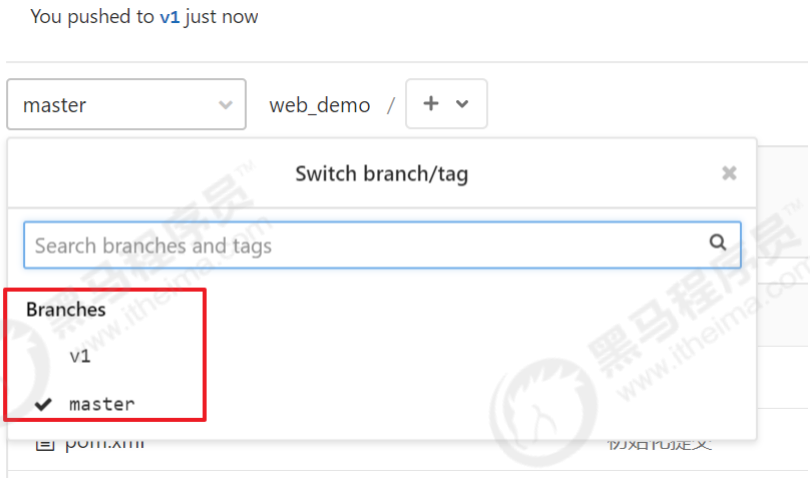
项目创建分支，并推送到Gitlab上



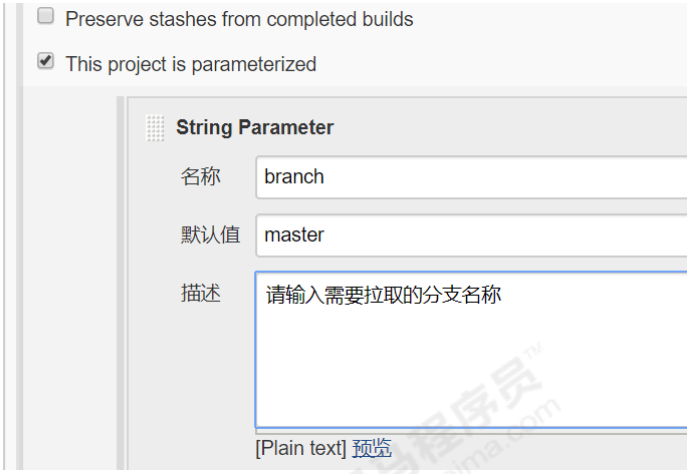
新

新建分支：v1，代码稍微改动下，然后提交到gitlab上。

这时看到gitlab上有一个两个分支：master和v1



在Jenkins添加字符串类型参数



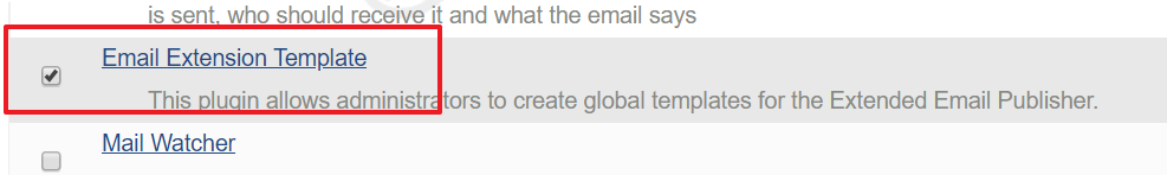
改动pipeline流水线代码

点击Build with Parameters

输入分支名称，构建即可！构建完成后访问Tomcat查看结果

Jenkins项目构建细节(4)-配置邮箱服务器发送构建结果

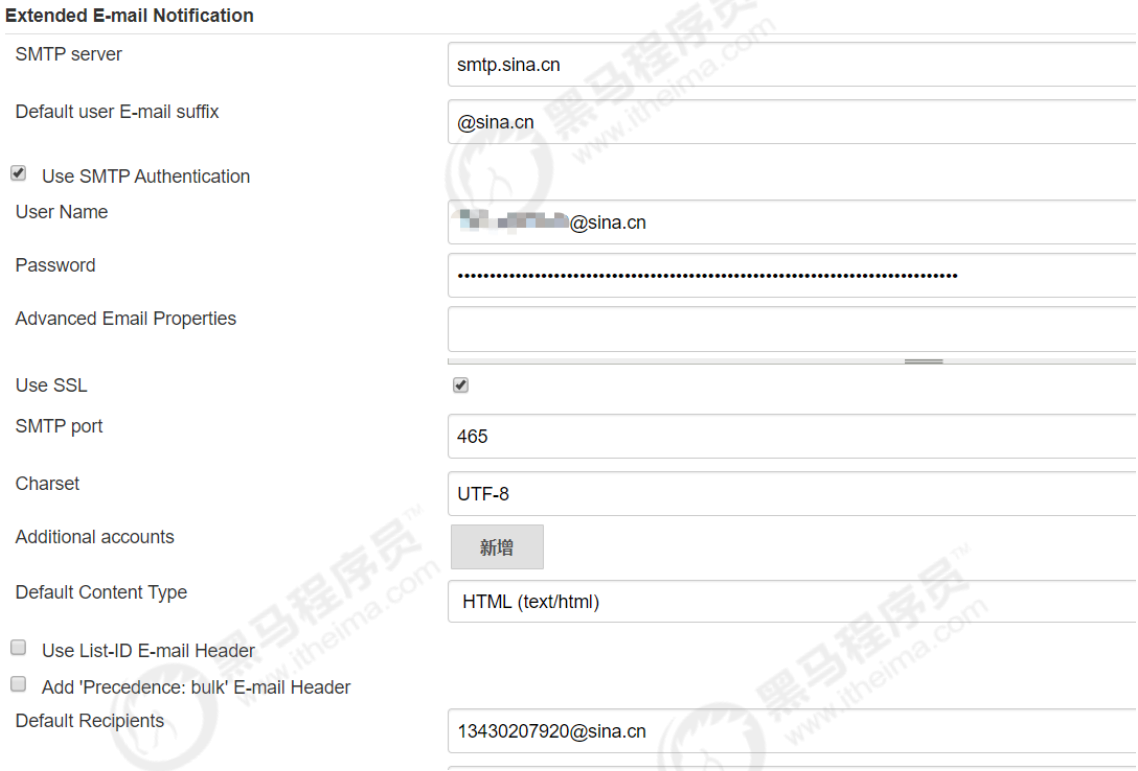
安装Email Extension插件



Jenkins设置邮箱相关参数

Manage Jenkins->Configure System

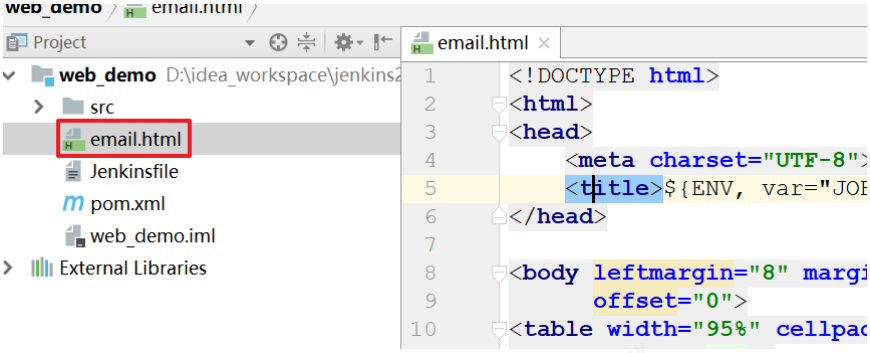
设置邮件参数



设置Jenkins默认邮箱信息

准备邮件内容

在项目根目录编写email.html，并把文件推送到Gitlab，内容如下：




```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>${ENV, var="JOB_NAME"}-第${BUILD_NUMBER}次构建日志</title>
</head>

<body leftmargin="8" marginwidth="0" topmargin="8" marginheight="4"
      offset="0">
<table width="95%" cellpadding="0" cellspacing="0"
       style="font-size: 11pt; font-family: Tahoma, Arial, Helvetica, sans-
serif">
    <tr>
        <td>(本邮件是程序自动下发的，请勿回复！)</td>
    </tr>
    <tr>
        <td><h2>
            <font color="#0000FF">构建结果 - ${BUILD_STATUS}</font>
        </h2></td>
    </tr>
    <tr>
        <td><br />
            <b><font color="#0B610B">构建信息</font></b>
            <hr size="2" width="100%" align="center" /></td>
    </tr>
    <tr>
        <td>
            <ul>
                <li>项目名称 :  ${PROJECT_NAME}</li>
                <li>构建编号 :  第${BUILD_NUMBER}次构建</li>
                <li>触发原因:  ${CAUSE}</li>
                <li>构建日志:  <a
href="${BUILD_URL}console">${BUILD_URL}console</a></li>
                <li>构建  Url :  <a
href="${BUILD_URL}">${BUILD_URL}</a></li>
                <li>工作目录 :  <a
href="${PROJECT_URL}ws">${PROJECT_URL}ws</a></li>
                <li>项目  Url :  <a
href="${PROJECT_URL}">${PROJECT_URL}</a></li>
            </ul>
        </td>
    </tr>
    <tr>
        <td><b><font color="#0B610B">Changes Since Last
            Successful Build:</font></b>
            <hr size="2" width="100%" align="center" /></td>
    </tr>
```

```
    <tr>
        <td>
            <ul>
                <li>历史变更记录 : <a
href="${PROJECT_URL}changes">${PROJECT_URL}changes</a></li>
            </ul> ${CHANGES_SINCE_LAST_SUCCESS,reverse=true, format="Changes for
Build #%n:<br />%c<br />",showPaths=true,changesFormat="<pre>[%a]<br
/>%m</pre>",pathFormat="    %p"}
        </td>
    </tr>
    <tr>
        <td><b>Failed Test Results</b>
            <hr size="2" width="100%" align="center" /></td>
    </tr>
    <tr>
        <td><pre
                style="font-size: 11pt; font-family: Tahoma, Arial, Helvetica,
sans-serif">$FAILED_TESTS</pre>
            <br /></td>
    </tr>
    <tr>
        <td><b><font color="#0B610B">构建日志 (最后 100行):</font></b>
            <hr size="2" width="100%" align="center" /></td>
    </tr>
    <tr>
        <td><textarea cols="80" rows="30" readonly="readonly"
                      style="font-family: Courier New">${BUILD_LOG,
maxLines=100}</textarea>
        </td>
    </tr>
</table>
</body>
</html>
```

编写Jenkinsfile添加构建后发送邮件

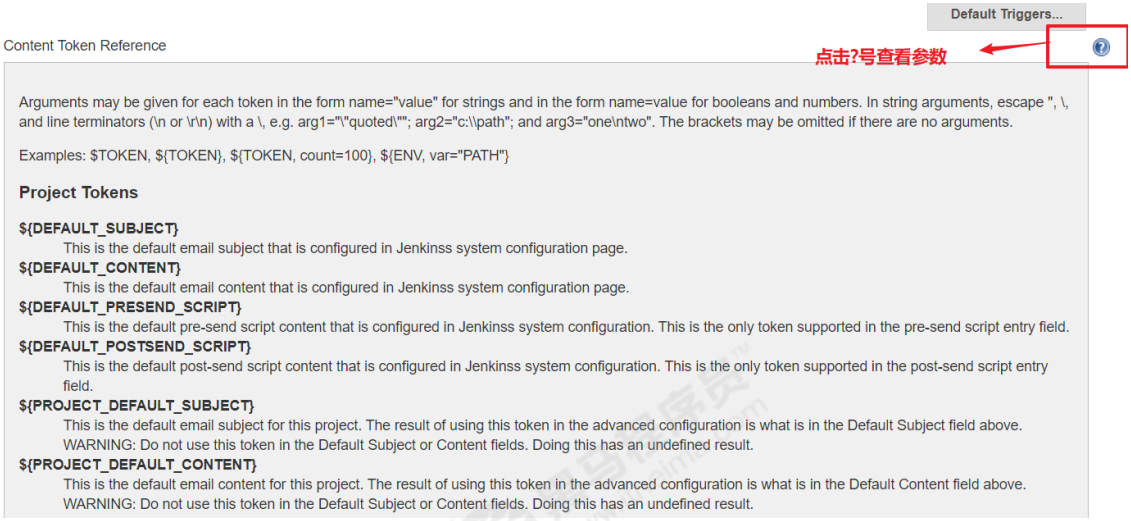
```
pipeline {
   agent any

   stages {
      stage('拉取代码') {
         steps {
            checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '68f2087f-a034-4d39-a9ff-1f776dd3dfa8', url:
'git@192.168.66.100:itheima_group/web_demo.git']]])
         }
      }
      stage('编译构建') {
         steps {
            sh label: '', script: 'mvn clean package'
         }
      }
      stage('项目部署') {
         steps {
```

```
            deploy adapters: [tomcat8(credentialsId: 'afc43e5e-4a4e-4de6-984f-
b1d5a254e434', path: '', url: 'http://192.168.66.102:8080')], contextPath: null,
war: 'target/*.war'
         }
      }
   }
   post {
      always {
         emailext(
            subject: '构建通知：${PROJECT_NAME} - Build # ${BUILD_NUMBER} -
${BUILD_STATUS}!',
            body: '${FILE,path="email.html"}',
            to: 'xxx@qq.com'
         )
      }
   }
}
```

测试

PS：邮件相关全局参数参考列表：

系统设置->Extended E-mail Notification->Content Token Reference，点击旁边的?号



Jenkins+SonarQube代码审查(1) - 安装SonarQube

SonaQube简介

SonarQube是一个用于管理代码质量的开放平台，可以快速的定位代码中潜在的或者明显的错误。目前支持java,C#,C/C++,Python,PL/SQL,Cobol,JavaScrip,Groovy等二十几种编程语言的代码质量管理与检测。

官网：https://www.sonarqube.org/

环境要求

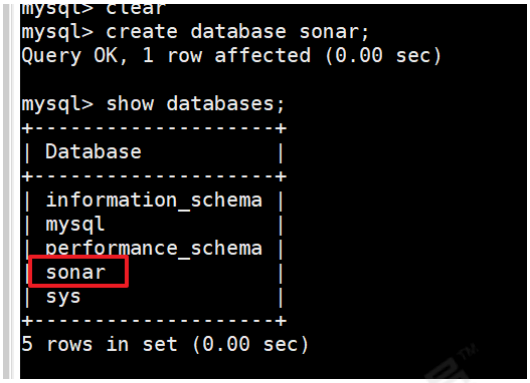
软件	服务器	版本
JDK	192.168.66.101	1.8
MySQL	192.168.66.101	5.7
SonarQube	192.168.66.101	6.7.4

安装SonarQube

1）安装MySQL（已完成）

2）安装SonarQube

在MySQL创建sonar数据库



下载sonar压缩包：

https://www.sonarqube.org/downloads/

解压sonar，并设置权限

> yum install unzip
>
> unzip sonarqube-6.7.4.zip 解压
>
> mkdir /opt/sonar 创建目录
>
> mv sonarqube-6.7.4/* /opt/sonar 移动文件
>
> useradd sonar 创建sonar用户，必须sonar用于启动，否则报错
>
> chown -R sonar. /opt/sonar 更改sonar目录及文件权限

修改sonar配置文件

> vi /opt/sonarqube-6.7.4/conf/sonar.properties
>
> 内容如下：
>
> sonar.jdbc.username=root sonar.jdbc.password=Root@123
>
> sonar.jdbc.url=jdbc:mysql://localhost:3306/sonar?useUnicode=true&characterEncoding=utf8&rewriteBatchedStatements=true&useConfigs=maxPerformance&useSSL=false

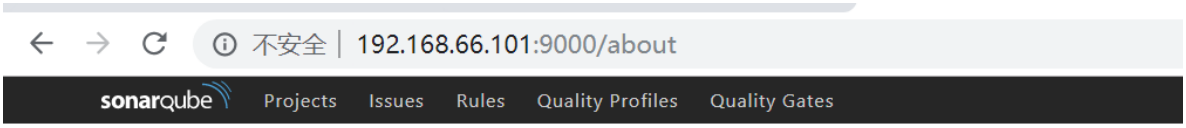
注意：sonar默认监听9000端口，如果9000端口被占用，需要更改。

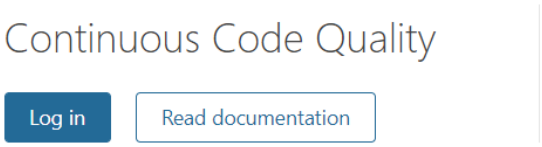
启动sonar

> cd /opt/sonarqube-6.7.4
>
> su sonar ./bin/linux-x86-64/sonar.sh start 启动
>
> su sonar ./bin/linux-x86-64/sonar.sh status 查看状态
>
> su sonar ./bin/linux-x86-64/sonar.sh stop 停止
>
> tail -f logs/sonar.logs 查看日志

访问sonar

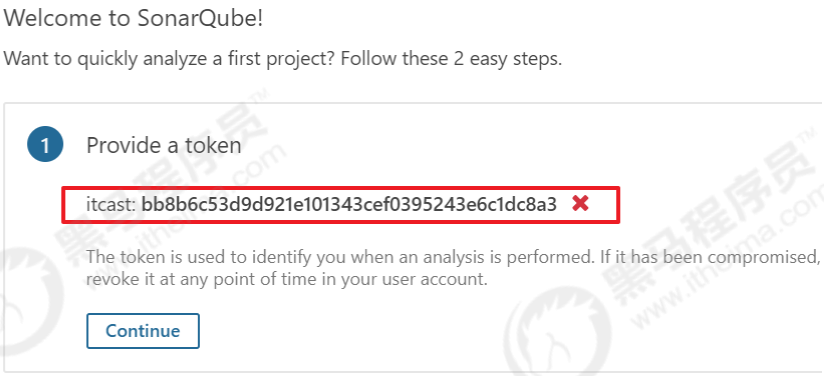
http://192.168.66.101:9000





默认账户：admin/admin

创建token

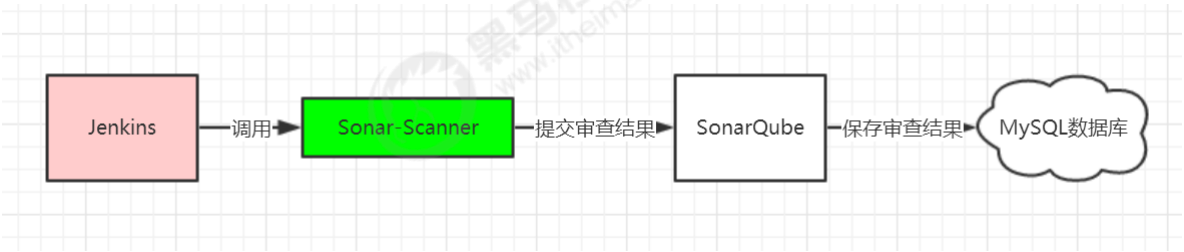


> **bb8b6c53d9d921e101343cef0395243e6c1dc8a3**

token要记下来后面要使用

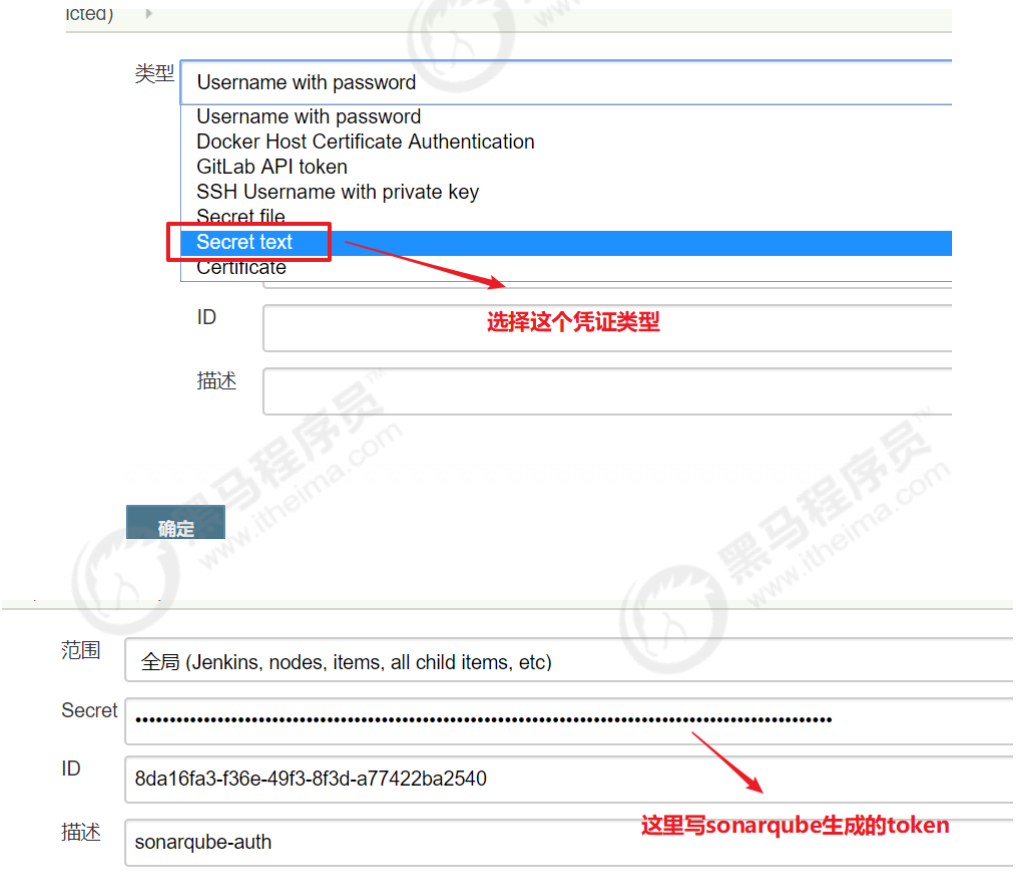
0151ae8c548a143eda9253e4334ad030b56047ee

Jenkins+SonarQube代码审查(2) - 实现代码审查



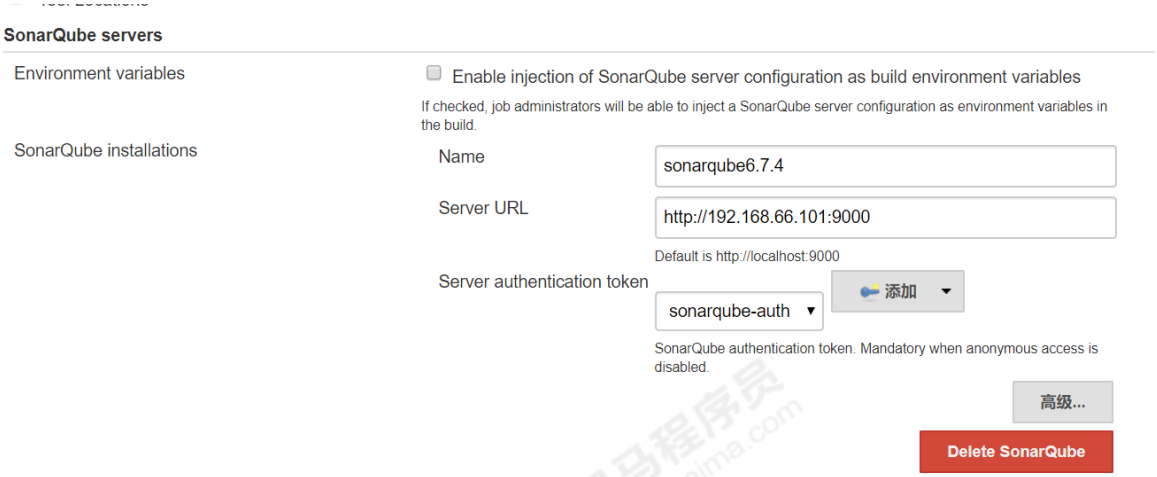
安装SonarQube Scanner插件

添加SonarQube凭证

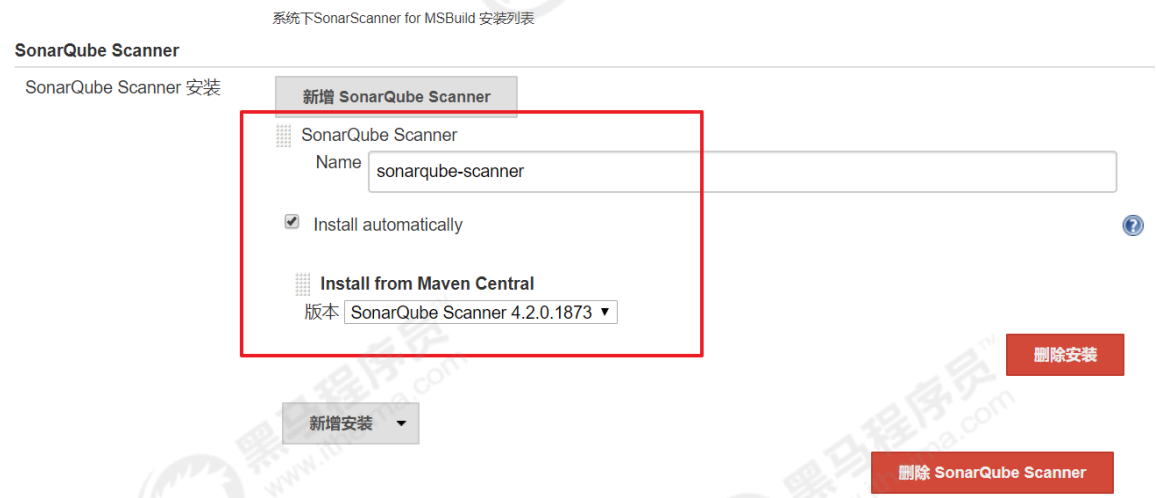


Jenkins进行SonarQube配置

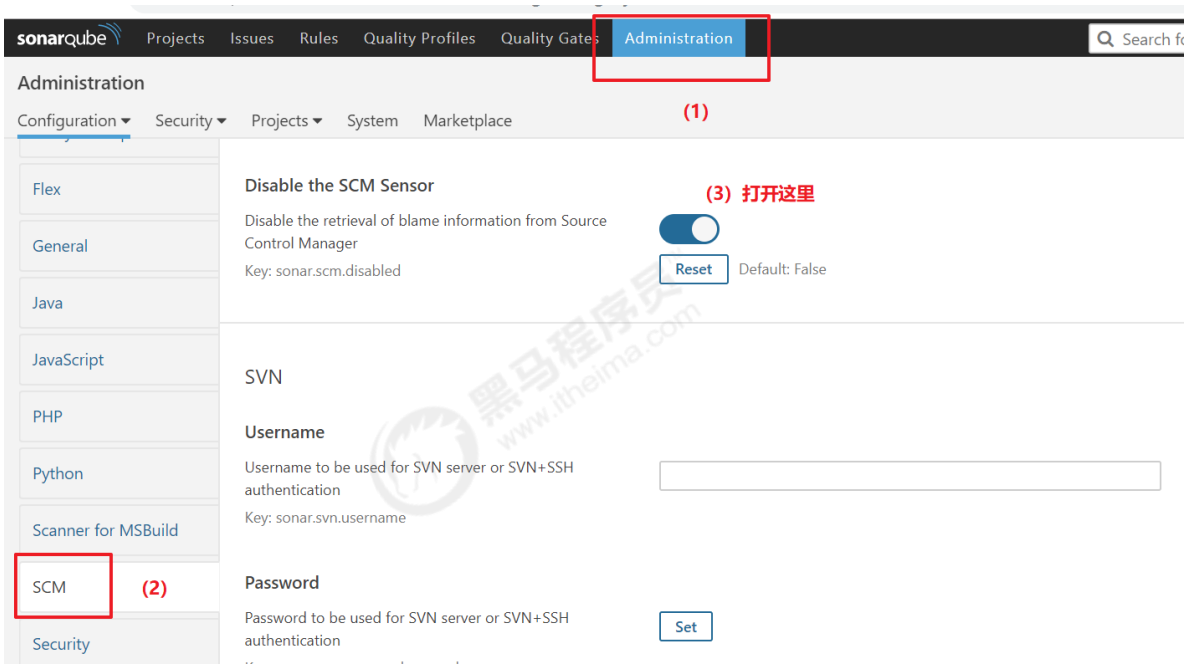
Manage Jenkins->Configure System->SonarQube servers



Manage Jenkins->Global Tool Configuration



SonaQube关闭审查结果上传到SCM功能



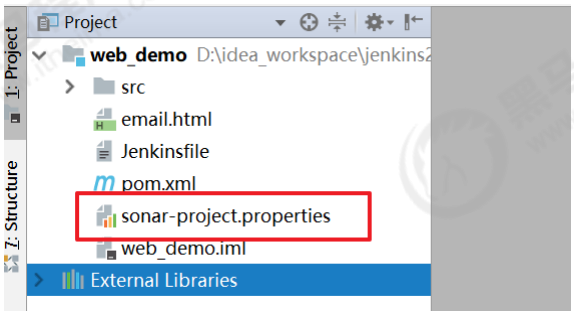
在项目添加SonaQube代码审查（非流水线项目）

添加构建步骤：

```
# must be unique in a given SonarQube instance
sonar.projectKey=web_demo
# this is the name and version displayed in the SonarQube UI. Was mandatory
prior to SonarQube 6.1.
sonar.projectName=web_demo
sonar.projectVersion=1.0

# Path is relative to the sonar-project.properties file. Replace "\" by "/" on
Windows.
# This property is optional if sonar.modules is set.
sonar.sources=.
sonar.exclusions=**/test/**,**/target/**

sonar.java.source=1.8
sonar.java.target=1.8

# Encoding of the source code. Default is default system encoding
sonar.sourceEncoding=UTF-8
```

在项目添加SonaQube代码审查（流水线项目）

1）项目根目录下，创建sonar-project.properties文件

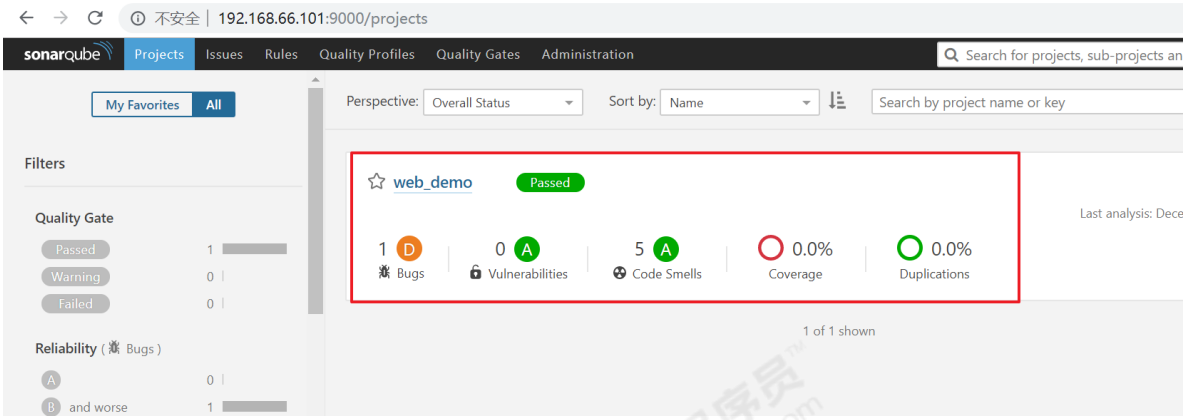



```
# must be unique in a given SonarQube instance
sonar.projectKey=web_demo
# this is the name and version displayed in the SonarQube UI. Was mandatory
prior to SonarQube 6.1.
sonar.projectName=web_demo
sonar.projectVersion=1.0

# Path is relative to the sonar-project.properties file. Replace "\" by "/" on
Windows.
# This property is optional if sonar.modules is set.
sonar.sources=.
sonar.exclusions=**/test/**,**/target/**

sonar.java.source=1.8
sonar.java.target=1.8

# Encoding of the source code. Default is default system encoding
sonar.sourceEncoding=UTF-8
```

2）修改Jenkinsfile，加入SonarQube代码审查阶段

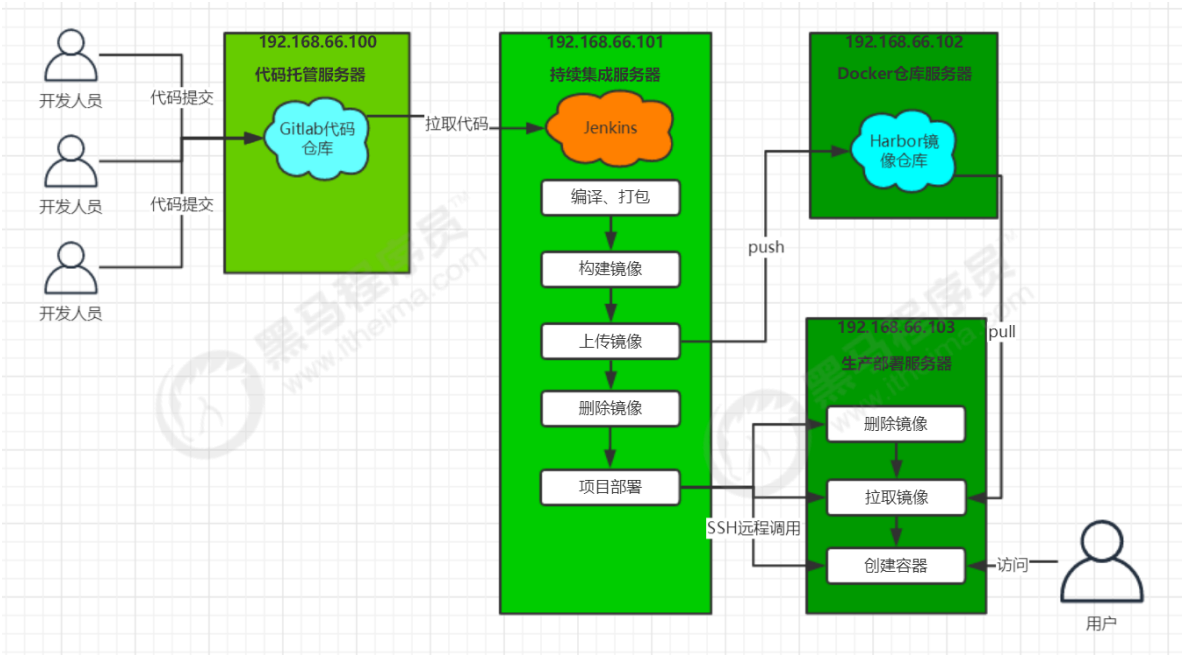
```
pipeline {
   agent any


   stages {
      stage('拉取代码') {
         steps {
            checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '68f2087f-a034-4d39-a9ff-1f776dd3dfa8', url:
'git@192.168.66.100:itheima_group/web_demo.git']]])
         }
      }
      stage('编译构建') {
         steps {
            sh label: '', script: 'mvn clean package'
         }
      }
      stage('SonarQube代码审查') {
           steps{
              script {
                   scannerHome = tool 'sonarqube-scanner'
              }
              withSonarQubeEnv('sonarqube6.7.4') {
                  sh "${scannerHome}/bin/sonar-scanner"
              }
           }
      }
      stage('项目部署') {
         steps {
            deploy adapters: [tomcat8(credentialsId: 'afc43e5e-4a4e-4de6-984f-
b1d5a254e434', path: '', url: 'http://192.168.66.102:8080')], contextPath: null,
war: 'target/*.war'
         }
      }
   }
   post {
      always {
         emailext(
            subject: '构建通知：${PROJECT_NAME} - Build # ${BUILD_NUMBER} -
${BUILD_STATUS}!',
            body: '${FILE,path="email.html"}',
            to: '1014671449@qq.com'
         )
      }
   }
}
```

3) 到SonarQube的UI界面查看审查结果



4、Jenkins+Docker+SpringCloud微服务持续集成(上)

Jenkins+Docker+SpringCloud持续集成流程说明



大致流程说明：

1）开发人员每天把代码提交到Gitlab代码仓库

2）Jenkins从Gitlab中拉取项目源码，编译并打成jar包，然后构建成Docker镜像，将镜像上传到Harbor私有仓库。

3）Jenkins发送SSH远程命令，让生产部署服务器到Harbor私有仓库拉取镜像到本地，然后创建容器。

4）最后，用户可以访问到容器

服务列表(红色的软件为需要安装的软件，黑色代表已经安装)

服务器名称	IP地址	安装的软件
代码托管服务器	192.168.66.100	Gitlab
持续集成服务器	192.168.66.101	Jenkins，Maven，Docker18.06.1-ce
Docker仓库服务器	192.168.66.102	Docker18.06.1-ce，Harbor1.9.2

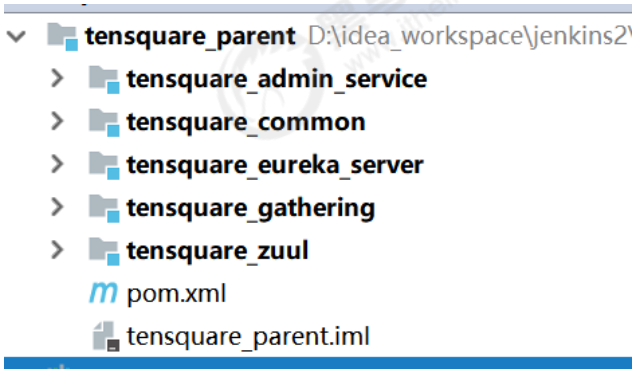
服务器名称	IP地址	安装的软件
生产部署服务器	192.168.66.103	Docker18.06.1-ce

SpringCloud微服务源码概述

项目架构：前后端分离

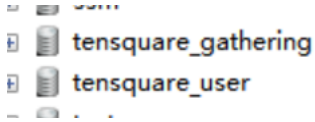
后端技术栈：SpringBoot+SpringCloud+SpringDataJpa（Spring全家桶）

微服务项目结构：



- tensquare_parent：父工程，存放基础配置
- tensquare_common：通用工程，存放工具类
- tensquare_eureka_server：SpringCloud的Eureka注册中心
- tensquare_zuul：SpringCloud的网关服务
- tensquare_admin_service：基础权限认证中心，负责用户认证（使用JWT认证）
- tensquare_gathering：一个简单的业务模块，活动微服务相关逻辑

数据库结构：



- tensquare_user：用户认证数据库，存放用户账户数据。对应tensquare_admin_service微服务
- tensquare_gathering：活动微服务数据库。对应tensquare_gathering微服务

微服务配置分析：

- tensquare_eureka
- tensquare_zuul
- tensquare_admin_service
- tensquare_gathering

本地部署(1)-SpringCloud微服务部署

本地运行微服务

1）逐一启动微服务

2）使用postman测试功能是否可用

本地部署微服务

1）SpringBoot微服务项目打包

必须导入该插件

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

打包后在target下产生jar包

2）本地运行微服务的jar包

> java -jar xxx.jar

3）查看效果

本地部署(2)-前端静态web网站

前端技术栈：NodeJS+VueJS+ElementUI

使用Visual Studio Code打开源码

1）本地运行

> npm run dev

2）打包静态web网站

> npm run build

打包后，产生dist目录的静态文件

3）部署到nginx服务器

把dist目录的静态文件拷贝到nginx的html目录，启动nginx

4）启动nginx，并访问

http://localhost:82

环境准备(1)-Docker快速入门

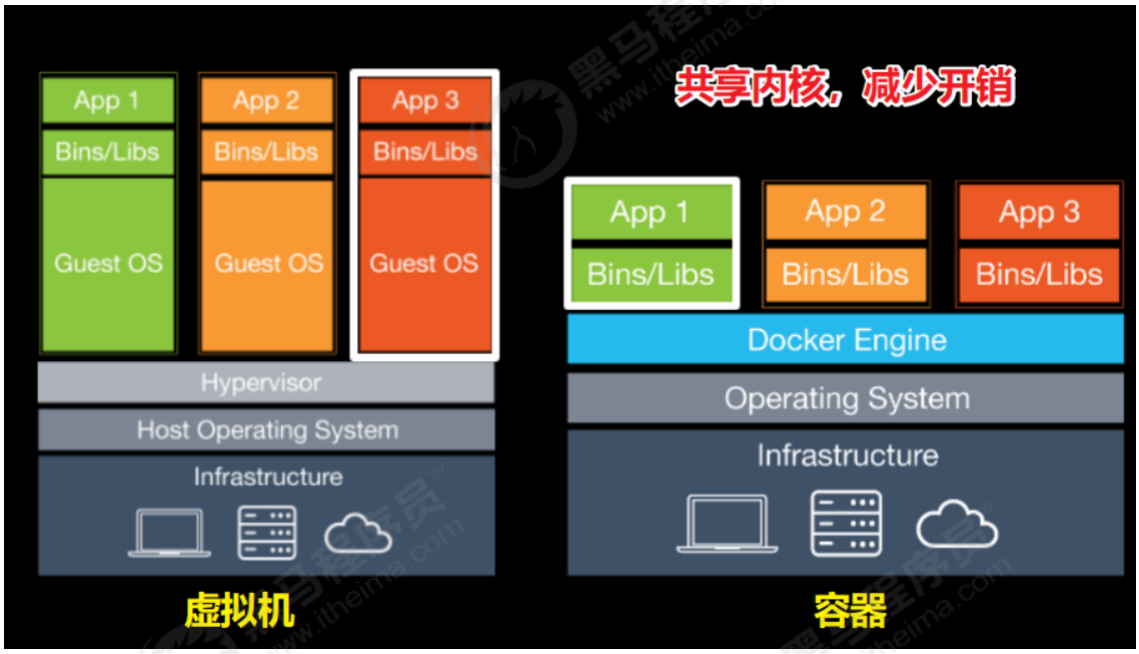
Docker简介

Docker 是一个开源的应用容器引擎，基于 Go 语言 并遵从 Apache2.0 协议开源。

Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）,更重要的是容器性能开销极低。

Docker容器技术 vs 传统虚拟机技术



	虚拟机	容器
占用磁盘空间	非常大，GB级	小，MB甚至KB级
启动速度	慢，分钟级	快，秒级
运行形态	运行于Hypervisor上	直接运行在宿主机内核上
并发性	一台宿主机上十几个，最多几十个	上百个，甚至数百上千个
性能	逊于宿主机	接近宿主机本地进程
资源利用率	低	高

简单一句话总结：**Docker技术就是让我们更加高效轻松地将任何应用在Linux服务器部署和使用。**

Docker安装

1）卸载旧版本

> yum list installed | grep docker 列出当前所有docker的包
>
> yum -y remove docker的包名称 卸载docker包

> rm -rf /var/lib/docker 删除docker的所有镜像和容器

2）安装必要的软件包

> sudo yum install -y yum-utils \ device-mapper-persistent-data \ lvm2

3）设置下载的镜像仓库

> sudo yum-config-manager \ --add-repo \ https://download.docker.com/linux/centos/docker-ce.repo

4）列出需要安装的版本列表

> yum list docker-ce --showduplicates | sort -r

```
docker-ce.x86_64  3:18.09.1-3.el7                     docker-ce-stable
docker-ce.x86_64  3:18.09.0-3.el7                     docker-ce-stable
docker-ce.x86_64  18.06.1.ce-3.el7                    docker-ce-stable
docker-ce.x86_64  18.06.0.ce-3.el7                    docker-ce-stable
......
```

5）安装指定版本（这里使用18.0.1版本）

> sudo yum install docker-ce-18.06.1.ce

6）查看版本

> docker -v

7）启动Docker

> sudo systemctl start docker 启动
>
> sudo systemctl enable docker 设置开机启动

8）添加阿里云镜像下载地址

> vi /etc/docker/daemon.json

内容如下：

```
{
"registry-mirrors": ["https://zydiol88.mirror.aliyuncs.com"]
}
```

9）重启Docker

> sudo systemctl restart docker

Docker基本命令快速入门

1）镜像命令

镜像：相当于应用的安装包，在Docker部署的任何应用都需要先构建成为镜像

> docker search 镜像名称 搜索镜像
>
> docker pull 镜像名称 拉取镜像
>
> docker images 查看本地所有镜像

> docker rmi -f 镜像名称 删除镜像

> docker pull openjdk:8-jdk-alpine

2）容器命令

容器：容器是由镜像创建而来。容器是Docker运行应用的载体，每个应用都分别运行在Docker的每个容器中。

> docker run -i 镜像名称:标签 运行容器（默认是前台运行）
>
> docker ps 查看运行的容器
>
> docker ps -a 查询所有容器

常用的参数：

-i：运行容器

-d：后台守方式运行（守护式）

--name：给容器添加名称

-p：公开容器端口给当前宿主机

-v：挂载目录

> docker exec -it 容器ID/容器名称 /bin/bash 进入容器内部
>
> docker start/stop/restart 容器名称/ID 启动/停止/重启容器
>
> docker rm -f 容器名称/ID 删除容器

环境准备(2)-Dockerfile镜像脚本快速入门

Dockerfile简介

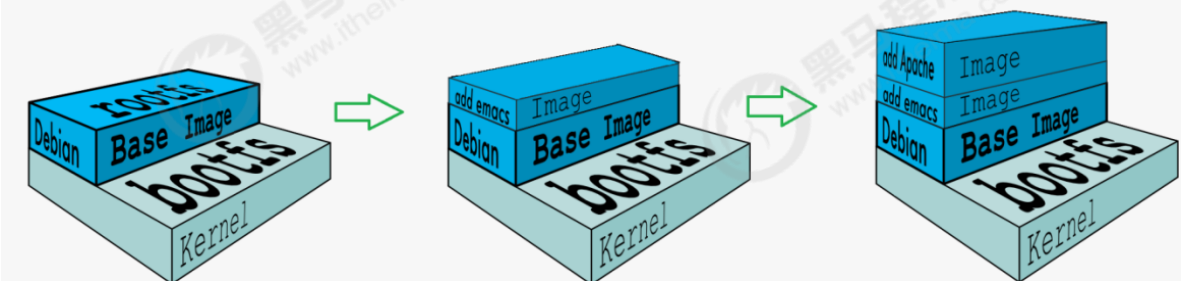
Dockerfile其实就是我们用来构建Docker镜像的源码，当然这不是所谓的编程源码，而是一些命令的组合，只要理解它的逻辑和语法格式，就可以编写Dockerfile了。

简单点说，Dockerfile的作用：它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	
MAINTAINER user_name	声明镜像的作者
ENV key value	设置环境变量 (可以写多条)
RUN command	编译镜像时运行的脚本(可以写多条)
CMD	设置容器的启动命令
ENTRYPOINT	设置容器的入口程序
ADD source_dir/file dest_dir/file	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 source_dir/file dest_dir/file	和ADD相似，但是如果有压缩文件并不能解压
WORKDIR path_dir	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

镜像构建示意图：



可以看到，新镜像是从基础镜像一层一层叠加生成的。每安装一个软件，就在现有镜像的基础上增加一层

- RUN、CMD、ENTRYPOINT的区别？

RUN：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， 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。

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 。

以下是mysql官方镜像的Dockerfile示例：

```
FROM oraclelinux:7-slim

ARG MYSQL_SERVER_PACKAGE=mysql-community-server-minimal-5.7.28
ARG MYSQL_SHELL_PACKAGE=mysql-shell-8.0.18
```

```
# Install server
RUN yum install -y https://repo.mysql.com/mysql-community-minimal-release-
el7.rpm \
      https://repo.mysql.com/mysql-community-release-el7.rpm \
  && yum-config-manager --enable mysql57-server-minimal \
  && yum install -y \
      $MYSQL_SERVER_PACKAGE \
      $MYSQL_SHELL_PACKAGE \
      libpwquality \
  && yum clean all \
  && mkdir /docker-entrypoint-initdb.d

VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /entrypoint.sh
COPY healthcheck.sh /healthcheck.sh
ENTRYPOINT ["/entrypoint.sh"]
HEALTHCHECK CMD /healthcheck.sh
EXPOSE 3306 33060
CMD ["mysqld"]
```

使用Dockerfile制作微服务镜像

我们利用Dockerfile制作一个Eureka注册中心的镜像

1）上传Eureka的微服务jar包到linux

2）编写Dockerfile

```
FROM openjdk:8-jdk-alpine
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
EXPOSE 10086
ENTRYPOINT ["java","-jar","/app.jar"]
```

3）构建镜像

> docker build --build-arg JAR_FILE=tensquare_eureka_server-1.0-SNAPSHOT.jar -t eureka:v1 .

4）查看镜像是否创建成功

> docker images

5）创建容器

> docker run -i --name=eureka -p 10086:10086 eureka:v1

6）访问容器

http://192.168.66.101:10086

环境准备(3)-Harbor镜像仓库安装及使用

Harbor简介

Harbor（港口，港湾）是一个用于存储和分发Docker镜像的企业级Registry服务器。

除了Harbor这个私有镜像仓库之外，还有Docker官方提供的Registry。相对Registry，Harbor具有很多优势：

1. 提供分层传输机制，优化网络传输 Docker镜像是是分层的，而如果每次传输都使用全量文件(所以用FTP的方式并不适合)，显然不经济。必须提供识别分层传输的机制，以层的UUID为标识，确定传输的对象。
2. 提供WEB界面，优化用户体验 只用镜像的名字来进行上传下载显然很不方便，需要有一个用户界面可以支持登陆、搜索功能，包括区分公有、私有镜像。
3. 支持水平扩展集群 当有用户对镜像的上传下载操作集中在某服务器，需要对相应的访问压力作分解。
4. 良好的安全机制 企业中的开发团队有很多不同的职位，对于不同的职位人员，分配不同的权限，具有更好的安全性。

Harbor安装

Harbor需要安装在192.168.66.102上面

1）先安装Docker并启动Docker（已完成）

参考之前的安装过程

2）先安装docker-compose

```
sudo curl -L https://github.com/docker/compose/releases/download/1.21.2/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

3）给docker-compose添加执行权限

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

4）查看docker-compose是否安装成功

> docker-compose -version

5）下载Harbor的压缩包（本课程版本为：v1.9.2）

https://github.com/goharbor/harbor/releases

6）上传压缩包到linux，并解压

> tar -xzf harbor-offline-installer-v1.9.2.tgz
>
> mkdir /opt/harbor
>
> mv harbor/* /opt/harbor
>
> cd /opt/harbor

7）修改Harbor的配置

> vi harbor.yml

修改hostname和port

> hostname: 192.168.66.102
>
> port: 85

8）安装Harbor

> ./prepare
>
> ./install.sh

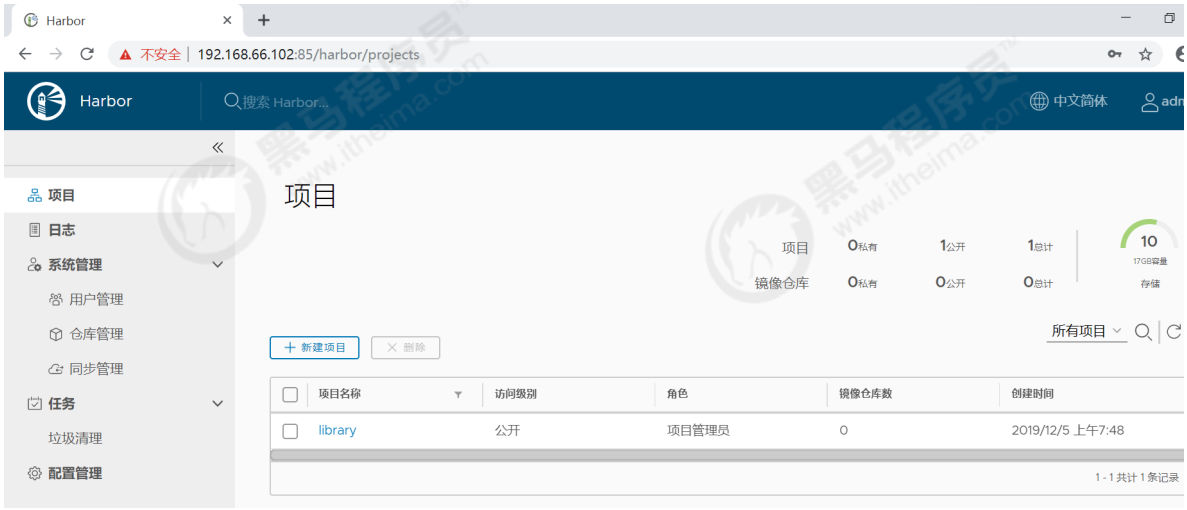
9）启动Harbor

> docker-compose up -d 启动
>
> docker-compose stop 停止
>
> docker-compose restart 重新启动

10）访问Harbor

http://192.168.66.102:85

默认账户密码：admin/Harbor12345



在Harbor创建用户和项目

1）创建项目

Harbor的项目分为公开和私有的：

公开项目：所有用户都可以访问，通常存放公共的镜像，默认有一个library公开项目。

私有项目：只有授权用户才可以访问，通常存放项目本身的镜像。

我们可以为微服务项目创建一个新的项目：


项目
项目 0私有
镜像仓库 0私有
新建项目
删除
项目名称
访问级别
角色
镜像仓库数
library
公开
项目管理员
0


新建项目
项目名称 *
tensquare
访问级别
公开
存储数量 *
-1
存储容量 *
-1
GB
取消
确定

2) 创建用户


用户管理
创建用户
设置为管理员
操作
用户名
管理员
邮件

创建的用户为： itcast/Itcast123

3) 给私有项目分配用户

进入tensquare项目->成员

角色	权限说明
访客	对于指定项目拥有只读权限
开发人员	对于指定项目拥有读写权限
维护人员	对于指定项目拥有读写权限，创建 Webhooks
项目管理员	除了读写权限，同时拥有用户管理/镜像扫描等管理权限

4) 以新用户登录Harbor

把镜像上传到Harbor

1) 给镜像打上标签

> docker tag eureka:v1 192.168.66.102:85/tensquare/eureka:v1

2) 推送镜像

docker push 192.168.66.102:85/tensquare/eureka:v1

```
The push refers to repository [192.168.66.102:85/tensquare/eureka]
Get https://192.168.66.102:85/v2/: http: server gave HTTP response to HTTPS
client
```

这时会出现以上报错，是因为Docker没有把Harbor加入信任列表中

3) 把Harbor地址加入到Docker信任列表

> vi /etc/docker/daemon.json

```
{
"registry-mirrors": ["https://zydiol88.mirror.aliyuncs.com"],
"insecure-registries": ["192.168.66.102:85"]
}
```

需要重启Docker

4) 再次执行推送命令，会提示权限不足

```
denied: requested access to the resource is denied
```

需要先登录Harbor，再推送镜像

5) 登录Harbor

> docker login -u 用户名 -p 密码 192.168.66.102:85

```
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

从Harbor下载镜像

需求：在192.168.66.103服务器完成从Harbor下载镜像

1) 安装Docker，并启动Docker (已经完成)

2) 修改Docker配置

> vi /etc/docker/daemon.json

```
{
"registry-mirrors": ["https://zydiol88.mirror.aliyuncs.com"],
"insecure-registries": ["192.168.66.102:85"]
}
```

重启docker

3) 先登录，再从Harbor下载镜像

> docker login -u 用户名 -p 密码 192.168.66.102:85

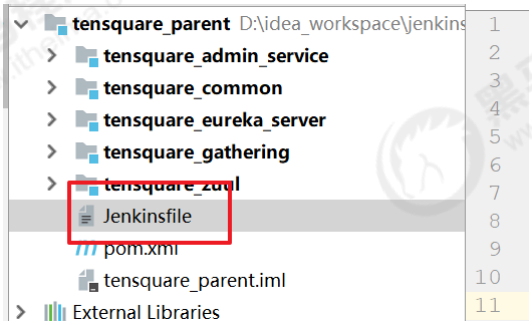
> docker pull 192.168.66.102:85/tensquare/eureka:v1

微服务持续集成(1)-项目代码上传到Gitlab

在IDEA操作即可，参考之前的步骤。包括后台微服务和前端web网站代码

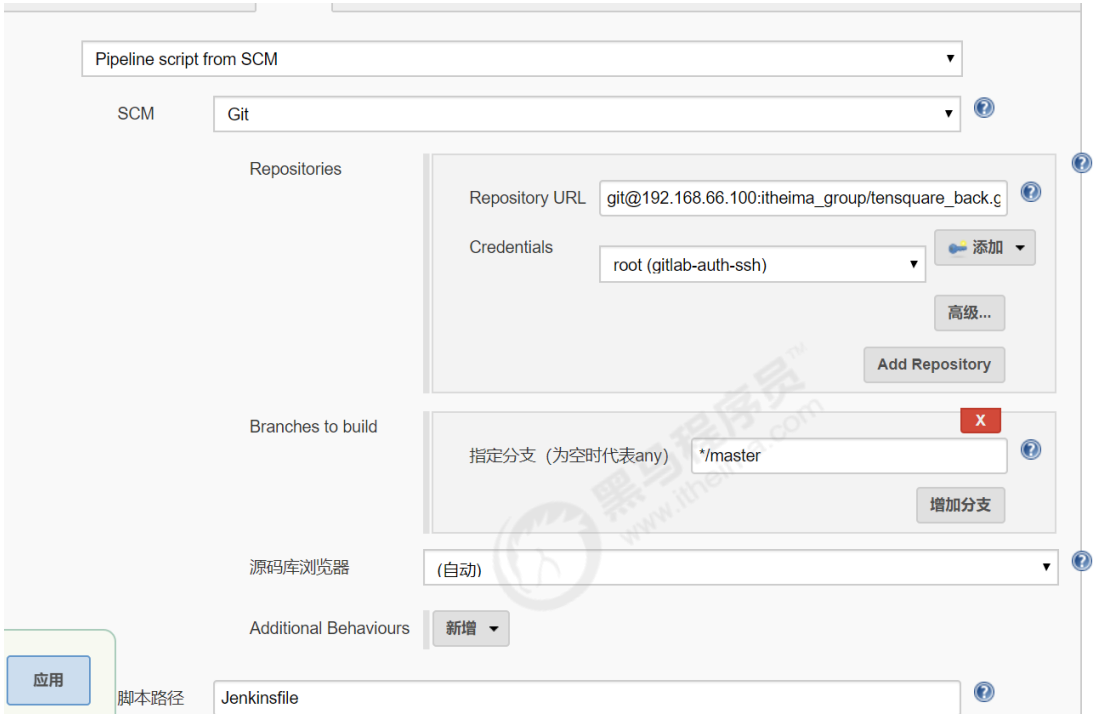
微服务持续集成(2)-从Gitlab拉取项目源码

1) 创建Jenkinsfile文件




```
//gitlab的凭证
def git_auth = "68f2087f-a034-4d39-a9ff-1f776dd3dfa8"

node {
   stage('拉取代码') {
       checkout([$class: 'GitSCM', branches: [[name: '*/${branch}']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: "${git_auth}", url:
'git@192.168.66.100:itheima_group/tensquare_back.git']]])
   }
}
```

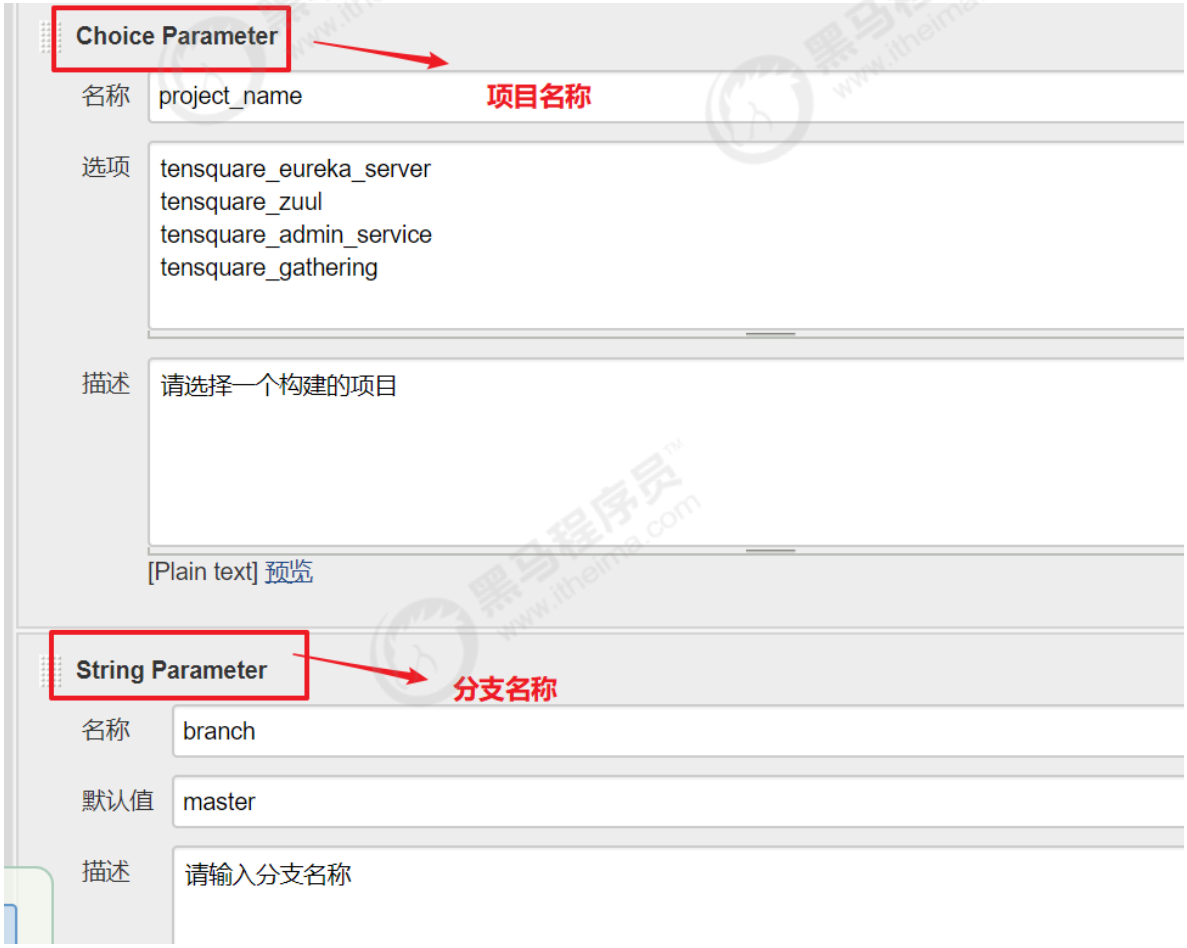
2) 拉取Jenkinsfile文件



微服务持续集成(3)-提交到SonarQube代码审查

1）创建项目，并设置参数

创建tensquare_back项目，添加两个参数



2）每个项目的根目录下添加sonar-project.properties

```
# must be unique in a given SonarQube instance
sonar.projectKey=tensquare_zuul
# this is the name and version displayed in the SonarQube UI. Was mandatory
prior to SonarQube 6.1.
sonar.projectName=tensquare_zuul
sonar.projectVersion=1.0

# Path is relative to the sonar-project.properties file. Replace "\" by "/" on
Windows.
# This property is optional if sonar.modules is set.
sonar.sources=.
sonar.exclusions=**/test/**,**/target/**
sonar.java.binaries=.

sonar.java.source=1.8
sonar.java.target=1.8
sonar.java.libraries=**/target/classes/**

# Encoding of the source code. Default is default system encoding
sonar.sourceEncoding=UTF-8
```

注意：修改sonar.projectKey和sonar.projectName

3) 修改Jenkinsfile构建脚本

```
//gitlab的凭证
def git_auth = "68f2087f-a034-4d39-a9ff-1f776dd3dfa8"
//构建版本的名称
def tag = "latest"

node {
    stage('拉取代码') {
       checkout([$class: 'GitSCM', branches: [[name: '*/${branch}']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: "${git_auth}", url:
'git@192.168.66.100:itheima_group/tensquare_back.git']]])
   }
   stage('代码审查') {
       def scannerHome = tool 'sonarqube-scanner'
       withSonarQubeEnv('sonarqube6.7.4') {
           sh """
              cd ${project_name}
              ${scannerHome}/bin/sonar-scanner
           """
       }
   }
}
```

微服务持续集成(4)-使用Dockerfile编译、生成镜像

利用dockerfile-maven-plugin插件构建Docker镜像

1) 在每个微服务项目的pom.xml加入dockerfile-maven-plugin插件

```
<plugin>
     <groupId>com.spotify</groupId>
     <artifactId>dockerfile-maven-plugin</artifactId>
     <version>1.3.6</version>
     <configuration>
          <repository>${project.artifactId}</repository>
          <buildArgs>
               <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
          </buildArgs>
     </configuration>
</plugin>
```

2) 在每个微服务项目根目录下建立Dockerfile文件

```
#FROM java:8
FROM openjdk:8-jdk-alpine
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
EXPOSE 10086
ENTRYPOINT ["java","-jar","/app.jar"]
```

注意：每个项目公开的端口不一样

3) 修改Jenkinsfile构建脚本

```
//gitlab的凭证
def git_auth = "68f2087f-a034-4d39-a9ff-1f776dd3dfa8"
//构建版本的名称
def tag = "latest"
//Harbor私服地址
def harbor_url = "192.168.66.102:85/tensquare/"

node {
   stage('拉取代码') {
       checkout([$class: 'GitSCM', branches: [[name: '*/${branch}']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: "${git_auth}", url:
'git@192.168.66.100:itheima_group/tensquare_back.git']]])
   }
   stage('代码审查') {
       def scannerHome = tool 'sonarqube-scanner'
       withSonarQubeEnv('sonarqube6.7.4') {
           sh """
              cd ${project_name}
              ${scannerHome}/bin/sonar-scanner
           """
       }
   }
   stage('编译，构建镜像') {
       //定义镜像名称
       def imageName = "${project_name}:${tag}"

       //编译，安装公共工程
```

```
        sh "mvn -f tensquare_common clean install"

        //编译，构建本地镜像
        sh "mvn -f ${project_name} clean package dockerfile:build"

    }

}
```

注意：如果出现找不到父工程依赖，需要手动把父工程的依赖上传到仓库中

微服务持续集成(5)-上传到Harbor镜像仓库

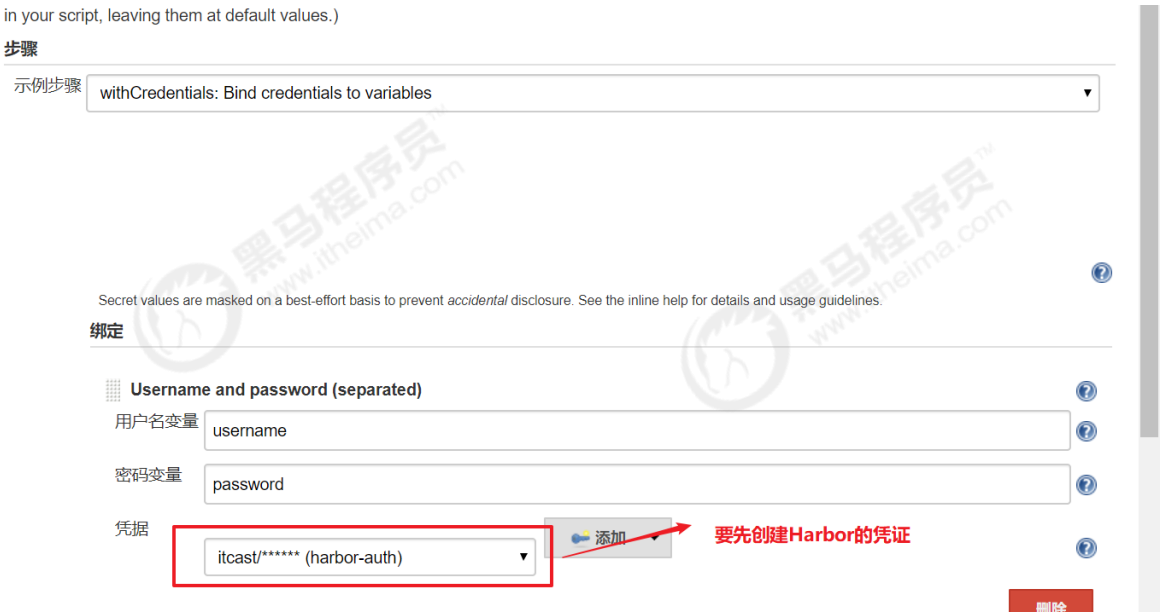
1) 修改Jenkinsfile构建脚本

```
//gitlab的凭证
def git_auth = "68f2087f-a034-4d39-a9ff-1f776dd3dfa8"
//构建版本的名称
def tag = "latest"
//Harbor私服地址
def harbor_url = "192.168.66.102:85"
//Harbor的项目名称
def harbor_project_name = "tensquare"
//Harbor的凭证
def harbor_auth = "ef499f29-f138-44dd-975e-ff1ca1d8c933"

node {
   stage('拉取代码') {
        checkout([$class: 'GitSCM', branches: [[name: '*/${branch}']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: "${git_auth}", url:
'git@192.168.66.100:itheima_group/tensquare_back.git']]])
   }
   stage('代码审查') {
        def scannerHome = tool 'sonarqube-scanner'
        withSonarQubeEnv('sonarqube6.7.4') {
            sh """
               cd ${project_name}
               ${scannerHome}/bin/sonar-scanner
            """
        }
   }
   stage('编译，构建镜像') {
        //定义镜像名称
        def imageName = "${project_name}:${tag}"

        //编译，安装公共工程
        sh "mvn -f tensquare_common clean install"

        //编译，构建本地镜像
        sh "mvn -f ${project_name} clean package dockerfile:build"

        //给镜像打标签
        sh "docker tag ${imageName}
${harbor_url}/${harbor_project_name}/${imageName}"
```

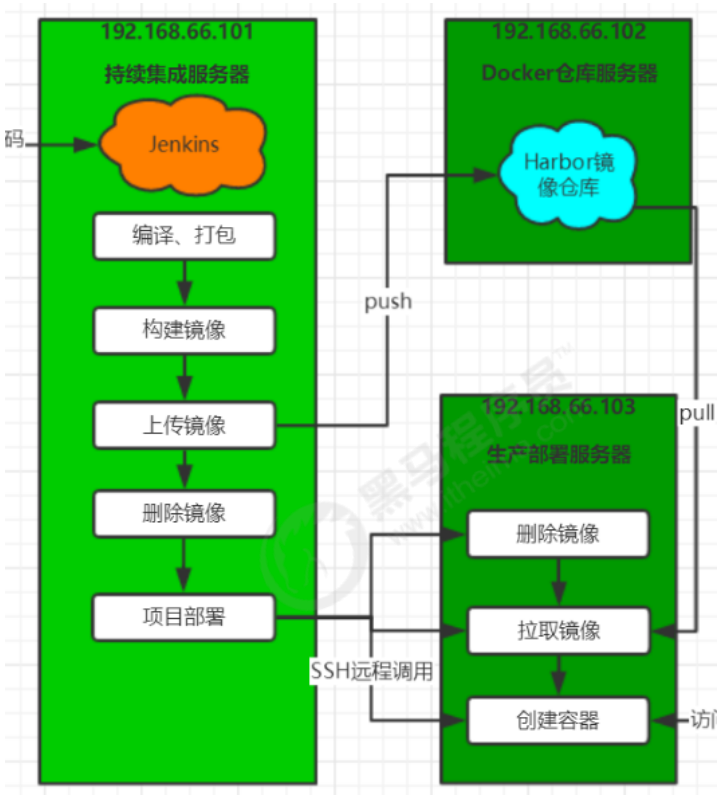
```
        //登录Harbor，并上传镜像
        withCredentials([usernamePassword(credentialsId: "${harbor_auth}",
passwordVariable: 'password', usernameVariable: 'username')]) {
            //登录
            sh "docker login -u ${username} -p ${password} ${harbor_url}"
            //上传镜像
            sh "docker push ${harbor_url}/${harbor_project_name}/${imageName}"
        }

        //删除本地镜像
        sh "docker rmi -f ${imageName}"
        sh "docker rmi -f ${harbor_url}/${harbor_project_name}/${imageName}"
    }

}
```

2）使用凭证管理Harbor私服账户和密码

先在凭证建立Harbor的凭证，在生成凭证脚本代码



微服务持续集成(6)-拉取镜像和发布应用



注意：192.168.66.103服务已经安装Docker并启动

安装 Publish Over SSH 插件

安装以下插件，可以实现远程发送Shell命令

配置远程部署服务器

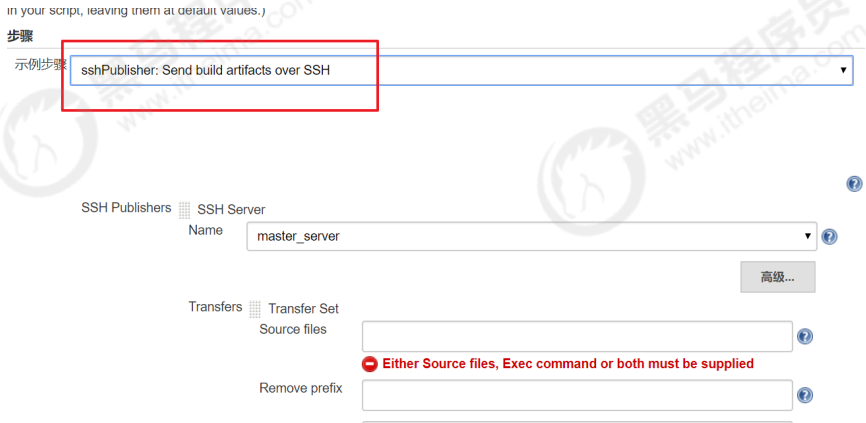
1）拷贝公钥到远程服务器

```
ssh-copy-id 192.168.66.103
```

2）系统配置->添加远程服务器

修改Jenkinsfile构建脚本

生成远程调用模板代码



添加一个port参数


```
//gitlab的凭证
def git_auth = "68f2087f-a034-4d39-a9ff-1f776dd3dfa8"
//构建版本的名称
def tag = "latest"
```

```
//Harbor私服地址
def harbor_url = "192.168.66.102:85"
//Harbor的项目名称
def harbor_project_name = "tensquare"
//Harbor的凭证
def harbor_auth = "ef499f29-f138-44dd-975e-ff1ca1d8c933"

node {
   stage('拉取代码') {
       checkout([$class: 'GitSCM', branches: [[name: '*/${branch}']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: "${git_auth}", url:
'git@192.168.66.100:itheima_group/tensquare_back.git']]])
   }
   stage('代码审查') {
       def scannerHome = tool 'sonarqube-scanner'
       withSonarQubeEnv('sonarqube6.7.4') {
           sh """
              cd ${project_name}
              ${scannerHome}/bin/sonar-scanner
           """
       }
   }
   stage('编译，构建镜像，部署服务') {
       //定义镜像名称
       def imageName = "${project_name}:${tag}"

       //编译并安装公共工程
       sh "mvn -f tensquare_common clean install"

       //编译，构建本地镜像
       sh "mvn -f ${project_name} clean package dockerfile:build"

       //给镜像打标签
       sh "docker tag ${imageName}
${harbor_url}/${harbor_project_name}/${imageName}"

       //登录Harbor，并上传镜像
       withCredentials([usernamePassword(credentialsId: "${harbor_auth}",
passwordVariable: 'password', usernameVariable: 'username')]) {
           //登录
           sh "docker login -u ${username} -p ${password} ${harbor_url}"
           //上传镜像
           sh "docker push ${harbor_url}/${harbor_project_name}/${imageName}"
       }

       //删除本地镜像
       sh "docker rmi -f ${imageName}"
       sh "docker rmi -f ${harbor_url}/${harbor_project_name}/${imageName}"


       //=====以下为远程调用进行项目部署========
```

```
        sshPublisher(publishers: [sshPublisherDesc(configName: 'master_server',
transfers: [sshTransfer(cleanRemote: false, excludes: '', execCommand:
"/opt/jenkins_shell/deploy.sh $harbor_url $harbor_project_name $project_name
$tag $port", execTimeout: 120000, flatten: false, makeEmptyDirs: false,
noDefaultExcludes: false, patternSeparator: '[, ]+', remoteDirectory: '',
remoteDirectorySDF: false, removePrefix: '', sourceFiles: '')],
usePromotionTimestamp: false, useWorkspaceInPromotion: false, verbose: false)])
    }
}
```

编写deploy.sh部署脚本

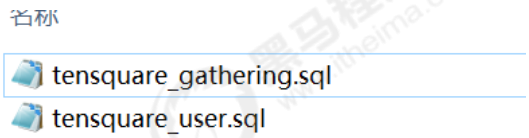
```
#! /bin/sh
#接收外部参数
harbor_url=$1
harbor_project_name=$2
project_name=$3
tag=$4
port=$5

imageName=$harbor_url/$harbor_project_name/$project_name:$tag

echo "$imageName"

#查询容器是否存在，存在则删除
containerId=`docker ps -a | grep -w ${project_name}:${tag}  | awk '{print $1}'`
if [ "$containerId" !=  "" ] ; then
    #停掉容器
    docker stop $containerId

    #删除容器
    docker rm $containerId

    echo "成功删除容器"
fi

#查询镜像是否存在，存在则删除
imageId=`docker images | grep -w $project_name  | awk '{print $3}'`

if [ "$imageId" !=  "" ] ; then

    #删除镜像
    docker rmi -f $imageId

    echo "成功删除镜像"
fi

# 登录Harbor私服
docker login -u itcast -p Itcast123 $harbor_url

# 下载镜像
docker pull $imageName

# 启动容器
docker run -di -p $port:$port $imageName
```

```
echo "容器启动成功"
```

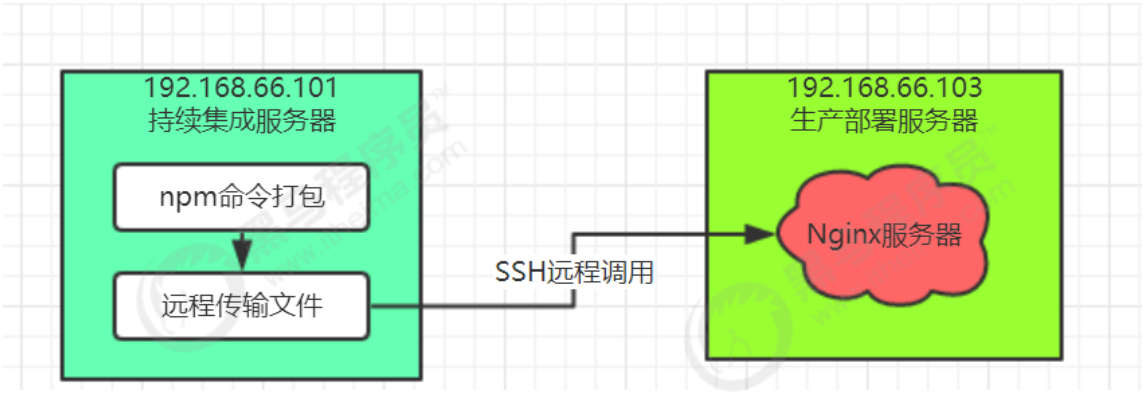
上传deploy.sh文件到/opt/jenkins_shell目录下，且文件至少有执行权限！

> chmod +x deploy.sh 添加执行权限

导入数据，测试微服务



微服务持续集成(7)-部署前端静态web网站



安装Nginx服务器

> yum install epel-release
>
> yum -y install nginx 安装

修改nginx的端口，默认80，改为9090：

> vi /etc/nginx/nginx.conf

```
server {
        listen       9090 default_server;
        listen       [::]:9090 default_server;
        server_name  _;
        root         /usr/share/nginx/html;
```

还需要关闭selinux，将SELINUX=disabled

> setenforce 0 先临时关闭
>
> vi /etc/selinux/config 编辑文件，永久关闭 SELINUX=disabled

启动Nginx

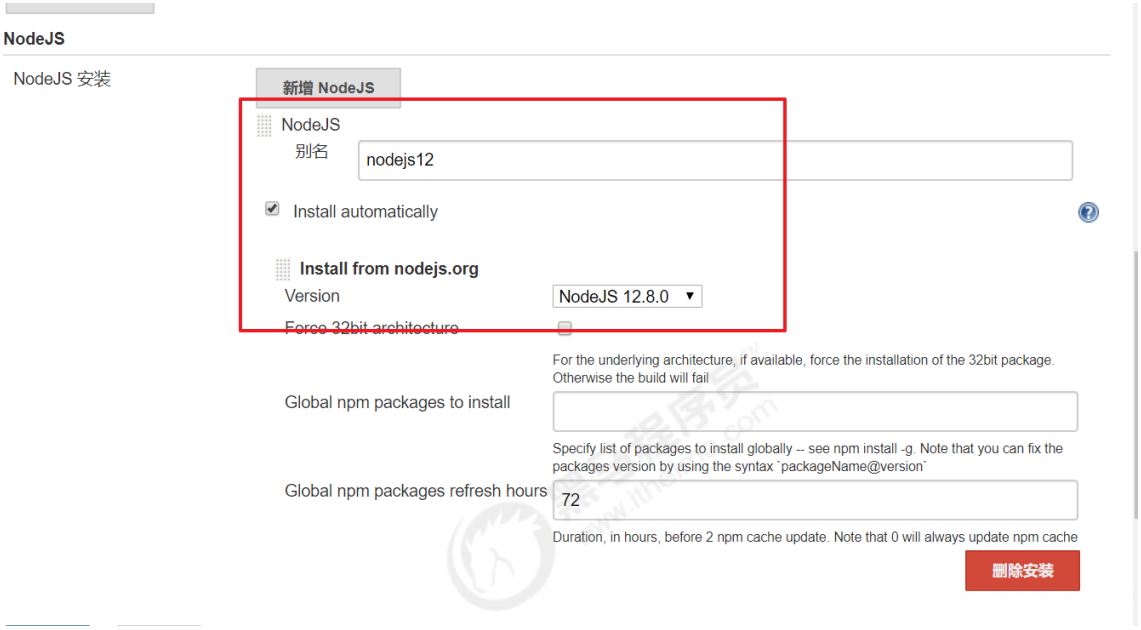
> systemctl enable nginx 设置开机启动
>
> systemctl start nginx 启动
>
> systemctl stop nginx 停止
>
> systemctl restart nginx 重启

访问：http://192.168.66.103:9090/

安装NodeJS插件

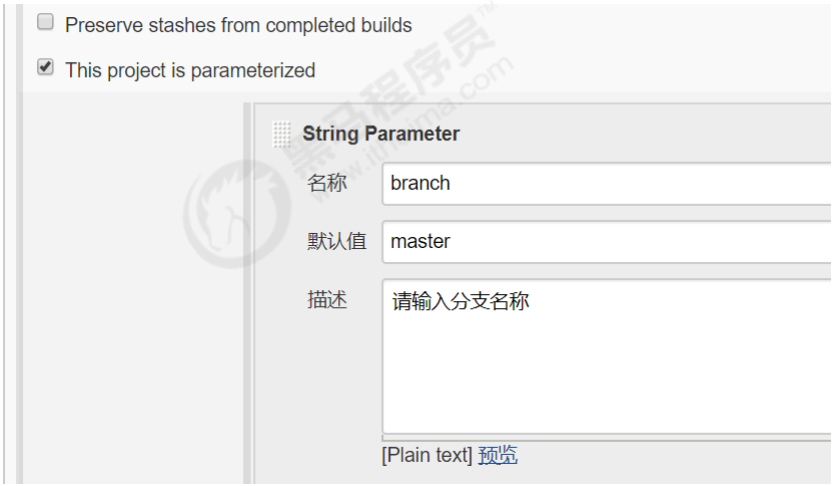
Jenkins配置Nginx服务器

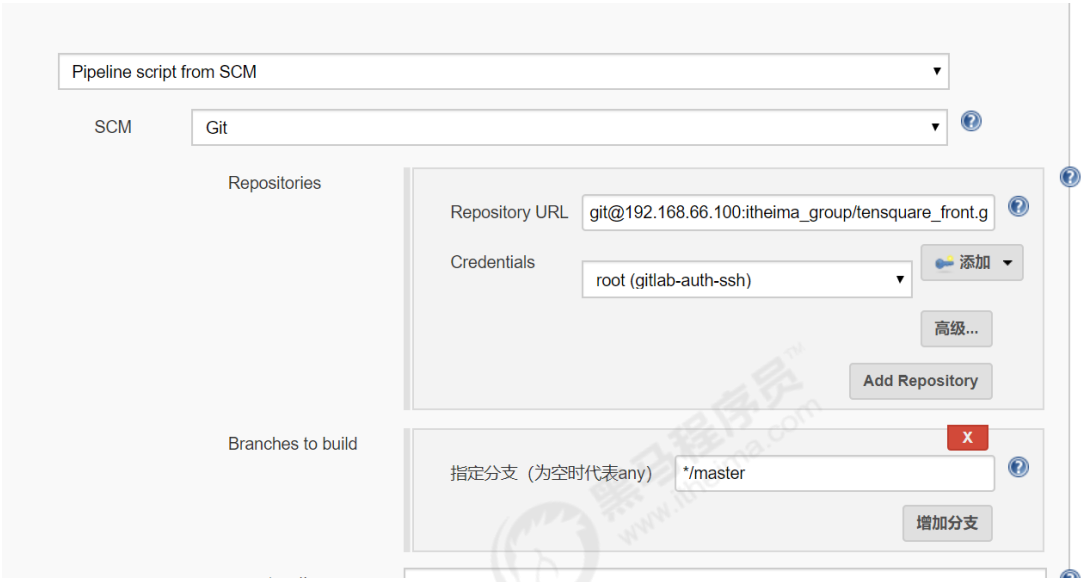
Manage Jenkins->Global Tool Configuration


NodeJS
NodeJS 安装
新增 NodeJS
NodeJS
别名
nodejs12
Install automatically
Install from nodejs.org
Version
NodeJS 12.8.0
Force 32bit architecture
For the underlying architecture, if available, force the installation of the 32bit package. Otherwise the build will fail
Global npm packages to install
Specify list of packages to install globally -- see npm install -g. Note that you can fix the packages version by using the syntax `packageName@version`
Global npm packages refresh hours
72
Duration, in hours, before 2 npm cache update. Note that 0 will always update npm cache
删除安装

创建前端流水线项目


输入一个任务名称
tensquare_front
» 必填项
Freestyle project
This is the central feature of Jenkins. Jenkins will
something other than software build.
构建一个maven项目
构建一个maven项目.Jenkins利用你的POM文件,
流水线
精心地组织一个可以长期运行在多个节点上的任务
型。
构建一个多配置项目


Preserve stashes from completed builds
This project is parameterized
String Parameter
名称
branch
默认值
master
描述
请输入分支名称
[Plain text] 预览



建立Jenkinsfile构建脚本

```
//gitlab的凭证
def git_auth = "68f2087f-a034-4d39-a9ff-1f776dd3dfa8"

node {
   stage('拉取代码') {
       checkout([$class: 'GitSCM', branches: [[name: '*/${branch}']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: "${git_auth}", url:
'git@192.168.66.100:itheima_group/tensquare_front.git']]])
   }
   stage('打包，部署网站') {
       //使用NodeJS的npm进行打包
       nodejs('nodejs12'){
            sh '''
                npm install
                npm run build
            '''
       }

       //=====以下为远程调用进行项目部署=========
       sshPublisher(publishers: [sshPublisherDesc(configName: 'master_server',
transfers: [sshTransfer(cleanRemote: false, excludes: '', execCommand: '',
execTimeout: 120000, flatten: false, makeEmptyDirs: false, noDefaultExcludes:
false, patternSeparator: '[, ]+', remoteDirectory: '/usr/share/nginx/html',
remoteDirectorySDF: false, removePrefix: 'dist', sourceFiles: 'dist/**')],
usePromotionTimestamp: false, useWorkspaceInPromotion: false, verbose: false)])
   }
}
```

完成后，访问：http://192.168.66.103:9090 进行测试。

5、Jenkins+Docker+SpringCloud微服务持续集成(下)

Jenkins+Docker+SpringCloud部署方案优化

上面部署方案存在的问题：

1）一次只能选择一个微服务部署

2）只有一台生产者部署服务器

3）每个微服务只有一个实例，容错率低

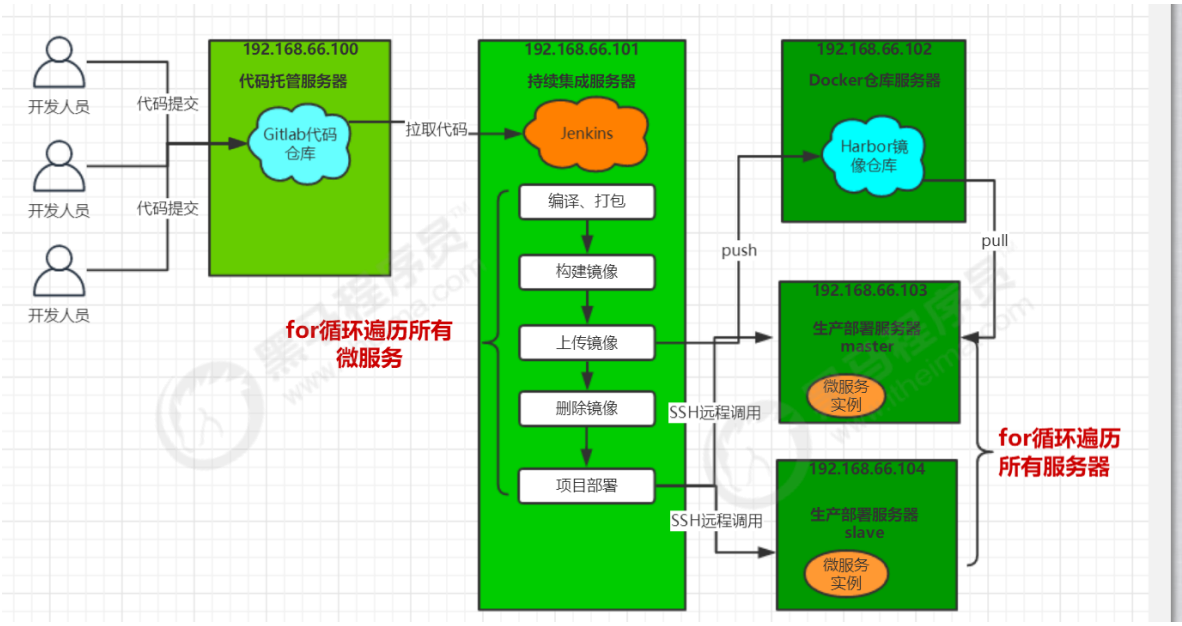
优化方案：

1）在一个Jenkins工程中可以选择多个微服务同时发布

2）在一个Jenkins工程中可以选择多台生产服务器同时部署

3）每个微服务都是以**集群高可用**形式部署

Jenkins+Docker+SpringCloud集群部署流程说明



修改所有微服务配置

注册中心配置(*)

```
# 集群版
spring:
  application:
    name: EUREKA-HA

---
server:
  port: 10086
spring:
  # 指定profile=eureka-server1
  profiles: eureka-server1
eureka:
  instance:
    # 指定当profile=eureka-server1时，主机名是eureka-server1
    hostname: 192.168.66.103
  client:
```

```
    service-url:
     # 将自己注册到eureka-server1、eureka-server2这个Eureka上面去
      defaultZone:
http://192.168.66.103:10086/eureka/,http://192.168.66.104:10086/eureka/

---
server:
  port: 10086
spring:
  profiles: eureka-server2
eureka:
  instance:
    hostname: 192.168.66.104
  client:
    service-url:
      defaultZone:
http://192.168.66.103:10086/eureka/,http://192.168.66.104:10086/eureka/
```

在启动微服务的时候，加入参数: **spring.profiles.active** 来读取对应的配置

其他微服务配置

除了Eureka注册中心以外，其他微服务配置都需要加入所有Eureka服务

```
# Eureka配置
eureka:
  client:
    service-url:
      defaultZone:
http://192.168.66.103:10086/eureka,http://192.168.66.104:10086/eureka # Eureka访问地址
  instance:
    prefer-ip-address: true
```

把代码提交到Gitlab中

设计Jenkins集群项目的构建参数

1）安装Extended Choice Parameter插件

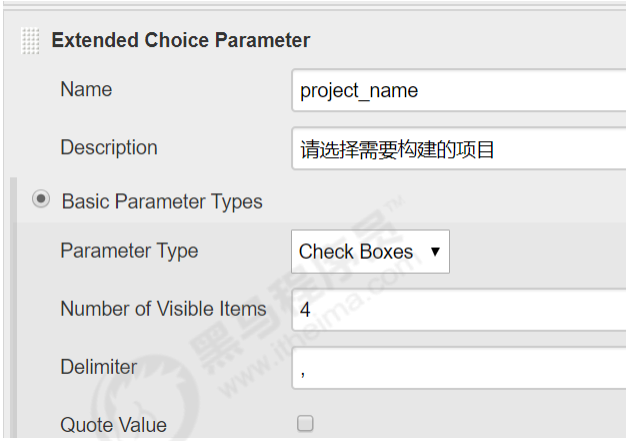
支持多选框

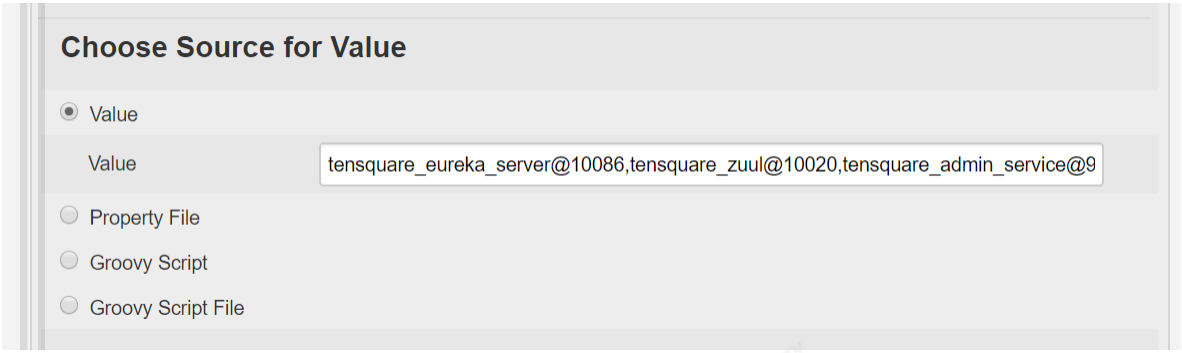
2）创建流水线项目

3) 添加参数

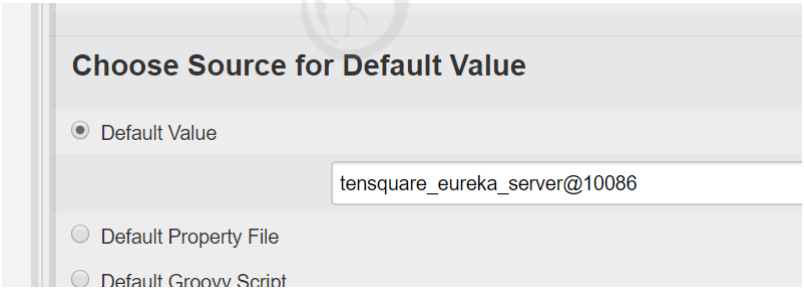
字符串参数：分支名称

多选框：项目名称






```
tensquare_eureka_server@10086,tensquare_zuul@10020,tensquare_admin_service@9001,
tensquare_gathering@9002
```



最后效果：

完成微服务构建镜像，上传私服

```
//gitlab的凭证
def git_auth = "68f2087f-a034-4d39-a9ff-1f776dd3dfa8"
```

```
//构建版本的名称
def tag = "latest"
//Harbor私服地址
def harbor_url = "192.168.66.102:85"
//Harbor的项目名称
def harbor_project_name = "tensquare"
//Harbor的凭证
def harbor_auth = "ef499f29-f138-44dd-975e-ff1ca1d8c933"

node {
   //把选择的项目信息转为数组
   def selectedProjects = "${project_name}".split(',')

   stage('拉取代码') {
       checkout([$class: 'GitSCM', branches: [[name: '*/${branch}']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '${git_auth}', url:
'git@192.168.66.100:itheima_group/tensquare_back_cluster.git']]])
   }
   stage('代码审查') {
       def scannerHome = tool 'sonarqube-scanner'
       withSonarQubeEnv('sonarqube6.7.4') {
           for(int i=0;i<selectedProjects.size();i++){
              //取出每个项目的名称和端口
              def currentProject = selectedProjects[i];
              //项目名称
              def currentProjectName = currentProject.split('@')[0]
              //项目启动端口
              def currentProjectPort = currentProject.split('@')[1]

              sh """
                    cd ${currentProjectName}
                    ${scannerHome}/bin/sonar-scanner
              """

              echo "${currentProjectName}完成代码审查"
           }
       }

   }
   stage('编译，构建镜像，部署服务') {

         //编译并安装公共工程
        sh "mvn -f tensquare_common clean install"

        for(int i=0;i<selectedProjects.size();i++){
              //取出每个项目的名称和端口
              def currentProject = selectedProjects[i];
              //项目名称
              def currentProjectName = currentProject.split('@')[0]
              //项目启动端口
              def currentProjectPort = currentProject.split('@')[1]


               //定义镜像名称
               def imageName = "${currentProjectName}:${tag}"

               //编译，构建本地镜像
```

```
                    sh "mvn -f ${currentProjectName} clean package
dockerfile:build"

                    //给镜像打标签
                    sh "docker tag ${imageName}
${harbor_url}/${harbor_project_name}/${imageName}"

                    //登录Harbor，并上传镜像
                    withCredentials([usernamePassword(credentialsId:
"${harbor_auth}", passwordVariable: 'password', usernameVariable: 'username')])
{
                        //登录
                        sh "docker login -u ${username} -p ${password}
${harbor_url}"
                        //上传镜像
                        sh "docker push
${harbor_url}/${harbor_project_name}/${imageName}"
                    }

                    //删除本地镜像
                    sh "docker rmi -f ${imageName}"
                    sh "docker rmi -f
${harbor_url}/${harbor_project_name}/${imageName}"


                    //=====以下为远程调用进行项目部署========
                    //sshPublisher(publishers: [sshPublisherDesc(configName:
'master_server', transfers: [sshTransfer(cleanRemote: false, excludes: '',
execCommand: "/opt/jenkins_shell/deployCluster.sh $harbor_url
$harbor_project_name $currentProjectName $tag $currentProjectPort", execTimeout:
120000, flatten: false, makeEmptyDirs: false, noDefaultExcludes: false,
patternSeparator: '[, ]+', remoteDirectory: '', remoteDirectorySDF: false,
removePrefix: '', sourceFiles: '')], usePromotionTimestamp: false,
useWorkspaceInPromotion: false, verbose: false)])

                    echo "${currentProjectName}完成编译，构建镜像"
            }

    }
}
```

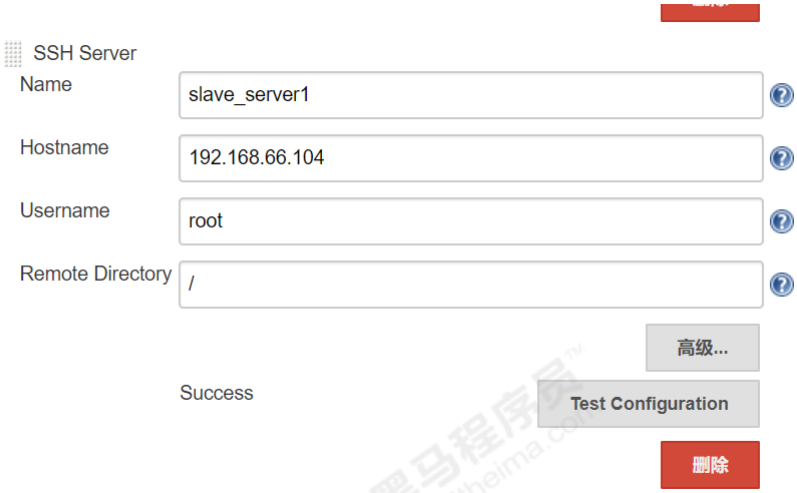
完成微服务多服务器远程发布

1）配置远程部署服务器

- 拷贝公钥到远程服务器

> ssh-copy-id 192.168.66.104

- 系统配置->添加远程服务器



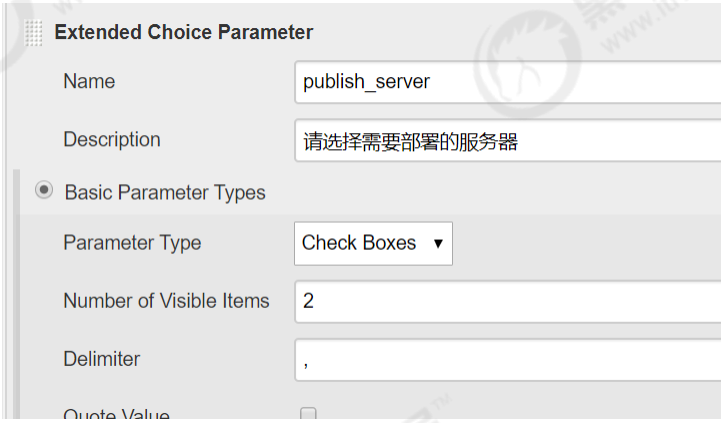
2) 修改Docker配置信任Harbor私服地址

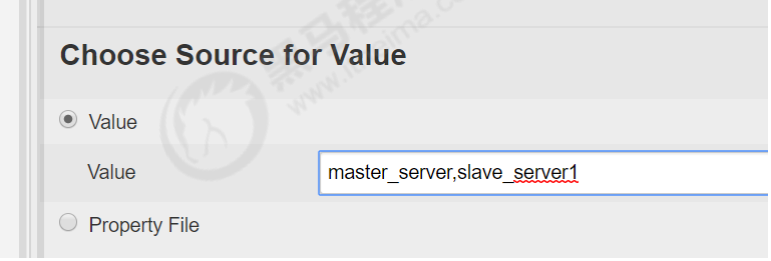
```
{
"registry-mirrors": ["https://zydiol88.mirror.aliyuncs.com"],
"insecure-registries": ["192.168.66.102:85"]
}
```

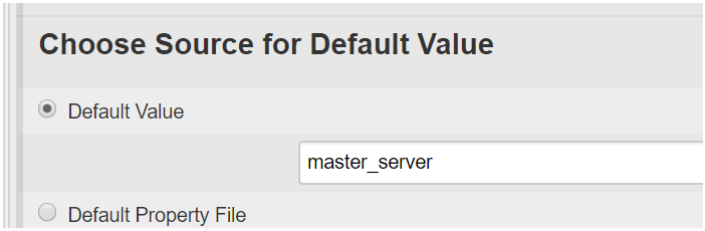
重启Docker

3) 添加参数

多选框：部署服务器







最终效果：

4) 修改Jenkinsfile构建脚本

```
//gitlab的凭证
def git_auth = "68f2087f-a034-4d39-a9ff-1f776dd3dfa8"
//构建版本的名称
def tag = "latest"
//Harbor私服地址
def harbor_url = "192.168.66.102:85"
//Harbor的项目名称
def harbor_project_name = "tensquare"
//Harbor的凭证
def harbor_auth = "ef499f29-f138-44dd-975e-ff1ca1d8c933"

node {
   //把选择的项目信息转为数组
   def selectedProjects = "${project_name}".split(',')
   //把选择的服务区信息转为数组
   def selectedServers = "${publish_server}".split(',')

   stage('拉取代码') {
       checkout([$class: 'GitSCM', branches: [[name: '*/${branch}']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '${git_auth}', url:
'git@192.168.66.100:itheima_group/tensquare_back_cluster.git']]])
   }
```

```
    stage('代码审查') {
        def scannerHome = tool 'sonarqube-scanner'
        withSonarQubeEnv('sonarqube6.7.4') {
            for(int i=0;i<selectedProjects.size();i++){
               //取出每个项目的名称和端口
               def currentProject = selectedProjects[i];
               //项目名称
               def currentProjectName = currentProject.split('@')[0]
               //项目启动端口
               def currentProjectPort = currentProject.split('@')[1]

               sh """
                     cd ${currentProjectName}
                     ${scannerHome}/bin/sonar-scanner
               """

               echo "${currentProjectName}完成代码审查"
            }
        }

    }
    stage('编译，构建镜像，部署服务') {

          //编译并安装公共工程
         sh "mvn -f tensquare_common clean install"

         for(int i=0;i<selectedProjects.size();i++){
               //取出每个项目的名称和端口
               def currentProject = selectedProjects[i];
               //项目名称
               def currentProjectName = currentProject.split('@')[0]
               //项目启动端口
               def currentProjectPort = currentProject.split('@')[1]


                //定义镜像名称
                def imageName = "${currentProjectName}:${tag}"

                //编译，构建本地镜像
                sh "mvn -f ${currentProjectName} clean package
dockerfile:build"

                //给镜像打标签
                sh "docker tag ${imageName}
${harbor_url}/${harbor_project_name}/${imageName}"

                //登录Harbor，并上传镜像
                withCredentials([usernamePassword(credentialsId:
"${harbor_auth}", passwordVariable: 'password', usernameVariable: 'username')])
{
                     //登录
                     sh "docker login -u ${username} -p ${password}
${harbor_url}"
                     //上传镜像
                     sh "docker push
${harbor_url}/${harbor_project_name}/${imageName}"
                }
```

```
                    //删除本地镜像
                    sh "docker rmi -f ${imageName}"
                    sh "docker rmi -f
${harbor_url}/${harbor_project_name}/${imageName}"


                    //=====以下为远程调用进行项目部署========
                    for(int j=0;j<selectedServers.size();j++){
                        //每个服务名称
                        def currentServer = selectedServers[j]

                        //添加微服务运行时的参数：spring.profiles.active
                        def activeProfile = "--spring.profiles.active="

                        if(currentServer=="master_server"){
                             activeProfile = activeProfile+"eureka-server1"
                        }else if(currentServer=="slave_server1"){
                             activeProfile = activeProfile+"eureka-server2"
                        }

                        sshPublisher(publishers: [sshPublisherDesc(configName:
"${currentServer}", transfers: [sshTransfer(cleanRemote: false, excludes: '',
execCommand: "/opt/jenkins_shell/deployCluster.sh $harbor_url
$harbor_project_name $currentProjectName $tag $currentProjectPort
$activeProfile", execTimeout: 120000, flatten: false, makeEmptyDirs: false,
noDefaultExcludes: false, patternSeparator: '[, ]+', remoteDirectory: '',
remoteDirectorySDF: false, removePrefix: '', sourceFiles: '')],
usePromotionTimestamp: false, useWorkspaceInPromotion: false, verbose: false)])
                    }

                    echo "${currentProjectName}完成编译，构建镜像"
                }

    }
}
```

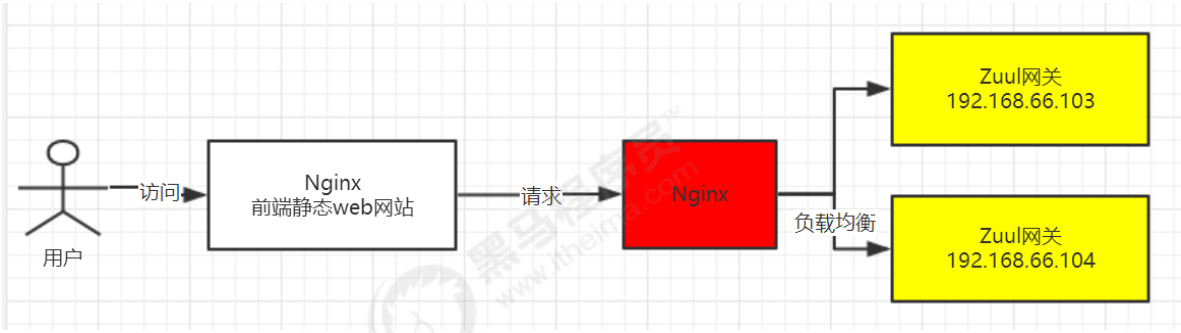
5) 编写deployCluster.sh部署脚本

```
#! /bin/sh
#接收外部参数
harbor_url=$1
harbor_project_name=$2
project_name=$3
tag=$4
port=$5
profile=$6

imageName=$harbor_url/$harbor_project_name/$project_name:$tag

echo "$imageName"

#查询容器是否存在，存在则删除
containerId=`docker ps -a | grep -w ${project_name}:${tag}  | awk '{print $1}'`
if [ "$containerId" !=  "" ] ; then
    #停掉容器
```

```
    docker stop $containerId

    #删除容器
    docker rm $containerId

    echo "成功删除容器"
fi

#查询镜像是否存在，存在则删除
imageId=`docker images | grep -w $project_name  | awk '{print $3}'`

if [ "$imageId" !=  "" ] ; then

    #删除镜像
    docker rmi -f $imageId

    echo "成功删除镜像"
fi

# 登录Harbor私服
docker login -u itcast -p Itcast123 $harbor_url

# 下载镜像
docker pull $imageName

# 启动容器
docker run -di -p $port:$port $imageName $profile

echo "容器启动成功"
```

6）集群效果

Nginx+Zuul集群实现高可用网关



1）安装Nginx（已完成）

2）修改Nginx配置

> vi /etc/nginx/nginx.conf

内容如下：

```
    upstream  zuulServer{
```

```
        server 192.168.66.103:10020 weight=1;
        server 192.168.66.104:10020 weight=1;
    }

    server {
        listen       85 default_server;
        listen       [::]:85 default_server;
        server_name  _;
        root         /usr/share/nginx/html;

        # Load configuration files for the default server block.
        include /etc/nginx/default.d/*.conf;

        location / {
            ### 指定服务器负载均衡服务器
            proxy_pass http://zuulServer/;
        }
```

3) 重启Nginx： systemctl restart nginx

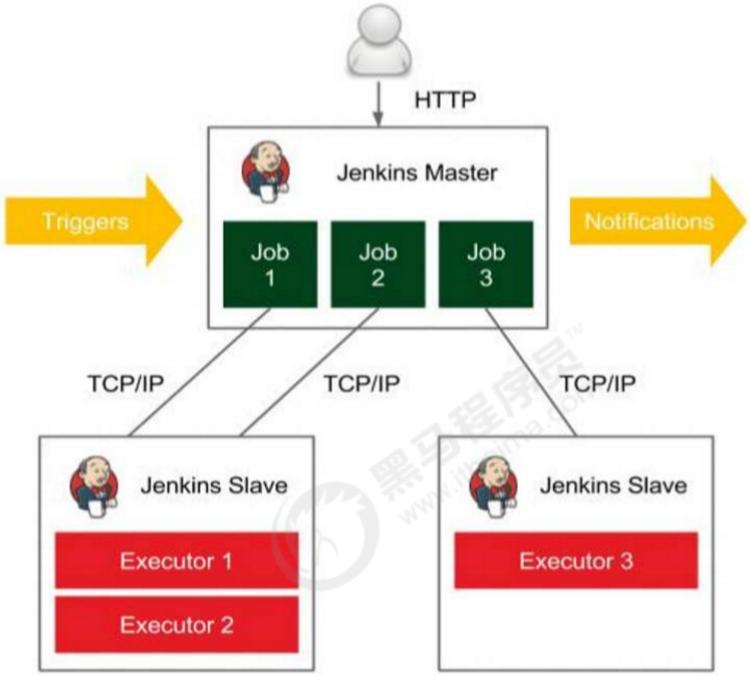
4) 修改前端Nginx的访问地址

```
'use strict'
module.exports = {
  NODE_ENV: '"production"',
  // BASE_API: '"http://192.168.207.131:7300/mock/5c0b42c85b4c750
  BASE_API: '"http://192.168.66.104:85"' // 管理员网关
}
```

6、基于Kubernetes/K8S构建Jenkins持续集成平台(上)

Jenkins的Master-Slave分布式构建

什么是Master-Slave分布式构建

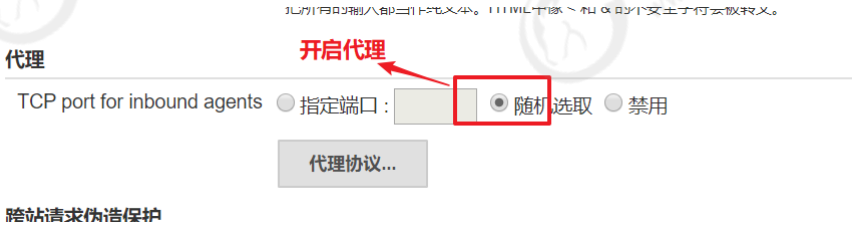


Jenkins的Master-Slave分布式构建，就是通过将构建过程分配到从属Slave节点上，从而减轻Master节点的压力，而且可以同时构建多个，有点类似负载均衡的概念。

如何实现Master-Slave分布式构建

1）开启代理程序的TCP端口

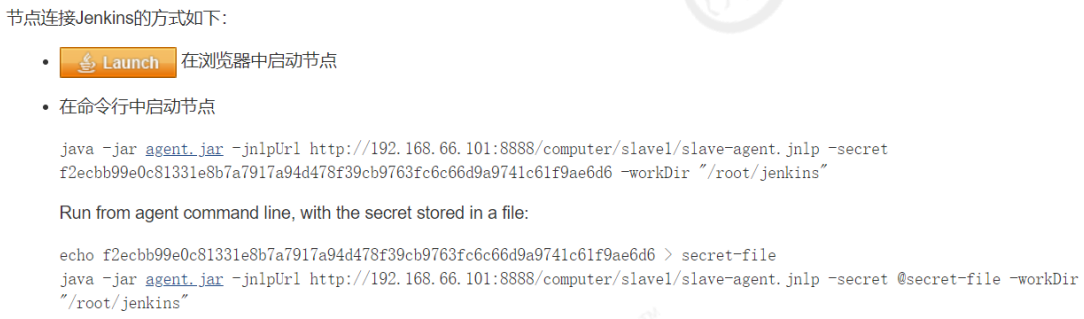
Manage Jenkins -> Configure Global Security



2）新建节点

Manage Jenkins—Manage Nodes—新建节点

有两种在Slave节点连接Master节点的方法



我们选择第二种：

2）安装和配置节点

下载agent.jar，并上传到Slave节点，然后执行页面提示的命令：

```
java -jar agent.jar -jnlpUrl http://192.168.66.101:8888/computer/slave1/slave-agent.jnlp -secret f2ecbb99e0c81331e8b7a7917a94d478f39cb9763fc6c66d9a9741c61f9ae6d6 -workDir "/root/jenkins"
```

刷新页面

3）测试节点是否可用

自由风格和Maven风格的项目：

流水线风格的项目：

```
node('slave1') {

   stage('check out') {
        checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '68f2087f-a034-4d39-a9ff-1f776dd3dfa8', url:
'git@192.168.66.100:itheima_group/tensquare_back_cluster.git']]])
   }

}
```

Kubernetes实现Master-Slave分布式构建方案

传统Jenkins的Master-Slave方案的缺陷

- Master节点发生单点故障时，整个流程都不可用了
- 每个 Slave节点的配置环境不一样，来完成不同语言的编译打包等操作，但是这些差异化的配置导致管理起来非常不方便，维护起来也是比较费劲
- 资源分配不均衡，有的 Slave节点要运行的job出现排队等待，而有的Slave节点处于空闲状态
- 资源浪费，每台 Slave节点可能是实体机或者VM，当Slave节点处于空闲状态时，也不会完全释放掉资源

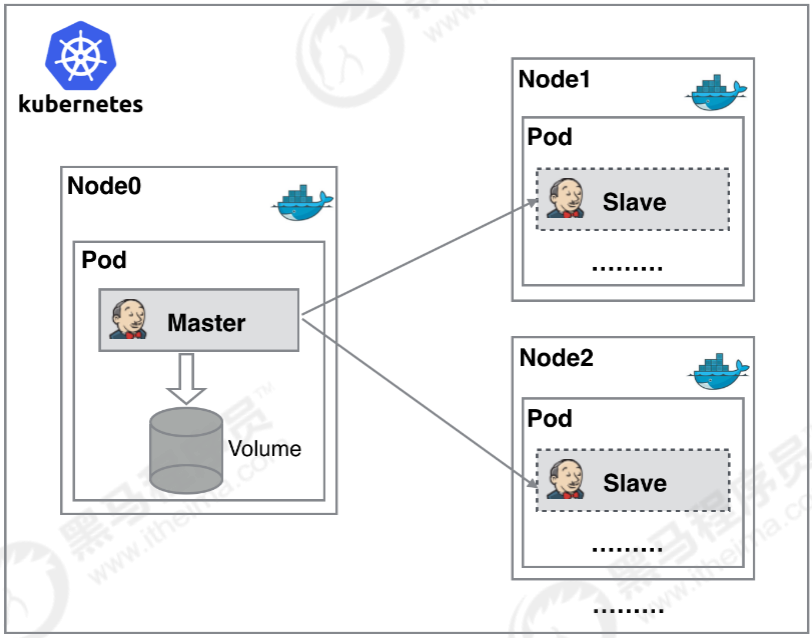
以上种种问题，我们可以引入Kubernates来解决！

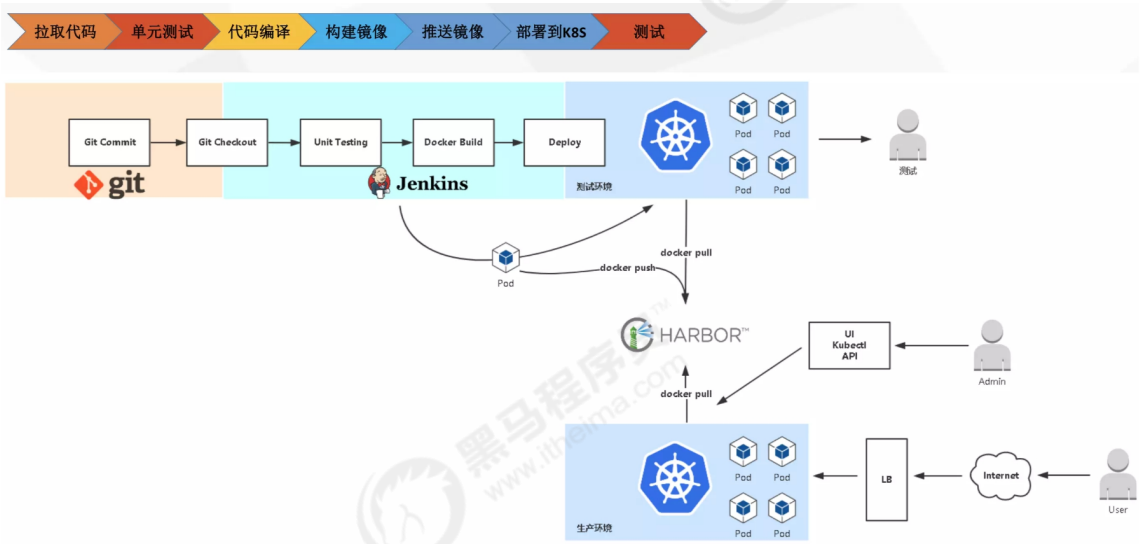
Kubernates简介

Kubernetes（简称，K8S）是Google开源的容器集群管理系统，在Docker技术的基础上，为容器化的应用提供部署运行、资源调度、服务发现和动态伸缩等一系列完整功能，提高了大规模容器集群管理的便捷性。 其主要功能如下：

- 使用Docker对应用程序包装(package)、实例化(instantiate)、运行(run)。
- 以集群的方式运行、管理跨机器的容器。以集群的方式运行、管理跨机器的容器。
- 解决Docker跨机器容器之间的通讯问题。解决Docker跨机器容器之间的通讯问题。
- Kubernetes的自我修复机制使得容器集群总是运行在用户期望的状态。

Kubernates+Docker+Jenkins持续集成架构图





大致工作流程：手动/自动构建 -> Jenkins 调度 K8S API - > 动态生成 Jenkins Slave pod - > Slave pod 拉取 Git 代码 / 编译 / 打包镜像 - > 推送到镜像仓库 Harbor - > Slave 工作完成，Pod 自动销毁 - > 部署到测试或生产 Kubernetes平台。（完全自动化，无需人工干预）

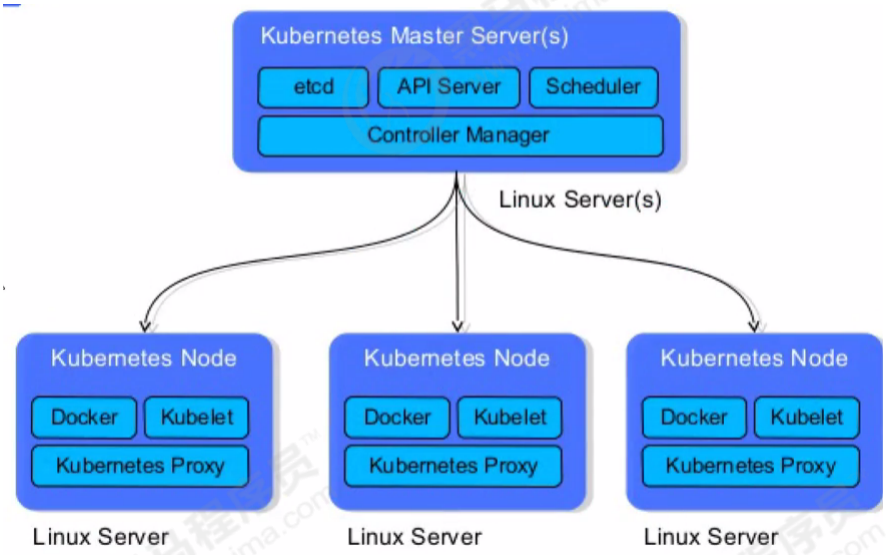
Kubernates+Docker+Jenkins持续集成方案好处

- 服务高可用：当 Jenkins Master 出现故障时，Kubernetes 会自动创建一个新的 Jenkins Master 容器，并且将 Volume 分配给新创建的容器，保证数据不丢失，从而达到集群服务高可用。

- 动态伸缩，合理使用资源：每次运行 Job 时，会自动创建一个 Jenkins Slave，Job 完成后，Slave 自动注销并删除容器，资源自动释放，而且 Kubernetes 会根据每个资源的使用情况，动态分配 Slave 到空闲的节点上创建，降低出现因某节点资源利用率高，还排队等待在该节点的情况。
- 扩展性好：当 Kubernetes 集群的资源严重不足而导致 Job 排队等待时，可以很容易的添加一个 Kubernetes Node 到集群中，从而实现扩展。

Kubeadm安装Kubernetes

Kubernetes的架构



API Server：用于暴露Kubernetes API，任何资源的请求的调用操作都是通过kube-apiserver提供的接口进行的。

Etcd：是Kubernetes提供默认的存储系统，保存所有集群数据，使用时需要为etcd数据提供备份计划。

Controller-Manager：作为集群内部的管理控制中心，负责集群内的Node、Pod副本、服务端点（Endpoint）、命名空间（Namespace）、服务账号（ServiceAccount）、资源定额（ResourceQuota）的管理，当某个Node意外宕机时，Controller Manager会及时发现并执行自动化修复流程，确保集群始终处于预期的工作状态。

Scheduler：监视新创建没有分配到Node的Pod，为Pod选择一个Node。

Kubelet：负责维护容器的生命周期，同时负责Volume和网络的管理

Kube proxy：是Kubernetes的核心组件，部署在每个Node节点上，它是实现Kubernetes Service的通信与负载均衡机制的重要组件。

安装环境说明

主机名称	IP地址	安装的软件
代码托管服务器	192.168.66.100	Gitlab-12.4.2
Docker仓库服务器	192.168.66.102	Harbor1.9.2
k8s-master	192.168.66.101	kube-apiserver、kube-controller-manager、kube-scheduler、docker、etcd、calico，NFS
k8s-node1	192.168.66.103	kubelet、kubeproxy、Docker18.06.1-ce
k8s-node2	192.168.66.104	kubelet、kubeproxy、Docker18.06.1-ce

三台机器都需要完成

修改三台机器的hostname及hosts文件

> hostnamectl set-hostname k8s-master
> hostnamectl set-hostname k8s-node1 hostnamectl set-hostname k8s-node2

> cat >>/etc/hosts<<EOF 192.168.66.101 k8s-master 192.168.66.103 k8s-node1
> 192.168.66.104 k8s-node2 EOF

关闭防火墙和关闭SELinux

> systemctl stop firewalld
>
> systemctl disable firewalld
>
> setenforce 0 临时关闭
>
> vi /etc/sysconfig/selinux 永久关闭

改为SELINUX=disabled

设置系统参数

设置允许路由转发，不对bridge的数据进行处理

创建文件

> vi /etc/sysctl.d/k8s.conf
>
> 内容如下：
>
> net.bridge.bridge-nf-call-ip6tables = 1 net.bridge.bridge-nf-call-iptables = 1
> net.ipv4.ip_forward = 1 vm.swappiness = 0

执行文件

> sysctl -p /etc/sysctl.d/k8s.conf

kube-proxy开启ipvs的前置条件

```
cat > /etc/sysconfig/modules/ipvs.modules <<EOF
#!/bin/bash
modprobe -- ip_vs
modprobe -- ip_vs_rr
modprobe -- ip_vs_wrr
modprobe -- ip_vs_sh
modprobe -- nf_conntrack_ipv4
EOF
chmod 755 /etc/sysconfig/modules/ipvs.modules && bash
/etc/sysconfig/modules/ipvs.modules && lsmod | grep -e ip_vs -e
nf_conntrack_ipv4
```

所有节点关闭swap

> swapoff -a 临时关闭
>
> vi /etc/fstab 永久关闭
>
> 注释掉以下字段
>
> /dev/mapper/cl-swap swap swap defaults 0 0

安装kubelet、kubeadm、kubectl

- kubeadm: 用来初始化集群的指令。
- kubelet: 在集群中的每个节点上用来启动 pod 和 container 等。
- kubectl: 用来与集群通信的命令行工具。

清空yum缓存

```
yum clean all
```

设置yum安装源

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64/
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF
```

安装：

```
yum install -y kubelet kubeadm kubectl
```

kubelet设置开机启动（注意：先不启动，现在启动的话会报错）

```
systemctl enable kubelet
```

查看版本

```
kubelet --version
```

安装的是最新版本：Kubernetes v1.16.3（可能会变化）

Master节点需要完成

1）运行初始化命令

```
kubeadm init --kubernetes-version=1.17.0 \
--apiserver-advertise-address=192.168.66.101 \
--image-repository registry.aliyuncs.com/google_containers \
--service-cidr=10.1.0.0/16 \
--pod-network-cidr=10.244.0.0/16
```

注意：apiserver-advertise-address这个地址必须是master机器的IP

常用错误：

错误一：[WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver

作为Docker cgroup驱动程序。，Kubernetes推荐的Docker驱动程序是“systemd”

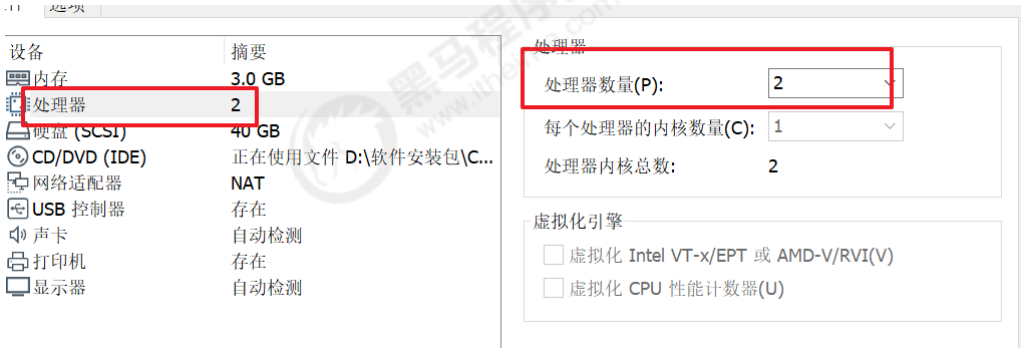
解决方案：修改Docker的配置: vi /etc/docker/daemon.json，加入

```
{
 "exec-opts":["native.cgroupdriver=systemd"]
}
```

然后重启Docker

错误二：[ERROR NumCPU]: the number of available CPUs 1 is less than the required 2

解决方案：修改虚拟机的CPU的个数，至少为2个



安装过程日志：

```
Last login: Sat Dec  7 16:38:59 2019 from 192.168.66.1
[root@k8s-master ~]# kubeadm init --kubernetes-version=1.16.3 \
> --apiserver-advertise-address=192.168.66.101 \
> --image-repository registry.aliyuncs.com/google_containers \
> --service-cidr=10.1.0.0/16 \
> --pod-network-cidr=10.244.0.0/16
[init] Using Kubernetes version: v1.16.3
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connect
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubead
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Activating the kubelet service
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [k8s-master kubernetes kubernetes.def
ubernetes.default.svc.cluster.local] and IPs [10.1.0.1 192.168.66.101]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [k8s-master localhost] and IPs [192
```

最后，会提示节点安装的命令，必须记下来

```
kubeadm join 192.168.66.101:6443 --token 754snw.9xq9cotze1ybwnti \
    --discovery-token-ca-cert-hash
sha256:3372ff6717ea5997121213e2c9d63fa7c8cdfb031527e17f2e20254f382ea03a
```

2) 启动kubelet

```
systemctl restart kubelet
```

```
    --discovery-token-ca-cert-hash sha256:86a6e0e0fbe149e9f87b64b6f
[root@k8s-master ~]# systemctl restart kubelet
[root@k8s-master ~]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since 六 2019-12-07 17:48:05 CST; 10s a
     Docs: https://kubernetes.io/docs/
 Main PID: 5670 (kubelet)
    Tasks: 15
   Memory: 33.1M
   CGroup: /system.slice/kubelet.service
```

3) 配置kubectl工具

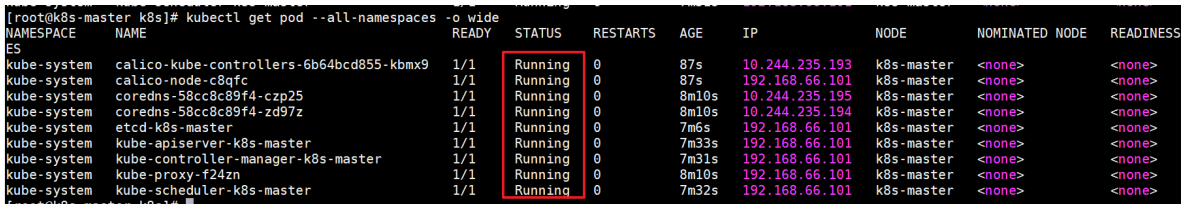
```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

4) 安装Calico

```
mkdir k8s
cd k8s
wget https://docs.projectcalico.org/v3.10/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml

sed -i 's/192.168.0.0/10.244.0.0/g' calico.yaml

kubectl apply -f calico.yaml
```

5）等待几分钟，查看所有Pod的状态，确保所有Pod都是Running状态

```
kubectl get pod --all-namespaces -o wide
```



```
[root@k8s-master k8s]# kubectl get pod --all-namespaces -o wide
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE     IP                NODE         NOMINATED NODE   READINESS
ES
kube-system   calico-kube-controllers-6b64bcd855-kbmx9   1/1     Running   0          87s     10.244.235.193    k8s-master   <none>           <none>
kube-system   calico-node-c8qfc                          1/1     Running   0          87s     192.168.66.101    k8s-master   <none>           <none>
kube-system   coredns-58cc8c89f4-czp25                   1/1     Running   0          8m10s   10.244.235.195    k8s-master   <none>           <none>
kube-system   coredns-58cc8c89f4-zd97z                   1/1     Running   0          8m10s   10.244.235.194    k8s-master   <none>           <none>
kube-system   etcd-k8s-master                            1/1     Running   0          7m6s    192.168.66.101    k8s-master   <none>           <none>
kube-system   kube-apiserver-k8s-master                  1/1     Running   0          7m33s   192.168.66.101    k8s-master   <none>           <none>
kube-system   kube-controller-manager-k8s-master         1/1     Running   0          7m31s   192.168.66.101    k8s-master   <none>           <none>
kube-system   kube-proxy-f24zn                           1/1     Running   0          8m10s   192.168.66.101    k8s-master   <none>           <none>
kube-system   kube-scheduler-k8s-master                  1/1     Running   0          7m32s   192.168.66.101    k8s-master   <none>           <none>
```

Slave节点需要完成

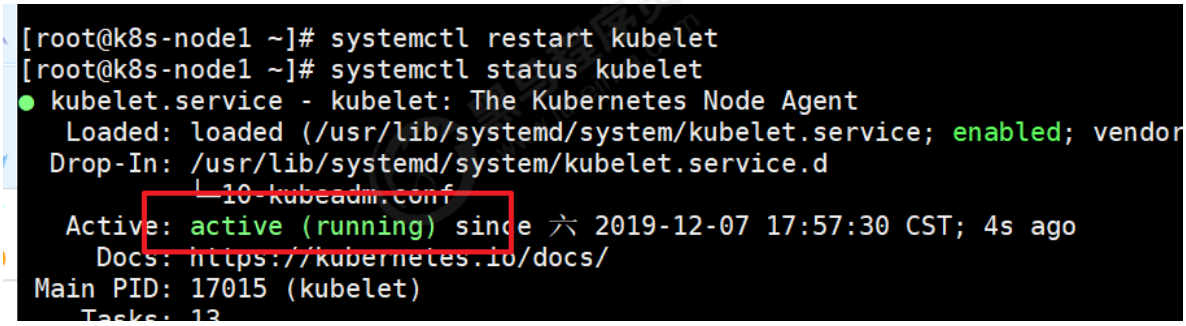
1）让所有节点让集群环境

使用之前Master节点产生的命令加入集群

```
kubeadm join 192.168.66.101:6443 --token 754snw.9xq9cotze1ybwnti \
    --discovery-token-ca-cert-hash
sha256:3372ff6717ea5997121213e2c9d63fa7c8cdfb031527e17f2e20254f382ea03a
```

2）启动kubelet

```
systemctl start kubelet
```



```
[root@k8s-node1 ~]# systemctl restart kubelet
[root@k8s-node1 ~]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since 六 2019-12-07 17:57:30 CST; 4s ago
     Docs: https://kubernetes.io/docs/
 Main PID: 17015 (kubelet)
    Tasks: 13
```

3）回到Master节点查看，如果Status全部为Ready，代表集群环境搭建成功！！！

```
kubectl get nodes
```

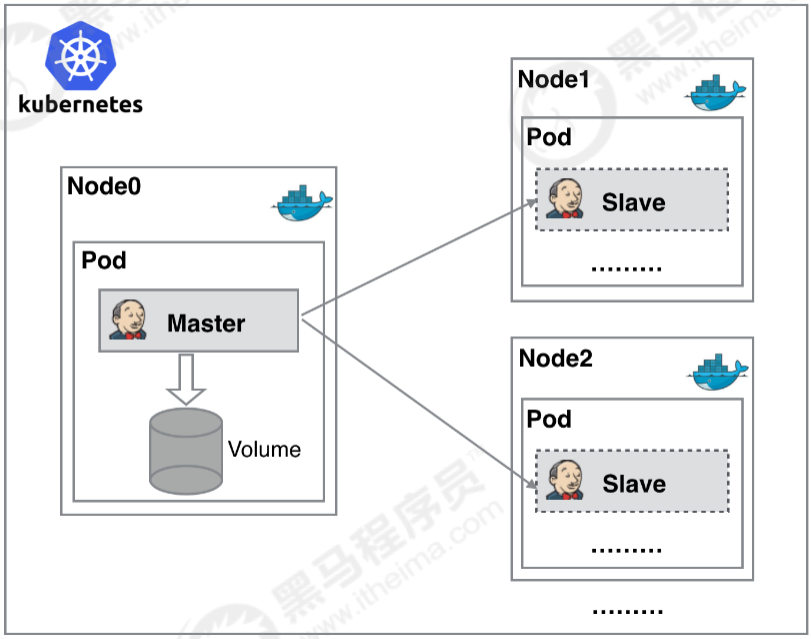
```
[root@k8s-master k8s]# kubectl get nodes
NAME         STATUS   ROLES    AGE    VERSION
k8s-master   Ready    master   174m   v1.16.3
k8s-node1    Ready    <none>   163m   v1.16.3
k8s-node2    Ready    <none>   46s    v1.16.3
```

kubectl常用命令

```
kubectl get nodes     查看所有主从节点的状态
kubectl get ns    获取所有namespace资源
kubectl get pods -n {$nameSpace}  获取指定namespace的pod
kubectl describe pod的名称  -n {$nameSpace}   查看某个pod的执行过程
kubectl logs --tail=1000 pod的名称 | less  查看日志
kubectl create -f xxx.yml  通过配置文件创建一个集群资源对象
kubectl delete -f xxx.yml  通过配置文件删除一个集群资源对象
kubectl delete pod名称 -n {$nameSpace}   通过pod删除集群资源
kubectl get service -n {$nameSpace}    查看pod的service情况
```

7、基于Kubernetes/K8S构建Jenkins持续集成平台(下)

Jenkins-Master-Slave架构图回顾：



安装和配置NFS

NFS简介

NFS（Network File System），它最大的功能就是可以通过网络，让不同的机器、不同的操作系统可以共享彼此的文件。我们可以利用NFS共享Jenkins运行的配置文件、Maven的仓库依赖文件等

NFS安装

我们把NFS服务器安装在192.168.66.101机器上

1）安装NFS服务（在所有K8S的节点都需要安装）

```
yum install -y nfs-utils
```

2）创建共享目录

```
mkdir -p /opt/nfs/jenkins
vi /etc/exports      编写NFS的共享配置
内容如下:
/opt/nfs/jenkins    *(rw,no_root_squash)        *代表对所有IP都开放此目录，rw是读写
```

3）启动服务

```
systemctl enable nfs    开机启动
systemctl start nfs      启动
```

4）查看NFS共享目录

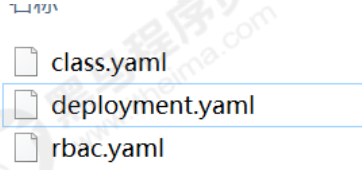
```
showmount -e 192.168.66.101
```

在Kubernetes安装Jenkins-Master

创建NFS client provisioner

nfs-client-provisioner 是一个Kubernetes的简易NFS的外部provisioner，本身不提供NFS，需要现有的NFS服务器提供存储。

1）上传nfs-client-provisioner构建文件



其中注意修改deployment.yaml，使用之前配置NFS服务器和目录

```
            mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: fuseim.pri/ifs
            - name: NFS_SERVER
              value: 192.168.66.101
            - name: NFS_PATH
              value: /opt/nfs/jenkins
      volumes:
        - name: nfs-client-root
          nfs:
            server: 192.168.66.101
            path: /opt/nfs/jenkins
```

2）构建nfs-client-provisioner的pod资源

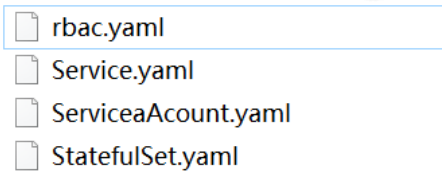
```
cd nfs-client
kubectl create -f .
```

3）查看pod是否创建成功

```
[root@k8s-master nfs-client]# kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
nfs-client-provisioner-86d4699744-ppg25  1/1     Running   0          35s
[root@k8s-master nfs-client]#
```

安装Jenkins-Master

1）上传Jenkins-Master构建文件



其中有两点注意：

第一、在StatefulSet.yaml文件，声明了利用nfs-client-provisioner进行Jenkins-Master文件存储

```
  volumeClaimTemplates:
  - metadata:
      name: jenkins-home
    spec:
      storageClassName: "managed-nfs-storage"
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

第二、Service发布方法采用NodePort，会随机产生节点访问端口

```
spec:
  selector:
    app: jenkins
  type: NodePort
  ports:
  - name: web
    port: 8080
    targetPort: web
  - name: agent
    port: 50000
    targetPort: agent
```

2) 创建kube-ops的namespace

因为我们把Jenkins-Master的pod放到kube-ops下

```
kubectl create namespace kube-ops
```

3) 构建Jenkins-Master的pod资源

```
cd jenkins-master
kubectl create -f .
```

4) 查看pod是否创建成功

```
kubectl get pods -n kube-ops
```

5) 查看信息，并访问

查看Pod运行在那个Node上

```
kubectl describe pods -n kube-ops
```

```
Type     Reason     Age                From               Message
----     ------     ----               ----               -------
Normal   Scheduled  4m26s              default-scheduler  Successfully assigned kube-ops/jenkins-0 to k8s-node1
Normal   Pulling    4m24s              kubelet, k8s-node1 Pulling image "jenkins/jenkins:lts-alpine"
Normal   Pulled     3m46s              kubelet, k8s-node1 Successfully pulled image "jenkins/jenkins:lts-alpine"
Normal   Created    3m46s              kubelet, k8s-node1 Created container jenkins
Normal   Started    3m45s              kubelet, k8s-node1 Started container jenkins
```

查看分配的端口

```
kubectl get service -n kube-ops
```

```
[root@k8s-master jenkins-master]# kubectl get service -n kube-ops
NAME      TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)                          AGE
jenkins   NodePort   10.1.121.82    <none>        8080:30136/TCP,50000:31370/TCP   6m1s
```

最终访问地址为：http://192.168.66.103:30136 （192.168.66.103为k8s-node1的IP）

安装过程跟之前是一样的！

6）先安装基本的插件

- Localization:Chinese
- Git
- Pipeline
- Extended Choice Parameter

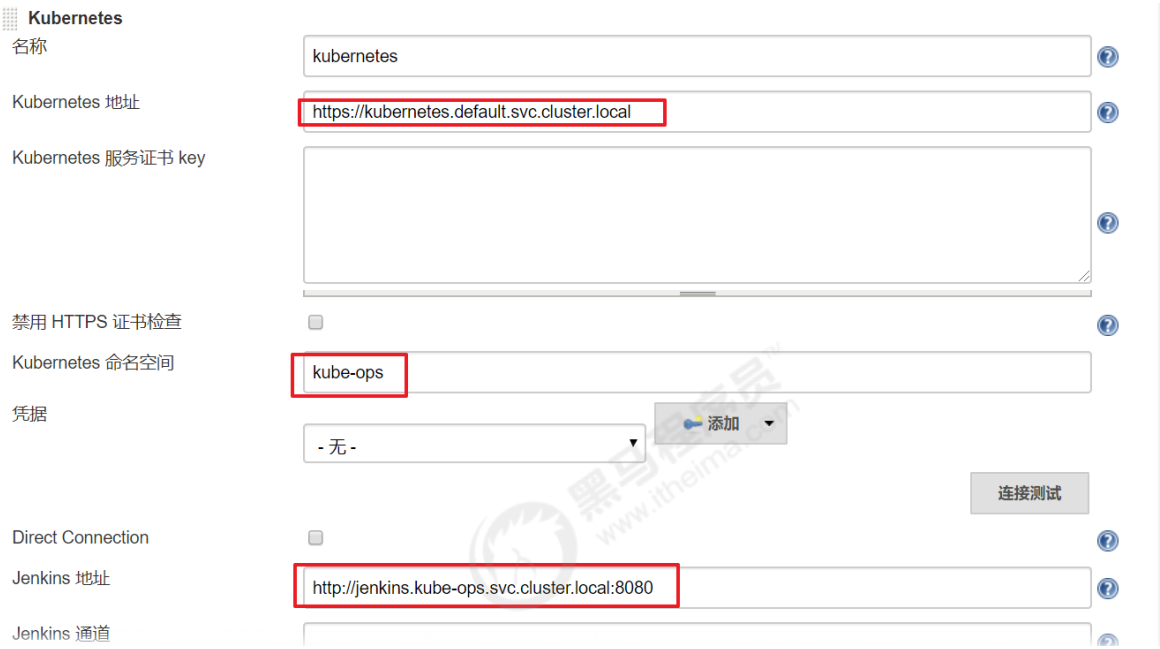
Jenkins与Kubernetes整合

安装Kubernetes插件

系统管理->插件管理->可选插件

实现Jenkins与Kubernetes整合

系统管理->系统配置->云->新建云->Kubernetes

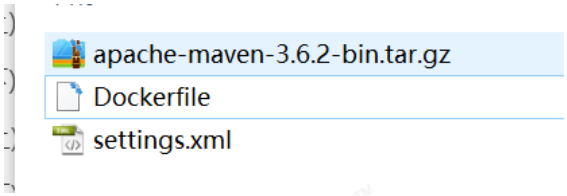


- kubernetes地址采用了kube的服务器发现：https://kubernetes.default.svc.cluster.local
- namespace填kube-ops，然后点击Test Connection，如果出现 Connection test successful 的提示信息证明 Jenkins 已经可以和 Kubernetes 系统正常通信
- Jenkins URL 地址：http://jenkins.kube-ops.svc.cluster.local:8080

构建Jenkins-Slave自定义镜像

Jenkins-Master在构建Job的时候，Kubernetes会创建Jenkins-Slave的Pod来完成Job的构建。我们选择运行Jenkins-Slave的镜像为官方推荐镜像：jenkins/jnlp-slave:latest，但是这个镜像里面并没有Maven环境，为了方便使用，我们需要自定义一个新的镜像：

准备材料：



Dockerfile文件内容如下：

```
FROM jenkins/jnlp-slave:latest

MAINTAINER itcast

# 切换到 root 账户进行操作
USER root

# 安装 maven
COPY apache-maven-3.6.2-bin.tar.gz .

RUN tar -zxf apache-maven-3.6.2-bin.tar.gz && \
    mv apache-maven-3.6.2 /usr/local && \
    rm -f apache-maven-3.6.2-bin.tar.gz && \
    ln -s /usr/local/apache-maven-3.6.2/bin/mvn /usr/bin/mvn && \
```

```
    ln -s /usr/local/apache-maven-3.6.2 /usr/local/apache-maven && \
    mkdir -p /usr/local/apache-maven/repo

COPY settings.xml /usr/local/apache-maven/conf/settings.xml

USER jenkins
```

构建出一个新镜像：jenkins-slave-maven:latest

然把镜像上传到Harbor的公共库library中

```
docker tag jenkins-slave-maven:latest 192.168.66.102:85/library/jenkins-slave-maven:latest
docker push 192.168.66.102:85/library/jenkins-slave-maven:latest
```

测试Jenkins-Slave是否可以创建

1）创建一个Jenkins流水线项目

2）编写Pipeline，从GItlab拉取代码

```
def git_address =
"http://192.168.66.100:82/itheima_group/tensquare_back_cluster.git"
def git_auth = "9d9a2707-eab7-4dc9-b106-e52f329cbc95"

//创建一个Pod的模板，label为jenkins-slave
podTemplate(label: 'jenkins-slave', cloud: 'kubernetes', containers: [
    containerTemplate(
        name: 'jnlp',
        image: "192.168.66.102:85/library/jenkins-slave-maven:latest"
    )
  ]
)
{
  //引用jenkins-slave的pod模块来构建Jenkins-Slave的pod
  node("jenkins-slave"){
      // 第一步
      stage('拉取代码'){
         checkout([$class: 'GitSCM', branches: [[name: 'master']],
userRemoteConfigs: [[credentialsId: "${git_auth}", url: "${git_address}"]]])
      }
  }
```

```
}
```

3）查看构建日志

Jenkins+Kubernetes+Docker完成微服务持续集成

拉取代码，构建镜像

1）创建NFS共享目录

让所有Jenkins-Slave构建指向NFS的Maven的共享仓库目录

```
vi /etc/exports
添加内容：
/opt/nfs/jenkins    *(rw,no_root_squash)
/opt/nfs/maven      *(rw,no_root_squash)
systemctl restart nfs   重启NFS
```

2）创建项目，编写构建Pipeline

```
def git_address =
"http://192.168.66.100:82/itheima_group/tensquare_back_cluster.git"
def git_auth = "9d9a2707-eab7-4dc9-b106-e52f329cbc95"
//构建版本的名称
def tag = "latest"
//Harbor私服地址
def harbor_url = "192.168.66.102:85"
//Harbor的项目名称
def harbor_project_name = "tensquare"
//Harbor的凭证
def harbor_auth = "71eff071-ec17-4219-bae1-5d0093e3d060"


podTemplate(label: 'jenkins-slave', cloud: 'kubernetes', containers: [
    containerTemplate(
        name: 'jnlp',
        image: "192.168.66.102:85/library/jenkins-slave-maven:latest"
    ),
    containerTemplate(
        name: 'docker',
        image: "docker:stable",
        ttyEnabled: true,
        command: 'cat'
    ),
  ],
  volumes: [
    hostPathVolume(mountPath: '/var/run/docker.sock', hostPath:
'/var/run/docker.sock'),
    nfsVolume(mountPath: '/usr/local/apache-maven/repo', serverAddress:
'192.168.66.101' , serverPath: '/opt/nfs/maven'),
  ],
)
{
  node("jenkins-slave"){
      // 第一步
      stage('拉取代码'){
         checkout([$class: 'GitSCM', branches: [[name: '${branch}']],
userRemoteConfigs: [[credentialsId: "${git_auth}", url: "${git_address}"]]])
      }
      // 第二步
      stage('代码编译'){
           //编译并安装公共工程
         sh "mvn -f tensquare_common clean install"
      }
      // 第三步
      stage('构建镜像，部署项目'){
            //把选择的项目信息转为数组
            def selectedProjects = "${project_name}".split(',')

            for(int i=0;i<selectedProjects.size();i++){
                //取出每个项目的名称和端口
                def currentProject = selectedProjects[i];
                //项目名称
                def currentProjectName = currentProject.split('@')[0]
                //项目启动端口
                def currentProjectPort = currentProject.split('@')[1]

                 //定义镜像名称
```

```
                def imageName = "${currentProjectName}:${tag}"

                //编译，构建本地镜像
                sh "mvn -f ${currentProjectName} clean package
dockerfile:build"
                container('docker') {

                    //给镜像打标签
                    sh "docker tag ${imageName}
${harbor_url}/${harbor_project_name}/${imageName}"

                    //登录Harbor，并上传镜像
                    withCredentials([usernamePassword(credentialsId:
"${harbor_auth}", passwordVariable: 'password', usernameVariable: 'username')])
{
                        //登录
                        sh "docker login -u ${username} -p ${password}
${harbor_url}"
                        //上传镜像
                        sh "docker push
${harbor_url}/${harbor_project_name}/${imageName}"
                    }

                    //删除本地镜像
                    sh "docker rmi -f ${imageName}"
                    sh "docker rmi -f
${harbor_url}/${harbor_project_name}/${imageName}"
                }
            }
        }
    }
}
```

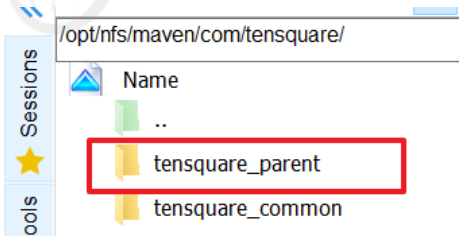
注意：在构建过程会发现无法创建仓库目录，是因为NFS共享目录权限不足，需更改权限

```
chown -R jenkins:jenkins /opt/nfs/maven
chmod -R 777 /opt/nfs/maven
```

还有Docker命令执行权限问题

```
chmod 777 /var/run/docker.sock
```

需要手动上传父工程依赖到NFS的Maven共享仓库目录中



微服务部署到K8S

修改每个微服务的application.yml

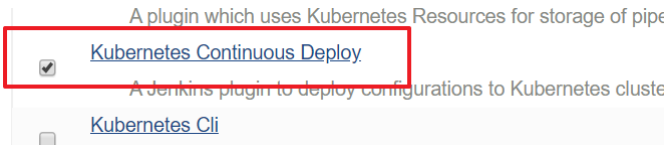
Eureka

```
server:
  port: ${PORT:10086}
spring:
  application:
    name: eureka

eureka:
  server:
    # 续期时间，即扫描失效服务的间隔时间（缺省为60*1000ms）
    eviction-interval-timer-in-ms: 5000
    enable-self-preservation: false
    use-read-only-response-cache: false
  client:
    # eureka client间隔多久去拉取服务注册信息 默认30s
    registry-fetch-interval-seconds: 5
    serviceUrl:
      defaultZone: ${EUREKA_SERVER:http://127.0.0.1:${server.port}/eureka/}
  instance:
    # 心跳间隔时间，即发送一次心跳之后，多久在发起下一次（缺省为30s）
    lease-renewal-interval-in-seconds: 5
    # 在收到一次心跳之后，等待下一次心跳的空档时间，大于心跳间隔即可，即服务续约到期时间（缺省
为90s）
    lease-expiration-duration-in-seconds: 10
    instance-id:
${EUREKA_INSTANCE_HOSTNAME:${spring.application.name}}:${server.port}@${random.l
ong(1000000,9999999)}
    hostname: ${EUREKA_INSTANCE_HOSTNAME:${spring.application.name}}
```

其他微服务需要注册到所有Eureka中

```
# Eureka配置
eureka:
  client:
    serviceUrl:
      defaultZone: http://eureka-0.eureka:10086/eureka/,http://eureka-
1.eureka:10086/eureka/ # Eureka访问地址
  instance:
    preferIpAddress: true
```

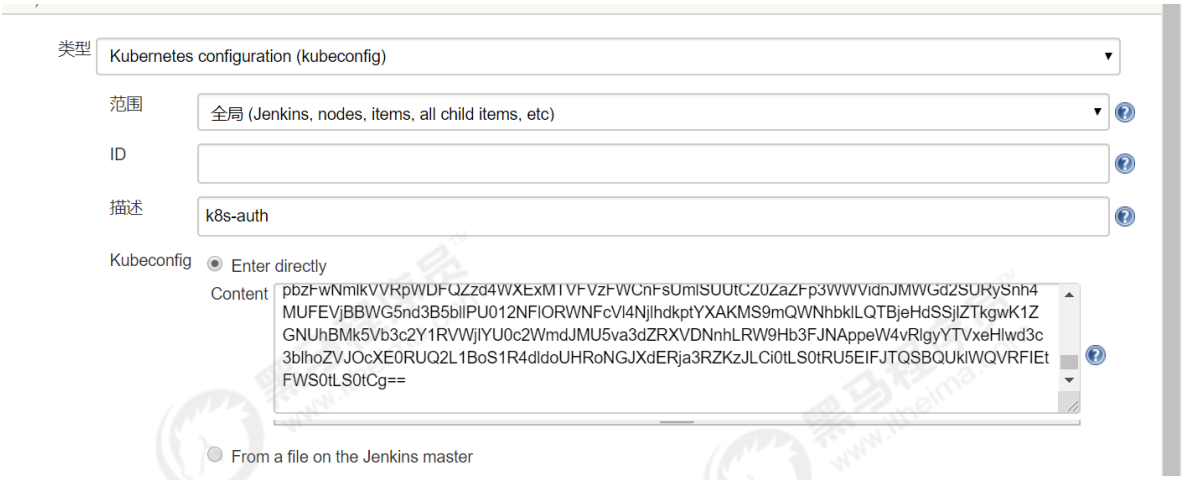
1）安装Kubernetes Continuous Deploy插件



2）修改后的流水线脚本

```
 def deploy_image_name = "${harbor_url}/${harbor_project_name}/${imageName}"

                //部署到K8S
                sh """
                       sed -i 's#\$IMAGE_NAME#${deploy_image_name}#'
${currentProjectName}/deploy.yml
                       sed -i 's#\$SECRET_NAME#${secret_name}#'
${currentProjectName}/deploy.yml
                """

                kubernetesDeploy configs: "${currentProjectName}/deploy.yml",
kubeconfigId: "${k8s_auth}"
```

3) 建立k8s认证凭证



kubeconfig到k8s的Master节点复制

```
cat /root/.kube/config
```

5) 生成Docker凭证

Docker凭证，用于Kubernetes到Docker私服拉取镜像

```
docker login -u itcast -p Itcast123 192.168.66.102:85   登录Harbor


kubectl create secret docker-registry registry-auth-secret  --docker-
server=192.168.66.102:85 --docker-username=itcast --docker-password=Itcast123 --
docker-email=itcast@itcast.cn    生成

kubectl get secret    查看密钥
```

6) 在每个项目下建立deploy.xml

Eureka的deply.yml

```

```

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: eureka
  labels:
    app: eureka
spec:
  type: NodePort
  ports:
    - port: 10086
      name: eureka
      targetPort: 10086
  selector:
    app: eureka
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: eureka
spec:
  serviceName: "eureka"
  replicas: 2
  selector:
    matchLabels:
      app: eureka
  template:
    metadata:
      labels:
        app: eureka
    spec:
      imagePullSecrets:
        - name: $SECRET_NAME
      containers:
        - name: eureka
          image: $IMAGE_NAME
          ports:
            - containerPort: 10086
          env:
            - name: MY_POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: EUREKA_SERVER
              value: "http://eureka-0.eureka:10086/eureka/,http://eureka-
1.eureka:10086/eureka/"
            - name: EUREKA_INSTANCE_HOSTNAME
              value: ${MY_POD_NAME}.eureka
  podManagementPolicy: "Parallel"
```

其他项目的deploy.yml主要把名字和端口修改：

```yaml
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: zuul
  labels:
    app: zuul
spec:
  type: NodePort
  ports:
    - port: 10020
      name: zuul
      targetPort: 10020
  selector:
    app: zuul
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: zuul
spec:
  serviceName: "zuul"
  replicas: 2
  selector:
    matchLabels:
      app: zuul
  template:
    metadata:
      labels:
        app: zuul
    spec:
      imagePullSecrets:
        - name: $SECRET_NAME
      containers:
        - name: zuul
          image: $IMAGE_NAME
          ports:
            - containerPort: 10020
  podManagementPolicy: "Parallel"
```

7）项目构建后，查看服务创建情况

```
kubectl get pods -owide
kubectl get service
```

效果如下：

EUREKA	n/a (2)	(2)	UP (2) - eureka-0.eureka:10086@4439740 , eureka-1.eureka:10086@6737812
TENSQUARE-GATHERING	n/a (2)	(2)	UP (2) - gathering-0.gathering.default.svc.cluster.local:tensquare-gathering:9002 , gathering-1.gathering.default.svc.cluster.local:tensquare-gathering:9002
TENSQUARE-ZUUL	n/a (2)	(2)	UP (2) - zuul-1.zuul.default.svc.cluster.local:tensquare-zuul:10020 , zuul-0.zuul.default.svc.cluster.local:tensquare-zuul:10020