# Jenkins2 Pipeline Docs

Version 0.0.1

☁️☀️

* [Home](https://zeyangli.github.io/)

# 5-2 Jenkins 声明式与脚本式语法

## 1. 声明式Pipeline

声明式Pipleine是最近添加到Jenkins流水线的，它在流水线子系统之上提供了一种更简单，更有主见的语法。 所有的声明式Pipeline都必须包含一个 pipeline块中，比如：

pipeline {

//run}

在声明式Pipeline中的基本语句和表达式遵循Groovy的语法。但是有以下例外：

* 流水线顶层必须是一个块，特别是pipelin{}。
* 不需要分号作为分割符，是按照行分割的。
* 语句块只能由阶段、指令、步骤、赋值语句组成。例如: input被视为input()。

### 1.1 agent(代理)

agent 指定了流水线的执行节点。

参数：

* any 在任何可用的节点上执行pipeline。
* none 没有指定agent的时候默认。
* label 在指定标签上的节点上运行Pipeline。

node 允许额外的选项。

这两种是一样的agent { node { label 'labelname' }}aget { label ' labelname '}

### 1.2 post

定义一个或多个steps ，这些阶段根据流水线或阶段的完成情况而 运行(取决于流水线中post部分的位置). post 支持以下 post-condition 块中的其中之一: always, changed, failure, success, unstable, 和 aborted。这些条件块允许在 post 部分的步骤的执行取决于流水线或阶段的完成状态。

* always 无论流水线或者阶段的完成状态。
* changed 只有当流水线或者阶段完成状态与之前不同时。
* failure 只有当流水线或者阶段状态为”failure”运行。
* success 只有当流水线或者阶段状态为”success”运行。
* unstable 只有当流水线或者阶段状态为”unstable”运行。例如：测试失败。

aborted 只有当流水线或者阶段状态为”aborted “运行。例如：手动取消。

pipeline {agent anystages {

stage('Example') {

steps {

echo 'Hello World'

}

}}post {

always {

echo 'I will always say Hello again!'

}}}

### 1.3 stages(阶段)

包含一系列一个或多个 stage 指令, 建议 stages 至少包含一个 stage 指令用于连续交付过程的每个离散部分,比如构建, 测试, 和部署。

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}}

### 1.4 steps(步骤)

step是每个阶段中要执行的每个步骤。

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}}

### 1.5 指令

#### 1.5.1 environment

environment 指令指定一个键值对序列，该序列将被定义为所有步骤的环境变量，或者是特定于阶段的步骤，这取决于 environment 指令在流水线内的位置。

该指令支持一个特殊的方法 credentials() ，该方法可用于在Jenkins环境中通过标识符访问预定义的凭证。对于类型为 “Secret Text”的凭证, credentials() 将确保指定的环境变量包含秘密文本内容。对于类型为 “SStandard username and password”的凭证, 指定的环境变量指定为 username:password ，并且两个额外的环境变量将被自动定义 :分别为 MYVARNAME\_USR 和 MYVARNAME\_PSW 。

pipeline {

agent any

environment {

CC = 'clang'

}

stages {

stage('Example') {

environment {

AN\_ACCESS\_KEY = credentials('my-prefined-secret-text')

}

steps {

sh 'printenv'

}

}

}}

#### 1.5.2 options

options 指令允许从流水线内部配置特定于流水线的选项。 流水线提供了许多这样的选项, 比如buildDiscarder,但也可以由插件提供, 比如 timestamps。

* buildDiscarder: 为最近的流水线运行的特定数量保存组件和控制台输出。
* disableConcurrentBuilds: 不允许同时执行流水线。 可被用来防止同时访问共享资源等。
* overrideIndexTriggers: 允许覆盖分支索引触发器的默认处理。
* skipDefaultCheckout: 在agent 指令中，跳过从源代码控制中检出代码的默认情况。
* skipStagesAfterUnstable: 一旦构建状态变得UNSTABLE，跳过该阶段。
* checkoutToSubdirectory: 在工作空间的子目录中自动地执行源代码控制检出。
* timeout: 设置流水线运行的超时时间, 在此之后，Jenkins将中止流水线。
* retry: 在失败时, 重新尝试整个流水线的指定次数。

timestamps 预测所有由流水线生成的控制台输出，与该流水线发出的时间一致。

//指定一个小时的全局执行超时, 在此之后，Jenkins将中止流水线运行。pipeline {agent anyoptions {

timeout(time: 1, unit: 'HOURS') }stages {

stage('Example') {

steps {

echo 'Hello World'

}

}}}

#### 1.5.3 参数

为流水线运行时设置项目相关的参数

string 字符串类型的参数, 例如:

parameters { string(name: 'DEPLOY\_ENV', defaultValue: 'staging', description: '') }

booleanParam 布尔参数, 例如:

parameters { booleanParam(name: 'DEBUG\_BUILD', defaultValue: true, description: '') }

示例

pipeline {agent anyparameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')}stages {

stage('Example') {

steps {

echo "Hello ${params.PERSON}"

}

}}}

#### 1.5.4 触发器

构建触发器

cron 计划任务定期执行构建。

triggers { cron('H \*/4 \* \* 1-5') }

pollSCM 与cron定义类似，但是由jenkins定期检测源码变化。

triggers { pollSCM('H \*/4 \* \* 1-5') }

upstream 接受逗号分隔的工作字符串和阈值。 当字符串中的任何作业以最小阈值结束时，流水线被重新触发。

triggers { upstream(upstreamProjects: 'job1,job2', threshold: hudson.model.Result.SUCCESS) }

示例

pipeline {agent anytriggers {

cron('H \*/4 \* \* 1-5')}stages {

stage('Example') {

steps {

echo 'Hello World'

}

}}}

#### 1.5.5 tool

获取通过自动安装或手动放置工具的环境变量。支持maven/jdk/gradle。工具的名称必须在系统设置->全局工具配置中定义。

示例:

pipeline {

agent any

tools {

maven 'apache-maven-3.0.1'

}

stages {

stage('Example') {

steps {

sh 'mvn --version'

}

}

}}

#### 1.5.6 input

input用户在执行各个阶段的时候，由人工确认是否继续进行。

* message 呈现给用户的提示信息。
* id 可选，默认为stage名称。
* ok 默认表单上的ok文本。
* submitter 可选的,以逗号分隔的用户列表或允许提交的外部组名。默认允许任何用户。
* submitterParameter 环境变量的可选名称。如果存在，用submitter 名称设置。
* parameters 提示提交者提供的一个可选的参数列表。

示例：

pipeline {

agent any

stages {

stage('Example') {

input {

message "Should we continue?"

ok "Yes, we should."

submitter "alice,bob"

parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

}

}

steps {

echo "Hello, ${PERSON}, nice to meet you."

}

}

}}

#### 1.5.7 when

when 指令允许流水线根据给定的条件决定是否应该执行阶段。 when 指令必须包含至少一个条件。 如果when 指令包含多个条件, 所有的子条件必须返回True，阶段才能执行。 这与子条件在 allOf 条件下嵌套的情况相同。

内置条件

branch: 当正在构建的分支与模式给定的分支匹配时，执行这个阶段,这只适用于多分支流水线例如:

when { branch 'master' }

environment: 当指定的环境变量是给定的值时，执行这个步骤,例如:

when { environment name: 'DEPLOY\_TO', value: 'production' }

expression 当指定的Groovy表达式评估为true时，执行这个阶段, 例如:

when { expression { return params.DEBUG\_BUILD } }

not 当嵌套条件是错误时，执行这个阶段,必须包含一个条件，例如:

when { not { branch 'master' } }

allOf 当所有的嵌套条件都正确时，执行这个阶段,必须包含至少一个条件，例如:

when { allOf { branch 'master'; environment name: 'DEPLOY\_TO', value: 'production' } }

anyOf 当至少有一个嵌套条件为真时，执行这个阶段,必须包含至少一个条件，例如:

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

示例：

pipeline {

agent any

stages {

stage('Example Build') {

steps {

echo 'Hello World'

}

}

stage('Example Deploy') {

when {

branch 'production'

}

steps {

echo 'Deploying'

}

}

}}

pipeline {

agent any

stages {

stage('Example Build') {

steps {

echo 'Hello World'

}

}

stage('Example Deploy') {

when {

branch 'production'

environment name: 'DEPLOY\_TO', value: 'production'

}

steps {

echo 'Deploying'

}

}

}}

pipeline {

agent any

stages {

stage('Example Build') {

steps {

echo 'Hello World'

}

}

stage('Example Deploy') {

when {

allOf {

branch 'production'

environment name: 'DEPLOY\_TO', value: 'production'

}

}

steps {

echo 'Deploying'

}

}

}}

pipeline {

agent any

stages {

stage('Example Build') {

steps {

echo 'Hello World'

}

}

stage('Example Deploy') {

when {

branch 'production'

anyOf {

environment name: 'DEPLOY\_TO', value: 'production'

environment name: 'DEPLOY\_TO', value: 'staging'

}

}

steps {

echo 'Deploying'

}

}

}}

pipeline {

agent any

stages {

stage('Example Build') {

steps {

echo 'Hello World'

}

}

stage('Example Deploy') {

when {

expression { BRANCH\_NAME ==~ /(production|staging)/ }

anyOf {

environment name: 'DEPLOY\_TO', value: 'production'

environment name: 'DEPLOY\_TO', value: 'staging'

}

}

steps {

echo 'Deploying'

}

}

}}

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'

}

}

}}

#### 1.5.8 并行

声明式流水线的阶段可以在他们内部声明多隔嵌套阶段, 它们将并行执行。 注意，一个阶段必须只有一个 steps 或 parallel的阶段。 嵌套阶段本身不能包含 进一步的 parallel 阶段, 但是其他的阶段的行为与任何其他 stageparallel的阶段不能包含 agent 或 tools阶段, 因为他们没有相关 steps。

另外, 通过添加 failFast true 到包含parallel的 stage中， 当其中一个进程失败时，你可以强制所有的 parallel 阶段都被终止。

示例:

pipeline {

agent any

stages {

stage('Non-Parallel Stage') {

steps {

echo 'This stage will be executed first.'

}

}

stage('Parallel Stage') {

when {

branch 'master'

}

failFast true

parallel {

stage('Branch A') {

agent {

label "for-branch-a"

}

steps {

echo "On Branch A"

}

}

stage('Branch B') {

agent {

label "for-branch-b"

}

steps {

echo "On Branch B"

}

}

}

}

}}

### 1.6 step步骤

#### 1.6.1 script

script 步骤需要 [scripted-pipeline]块并在声明式流水线中执行。对于大多数用例来说,应该声明式流水线中的“脚本”步骤是不必要的，但是它可以提供一个有用的”逃生出口”。非平凡的规模和/或复杂性的script块应该被转移到 共享库 。

示例：

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

script {

def browsers = ['chrome', 'firefox']

for (int i = 0; i < browsers.size(); ++i) {

echo "Testing the ${browsers[i]} browser"

}

}

}

}

}}

## 2. 脚本化Pipeline

脚本化流水线, 与声明式一样的是, 是建立在底层流水线的子系统上的。与声明式不同的是, 脚本化流水线实际上是由 Groovy构建的通用 DSL 。 Groovy 语言提供的大部分功能都可以用于脚本化流水线的用户。这意味着它是一个非常有表现力和灵活的工具，可以通过它编写持续交付流水线。

### 2.1 流程控制

脚本化流水线从Jenkinsfile的顶部开始向下串行执行, 就像 Groovy 或其他语言中的大多数传统脚本一样。 因此，提供流控制取决于 Groovy 表达式, 比如 if/else 条件, 例如:

node {

stage('Example') {

if (env.BRANCH\_NAME == 'master') {

echo 'I only execute on the master branch'

} else {

echo 'I execute elsewhere'

}

}}

另一种方法是使用Groovy的异常处理支持来管理脚本化流水线流控制。当 步骤 失败 ，无论什么原因，它们都会抛出一个异常。处理错误的行为必须使用Groovy中的 try/catch/finally 块 , 例如:

node {

stage('Example') {

try {

sh 'exit 1'

}

catch (exc) {

echo 'Something failed, I should sound the klaxons!'

throw

}

}}