

从所有教程的词条中查询...

首页 > 慕课教程 > Go工程师体系课全新版 > 8. jenkins的pipeline参数详解

全部开发者教程

1. go最常用的设计模式 - 函数选项

2. 单例模式和懒加载

3. 测试金字塔

第23周 protoc插件开发、cobra命令行

1. protoc调试源码

2. protoc自定义gin插件

第24周 log日志包设计

日志源码

第25周 ast代码生成工具开发

错误码

第26周 三层代码结构

通用app项目启动



bobby · 更新于 2022-11-16

上一节 7. 部署到远程服... 9. 定时构建的语法 下一节

官方文档

pipeline模式相对style free门槛较高， 比较复杂的构建， 以及构建过程很重要

Pipeline 是什么

Jenkins Pipeline 实际上是基于 Groovy 实现的 CI/CD 领域特定语言（DSL）， 主要分为两类， 一类叫做 Declarative Pipeline， 一类叫做 Scripted Pipeline。

Declarative Pipeline 体验上更接近于我们熟知的 travis CI 的 travis.yml， 通过声明自己要做的 事情来规范流程， 形如：

<> 代码块

```
1 pipeline {
2   agent any
3   stages {
4     stage('Build') {
5       steps {
6         //
7       }
8     }
9     stage('Test') {
10      steps {
11        //
12      }
13    }
14    stage('Deploy') {
15      steps {
16        //
17      }
18    }
19  }
20 }
```

而 Scripted Pipeline 则是旧版本中 Jenkins 支持的 Pipeline 模式， 主要是写一些 groovy 的代码来制定流程：

<> 代码块

```
1 node {
2   stage('Build') {
3     //
4   }
5   stage('Test') {
6     //
7   }
```

意见反馈

收藏教程

标记书签

```
10     }
11 }
```

一般情况下声明式的流水线已经可以满足我们的需要，只有在复杂的情况下才会需要脚本式流水线的参与。

过去大家经常在 Jenkins 的界面上直接写脚本来实现自动化，但是现在更鼓励大家通过在项目中增加 `Jenkinsfile` 的方式把流水线固定下来，实现 `Pipeline As Code`，Jenkins 的 Pipeline 插件将会自动发现并执行它。

语法

`Declarative Pipeline` 最外层有个 `pipeline` 表明它是一个声明式流水线，下面会有 4 个主要的部分：`agent`，`post`，`stages`，`steps`，我会逐一介绍一下。

Agent

`agent` 主要用于描述整个 Pipeline 或者指定的 Stage 由什么规则来选择节点执行。Pipeline 级别的 agent 可以视为 Stage 级别的默认值，如果 stage 中没有指定，将会使用与 Pipeline 一致的规则。在最新的 Jenkins 版本中，可以支持指定任意节点(`any`)，不指定(`none`)，标签(`label`)，节点(`node`)，`docker`，`dockerfile` 和 `kubernetes` 等，具体的配置细节可以查看文档，下面是一个使用 `docker` 的示例：

<> 代码块

```
1  agent {
2      docker {
3          image 'myregistry.com/node'
4          label 'my-defined-label'
5          registryUrl 'https://myregistry.com/'
6          registryCredentialsId 'myPredefinedCredentialsInJenkins'
7          args '-v /tmp:/tmp'
8      }
9  }
```

Tips:

- 如果 Pipeline 选择了 `none`，那么 stage 必须要指定一个有效的 agent，否则无法执行
- Jenkins 总是会使用 master 来执行 `scan multibranch` 之类的操作，即使 master 配置了 0 executors
- agent 指定的是规则而不是具体的节点，如果 stage 各自配置了自己的 agent，需要注意是不是在同一个节点执行的

Stages && Stage

Stages 是 Pipeline 中最主要的组成部分，Jenkins 将会按照 Stages 中描述的顺序从上往下的执行。Stages 中可以包括任意多个 Stage，而 Stage 与 Stages 又能互相嵌套，除此以外还有 `parallel` 指令可以让内部的 Stage 并行运行。实际上可以把 Stage 当作最小单元，Stages 指定的是顺序运行，而 `parallel` 指定的是并行运行。

接下来的这个 case 很好的说明了这一点：

<> 代码块

```
1  pipeline {
2      agent none
3      stages {
4          stage('Sequential') {
```

```
6         stage('In Sequential 1') {
7             steps {
8                 echo "In Sequential 1"
9             }
10        }
11        stage('In Sequential 2') {
12            steps {
13                echo "In Sequential 2"
14            }
15        }
16        stage('Parallel In Sequential') {
17            parallel {
18                stage('In Parallel 1') {
19                    steps {
20                        echo "In Parallel 1"
21                    }
22                }
23                stage('In Parallel 2') {
24                    steps {
25                        echo "In Parallel 2"
26                    }
27                }
28            }
29        }
30    }
31 }
32 }
33 }
```

除了指定 Stage 之间的顺序关系之外，我们还可以通过 `when` 来指定某个 Stage 指定与否：比如要配置只有在 Master 分支上才执行 push，其他分支上都只运行 build

<> 代码块

```
1  stages {
2      stage('Build') {
3          when {
4              not { branch 'master' }
5          }
6          steps {
7              sh './scripts/run.py build'
8          }
9      }
10     stage('Run') {
11         when {
12             branch 'master'
13         }
14         steps {
15             sh './scripts/run.py push'
16         }
17     }
18 }
```

还能在 Stage 的级别设置 `environment`，这些就不展开了，文档里有更详细的描述。

Steps

`steps` 是 Pipeline 中最核心的部分，每个 Stage 都需要指定 Steps。Steps 内部可以执行一系列的操作，任意操作执行出错都会返回错误。完整的 Steps 操作列表可以参考 [Pipeline Steps Reference](#)，这里只说一些使用时需要注意的点。

意见反馈

收藏教程

标记书签

- groovy 语法中有不同的字符串类型，其中 'abc' 是 Plain 字符串，不会转义 `${WORKSPACE}` 这样的变量，而 "abc" 会做这样的转换。此外还有 `''' xxx '''` 支持跨行字符串，`"""` 同理。
- 调用函数的 () 可以省略，使得函数调用形如 `updateGitlabCommitStatus name: 'build', state: 'success'`，通过 `,` 来分割不同的参数，支持换行。
- 可以在声明式流水线中通过 `script` 来插入一段 groovy 脚本

Post

`post` 部分将会在 pipeline 的最后执行，经常用于一些测试完毕后的清理和通知操作。文档中给出了一系列的情况，比较常用的是 `always`，`success` 和 `failure`。

比如说下面的脚本将会在成功和失败的时候更新 gitlab 的状态，在失败的时候发送通知邮件：

<> 代码块

```
1  post {
2    failure {
3      updateGitlabCommitStatus name: 'build', state: 'failed'
4      emailx body: '$DEFAULT_CONTENT', recipientProviders: [culprits()]
5    }
6    success {
7      updateGitlabCommitStatus name: 'build', state: 'success'
8    }
9  }
```

每个状态其实都相当于于一个 `steps`，都能够执行一系列的操作，不同状态的执行顺序是事先规定好的，就是文档中列出的顺序。

Shared Libraries

同一个 Team 产出的不同项目往往会有着相似的流程，比如 `golang` 的大部分项目都会执行同样的命令。这就导致了人们经常需要在不同的项目间复制同样的流程，而 Shared Libraries 就解决了这个问题。通过在 Pipeline 中引入共享库，把常用的流程抽象出来变成一个的指令，简化了大量重复的操作。

在配置好 lib 之后，Jenkins 会在每个 Pipeline 启动前去检查 lib 是否更新并 pull 到本地，根据配置决定是否直接加载。

所有的 Shared Libraries 都要遵循相同的项目结构：

<> 代码块

```
1  (root)
2  +- src                                # Groovy source files
3  |   +- org
4  |       +- foo
5  |           +- Bar.groovy # for org.foo.Bar class
6  +- vars
7  |   +- foo.groovy         # for global 'foo' variable
8  |   +- foo.txt            # help for 'foo' variable
9  +- resources              # resource files (external libraries only)
10 |   +- org
11 |       +- foo
12 |           +- bar.json   # static helper data for org.foo.Bar
```

目前我们的使用比较低级，所以只用到了 `vars` 来存储全局的变量。

`vars` 下的每一个 `foo.groovy` 文件都是一个独立的 namespace，在 Pipeline 中可以以 `foo.XXX` 的形式来导入。比如我们有 `vars/log.groovy`：

<> 代码块

```
1  def info(message) {
2      echo "INFO: ${message}"
3  }
4  def warning(message) {
5      echo "WARNING: ${message}"
6  }
```

那么 Jenkinsfile 中就可以这样调用：

<> 代码块

```
1  // Jenkinsfile
2  steps {
3      log.info 'Starting'
4      log.warning 'Nothing to do!'
5  }
```

大家可能已经注意到了，在 `groovy` 文件中，我们可以直接像在 `steps` 中一样调用已有的方法，比如 `echo` 和 `sh` 等。

我们也能在 `groovy` 文件中去引用 Java 的库并返回一个变量，比如：

<> 代码块

```
1  #!/usr/bin/env groovy
2  import java.util.Random;
3  def String name() {
4      def rand = new Random()
5      def t = rand.nextInt(1000)
6      return String.valueOf(t)
7  }
```

这样就能在 `JenkinsFile` 中去设置一个环境变量：

<> 代码块

```
1  // Jenkinsfile
2  environment {
3      NAME = random.name()
4  }
```

除了定义方法之外，我们还能让这个文件本身就能被调用，只需要定义一个 `call` 方法：

<> 代码块

```
1  #!/usr/bin/env groovy
2  def call() {
3      sh "hello, world"
4  }
```

还能够定义一个新的 `section`，接受一个 `Block`：

<> 代码块

```
1  def call(Closure body) {
2      node('windows') {
3          body()
```

[意见反馈](#)[收藏教程](#)[标记书签](#)

```
5      }  
    }  
  }
```

这样可以让指定的 Body 在 windows 节点上调用：

<> 代码块

```
1  // Jenkinsfile  
2  windows {  
3      bat "cmd /?"  
4  }
```

常用技巧

发送邮件通知

主要使用 `emailext`，需要在 Jenkins 的配置界面事先配置好，可用的环境变量和参数可以参考文档 [Email-ext plugin](#)

<> 代码块

```
1  emailext body: '$DEFAULT_CONTENT', recipientProviders: [culprits(),dev
```

结果同步到 gitlab

同样需要配置好 gitlab 插件，在 Pipeline 中指定 `options`：

<> 代码块

```
1  // Jenkinsfile  
2  options {  
3      gitLabConnection('gitlab')  
4  }
```

然后就可以在 `post` 中根据不同的状态来更新 gitlab 了：

<> 代码块

```
1  // Jenkinsfile  
2  failure {  
3      updateGitlabCommitStatus name: 'build', state: 'failed'  
4  }  
5  success {  
6      updateGitlabCommitStatus name: 'build', state: 'success'  
7  }
```

文档参考：[Build status configuration](#)

构建过程中可用的环境变量列表

Jenkins 会提供一个完整的列表，只需要访问 `<your-jenkins-url>/env-vars.html/` 即可，别忘了需要使用 `"${WORKSPACE}"`

在 Multibranch Pipeline 的默认流程中会在 checkout 之前和之后执行 `git clean -fdx`，如果在测试中以 root 权限创建了文件，那么 jenkins 会因为这个命令执行失败而报错。所以我们需要在 checkout 之前执行自定义的任务：

<> 代码块

```
1  #!/usr/bin/env groovy
2  // var/pre.groovy
3  def call(Closure body) {
4      body()
5      checkout scm
6  }
```

在 Jenkinsfile 中配置以跳过默认的 checkout 行为：

<> 代码块

```
1  // Jenkinsfile
2  options {
3      skipDefaultCheckout true
4  }
```

在每个 stage 中执行自定义的任务即可：

<> 代码块

```
1  // Jenkinsfile
2  stage('Compile') {
3      agent any
4      steps {
5          pre {
6              sh 'pre compile'
7          }
8          sh 'real compile'
9      }
10 }
```

总结

Jenkins 作为使用最为广泛的 CI/CD 平台，网上流传着无数的脚本和攻略，在学习和开发的时候一定要从基本出发，了解内部原理，多看官方的文档，不要拿到一段代码就开始用，这样才能不会迷失在各式各样的脚本之中。

更重要的是要结合自己的业务需求，开发和定制属于自己的流程，不要被 Jenkins 的框架限制住。比如我们是否可以定义一个自己的 YAML 配置文件，然后根据 YAML 来生成 Pipeline，不需要业务自己写 Pipeline 脚本，规范使用，提前检查不合法的脚本，核心的模块共同升级，避免了一个流程小改动需要所有项目组同步更新。

7. 部署到远程服务器并运行 ◀ 上一节 下一节 ▶ 9. 定时构建的语法

✎ 我要提出意见反馈