



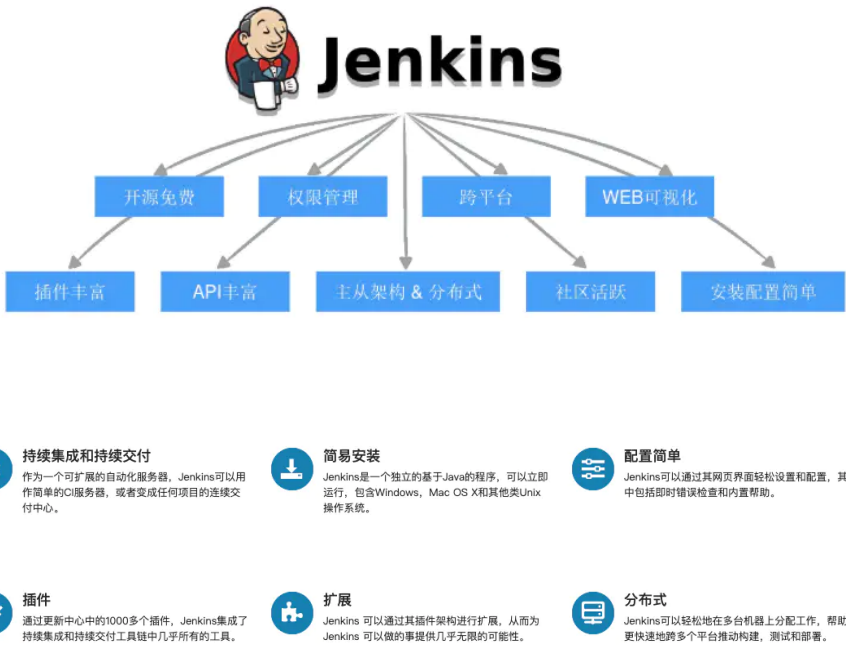
jenkins pipeline基础语法与示例



MR_Hanjc

 5 2019.03.24 23:34:50 字数 2,195 阅读 33,397

一、Jenkins介绍



二、Jenkins Pipeline介绍

Jenkins Pipeline总体介绍

- 1.Pipeline 是Jenkins 2.X核心特性，帮助Jenkins实现从CI到CD与DevOps的转变
- 2.Pipeline 简而言之，就是一套运行于Jenkins上的 workflow 框架，将原本独立运行于单个或者多个节点的任务连接起来，实现单个任务难以完成的复杂流程编排与可视化。

什么是Jenkins Pipeline

- 1.Jenkins Pipeline是一组插件，让Jenkins可以实现持续交付管道的落地和实施。
- 2.持续交付管道(CD Pipeline)是将软件从版本控制阶段到交付给用户或客户的完整过程的自动化表现。
- 3.软件的每一次更改（提交到源代码管理系统）都要经过一个复杂的过程才能被发布。

4. Pipeline提供了一组可扩展的工具，通过Pipeline Domain Specific Language (DSL) syntax可以达到Pipeline as Code的目的
5. Pipeline as Code: Jenkinsfile 存储在项目的源代码库

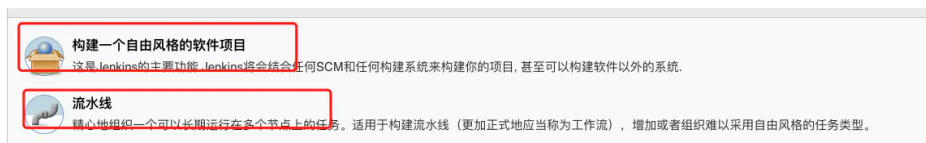
Why Pipeline?

本质上，Jenkins 是一个自动化引擎，它支持许多自动模式。 Pipeline向Jenkins中添加了一组强大的工具, 支持用例 简单的CI到全面的CD pipeline。通过对一系列的相关任务进行建模, 用户可以利用pipeline的很多特性:

- 代码：Pipeline以代码的形式实现，通常被检入源代码控制，使团队能够编辑，审查和迭代其CD流程。
- 可持续性：Jenkins重启或者中断后都不会影响Pipeline Job。

- 停顿：Pipeline可以选择停止并等待人工输入或批准，然后再继续Pipeline运行。
- 多功能：Pipeline支持现实世界的复杂CD要求，包括fork/join子进程，循环和并行执行工作的能力。
- 可扩展：Pipeline插件支持其DSL的自定义扩展以及与其他插件集成的多个选项。

Pipeline与freestyle区别



1.Job调度方式

pipeline: 通过结构化pipeline 语法进行调度，易于理解与阅读

freestyle: 通过jenkins api或者cli进行调度

2.Job显示形式

pipeline:提供上帝视角（全局视图）

freestyle: 没有视图

Jenkins Pipeline 基础语法

官网链接: <https://jenkins.io/doc/>

Pipeline 支持两种语法

1.声明式（jenkins2.5新加入的语法）

```
Jenkinsfile (Declarative Pipeline)
pipeline {
  agent any ❶
  stages {
    stage('Build') { ❷
      steps {
        // ❸
      }
    }
    stage('Test') { ❹
      steps {
        // ❺
      }
    }
    stage('Deploy') { ❻
      steps {
        // ❼
      }
    }
  }
}
```

特点:

- 1.最外层必须由pipeline{ //do something }来进行包裹
- 2.不需要分号作为分隔符，每个语句必须在一行内
- 3.不能直接使用groovy语句（例如循环判断等），需要被script {}包裹

2.脚本式

特点:

- 1.最外层有node{}包裹
- 2.可直接使用groovy语句

Declarative Pipeline（声明式）核心概念

核心概念用来组织pipeline的运行流程

- 1.pipeline :声明其内容为一个声明式的pipeline脚本
- 2.agent:执行节点（job运行的slave或者master节点）
- 3.stages:阶段集合，包裹所有的阶段（例如：打包，部署等各个阶段）
- 4.stage:阶段，被stages包裹，一个stages可以有多个stage
- 5.steps:步骤,为每个阶段的最小执行单元,被stage包裹
- 6.post:执行构建后的操作，根据构建结果来执行对应的操作

根据上面几个概念就能够轻易的创建一个简单的pipeline

```
1 pipeline{
2     agent any
3     stages{
4         stage("first stage"){
5             steps("first steps"){
6                 echo "this is first step"
7             }
8         }
9     }
10    post{
11        always{
12            echo "this is ending..."
13        }
14    }
15 }
```

下面针对几个核心概念，逐一进行说明

1.pipeline

作用域：应用于全局最外层，表明该脚本为声明式pipeline

是否必须：必须

参数：无

2.agent

作用域：可用在全局与stage内

是否必须：是，

参数：any,none, label, node,docker,dockerfile

```
1 pipeline{
2     agent any //全局必须带有agent表明此pipeline执行节点
3     stages{
4         stage("first stage"){
5             agent { label 'master' } //具体执行的步骤节点，非必须
6             steps{
7                 echo "this is first step"
8             }
9         }
10    }
11 }
```

参数示例：

```
1 //运行在任意的可用节点上
2 agent any
3 //全局不指定运行节点，由各自stage来决定
4 agent none
5 //运行在指定标签的机器上,具体标签名称由agent配置决定
6 agent { label 'master' }
7 //node参数可以扩展节点信息
8 agent {
9     node {
10         label 'master'
11         customWorkspace 'xxx'
12     }
13 }
14 //使用指定运行的容器
15 agent { docker 'python' }
```

3.stages

作用域：全局或者stage阶段内，每个作用域内只能使用一次
是否必须：全局必须
参数：无

```
1 pipeline{
2     agent any
3     stages{
4         stage("first stage"){
5             stages{ //嵌套在stage里
6                 stage("inside"){
7                     steps{
8                         echo "inside"
9                     }
10                }
11            }
12        }
13        stage("stage2"){
14            steps{
15                echo "outside"
16            }
17        }
18    }
19 }
```

看下运行结果,发现嵌套的stage也是能够展现在视图里面的



4.stage

作用域：被stages包裹，作用在自己的stage包裹范围内
是否必须：必须
参数：需要一个string参数，表示此阶段的工作内容
备注：stage内部可以嵌套stages，内部可单独制定运行的agent

5.steps

作用域：被stage包裹，作用在stage内部
是否必须：必须
参数：无

6.post

作用域：作用在pipeline结束后者stage结束后

条件：always、changed、failure、success、unstable、aborted

Declarative Pipeline（声明式）指令

指令是帮助pipeline更容易的执行命令，可以理解为一个封装好的公共函数和方法，提供给pipeline使用

1.environment：声明一个全局变量或者步骤内部的局部变量

```
1 pipeline{
2   agent any
3   environment {
4     P1="parameters 1"
5   }
6   stages{
7     stage("stage2"){
8       environment {
9         P2="parameters 2"
10      }
11      steps{
12        echo "$P1"
13        echo "$P2"
14      }
15    }
16  }
17 }
```

2.options:options指令能够提供给脚本更多的选项

- buildDiscarder:指定build history与console的保存数量
用法：options { buildDiscarder(logRotator(numToKeepStr: '1')) }
- disableConcurrentBuilds：设置job不能够同时运行
用法：options { disableConcurrentBuilds() }
- skipDefaultCheckout：跳过默认设置的代码check out
用法：options { skipDefaultCheckout() }
- skipStagesAfterUnstable:一旦构建状态变得UNSTABLE，跳过该阶段
用法：options { skipStagesAfterUnstable() }
- checkoutToSubdirectory:在工作空间的子目录进行check out
用法：options { checkoutToSubdirectory('children_path') }
- timeout:设置jenkins运行的超时时间，超过超时时间，job会自动被终止
用法：options { timeout(time: 1, unit: 'MINUTES') }
- retry :设置retry作用域范围的重试次数
用法：options { retry(3) }
- timestamps:为控制台输出增加时间戳
用法：options { timestamps() }

备注：当options作用在stage内部的时候，可选的只能是跟stage相关的选项（skipDefaultCheckout、timeout、retry、timestamps）

以其中几个作为例子

```
1 pipeline{
2   agent any
3   options {
4     timestamps()
5     disableConcurrentBuilds()
6   }
7   stages{
8     stage("stage1"){
9       options { timeout(time:1,unit:'MINUTES')
10        retry(2)
11      }
12      steps{
13        echo "beging======"
14        sh "xxx.sh"
15      }
16    }
17  }
18 }
```

3.parameters: 提供pipeline运行的参数

- 作用域: 被最外层pipeline所包裹, 并且只能出现一次, 参数可被全局使用
- 好处: 使用parameters好处是能够使参数也变成code,达到pipeline as code, pipeline中设置的参数会自动在job构建的时候生成, 形成参数化构建
- 用法:

```
1 pipeline{
2   agent any
3   parameters {
4     string(name: 'P1', defaultValue: 'it is p1', description: 'it is p1')
5     booleanParam(name: 'P2', defaultValue: true, description: 'it is p2')
6   }
7   stages{
8     stage("stage1"){
9       steps{
10         echo "$P1"
11         echo "$P2"
12       }
13     }
14   }
15 }
```

自动生成的构建参数

Pipeline test_9

需要如下参数用于构建项目:

P1
it is p1
P2 ☒
it is p2

开始构建

4.triggers:触发器是自动化运行pipeline的方法

- 作用域: 被pipeline包裹, 在符合条件下自动触发pipeline

目前包含三种自动触发的方式:

第一种: cron

- 作用: 以指定的时间来运行pipeline
- 用法: triggers { cron('*/*1 * * *') }

第二种: pollSCM

- 作用: 以固定的时间检查代码仓库更新 (或者当代码仓库有更新时) 自动触发pipeline构建
- 用法: triggers { pollSCM('H */4 * * 1-5') }或者triggers { pollSCM() } (后者需要配置post-commit/post-receive钩子)

第三种: upstream

- 作用: 可以利用上游Job的运行状态来进行触发
- 用法: triggers { upstream(upstreamProjects: 'job1,job2', threshold: hudson.model.Result.SUCCESS) }

```
1 pipeline{
2   agent any
3   //说明: 当test_8或者test_7运行成功的时候, 自动触发
4   triggers { upstream(upstreamProjects: 'test_8,test_7', threshold: hudson.model.Result.SUCCESS) }
5   stages{
6     stage("stage1"){
7       steps{
8         echo "hello"
9       }
10    }
11  }
12 }
```

5.tools:用于引用配置好的工具

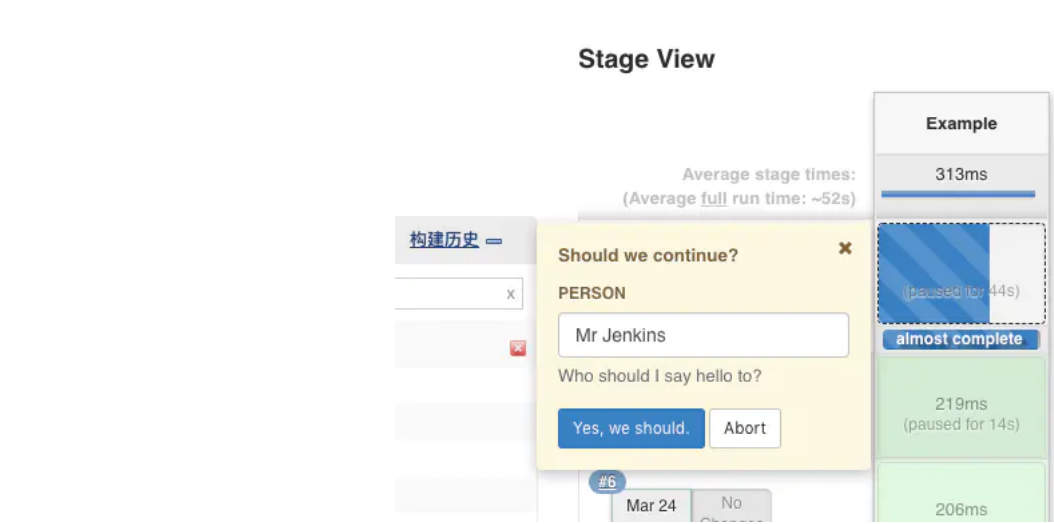
引用的工具需要在管理页面的全局工具配置里配置过

```
1 pipeline {
2   agent any
3   tools {
4     maven 'apache-maven-3.0.1'
5   }
6   stages {
7     stage('Example') {
8       steps {
9         sh 'mvn --version'
10      }
11    }
12  }
13 }
```

6.input:input指令允许暂时中断pipeline执行，等待用户输入，根据用户输入进行下一步动作

```
1 pipeline {
2   agent any
3   stages {
4     stage('Example') {
5       input {
6         message "Should we continue?"
7         ok "Yes, we should."
8         submitter "alice,bob"
9         parameters {
10          string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')
11        }
12      }
13      steps {
14        echo "Hello, ${PERSON}, nice to meet you."
15      }
16    }
17  }
18 }
```

看下效果



7.when: 根据when指令的判断结果来决定是否执行后面的阶段

可选条件

- branch：判断分支名称是否符合预期
用法: when { branch 'master' }
- environment：判断环境变量是否符合预期
用法: when { environment name: 'DEPLOY_TO', value: 'production' }
- expression: 判断表达式是否符合预期
用法: when { expression { return params.DEBUG_BUILD } }
- not: 判断条件是否为假
用法: when { not { branch 'master' } }

- allOf: 判断所有条件是不是都为真
用法: when { allOf { branch 'master'; environment name: 'DEPLOY_TO'; value: 'production' } }
- anyOf: 判断是否有一个条件为真
用法: when { anyOf { branch 'master'; branch 'staging' } }

特别的: 如果我们想要在进入agent之前进行判断, 需要将beforeAgent设置为true

```
1 pipeline {
2     agent none
3     stages {
4         stage('Example Build') {
5             steps {
6                 echo 'Hello World'
7             }
8         }
9         stage('Example Deploy') {
10            agent {
11                label "some-label"
12            }
13            when {
14                beforeAgent true //设置先对条件进行判断, 符合预期才进入steps
15                branch 'production'
16            }
17            steps {
18                echo 'Deploying'
19            }
20        }
21    }
22 }
```

并行执行

通过将阶段设置为parallel来表明该stage为并行运行, 但是需要注意以下几点

- 一个stage只能有一个steps或者parallel
- 嵌套的stages里不能使用parallel
- parallel不能包含agent或者tools
- 通过设置failFast 为true表示: 并行的job中如果其中的一个失败, 则终止其他并行的stage

```
1 pipeline {
2     agent any
3     stages {
4         stage('Non-Parallel Stage') {
5             steps {
6                 echo 'Non-parallel'
7             }
8         }
9         stage('Parallel Stage') {
10            agent any
11            failFast true
12            parallel {
13                stage('parallel 1') {
14                    agent any
15                    steps {
16                        echo "parallel 1"
17                    }
18                }
19                stage('parallel 2') {
20                    steps {
21                        echo "parallel 2"
22                    }
23                }
24            }
25        }
26    }
27 }
```

脚本

在声明式的pipeline中默认无法使用脚本语法, 但是pipeline提供了一个脚本环境入口: script(),通过使用script来包裹脚本语句, 即可使用脚本语法

- 条件判断:


```
1 pipeline {
2   agent any
3   stages {
4     stage('stage 1') {
5       steps {
6         script{
7           if ( "1" == "1" ) {
8             echo "lalala"
9           }else {
10            echo "oooo"
11          }
12        }
13      }
14    }
15  }
16 }
```

- 异常处理

```
1 pipeline {
2   agent any
3   stages {
4     stage('stage 1') {
5       steps {
6         script{
7           try {
8             sh 'exit 1'
9           }
10          catch (exc) {
11            echo 'Something failed'
12          }
13        }
14      }
15    }
16  }
17 }
18 }
```