**Day2-06月12日-Jenkins Pipeline As Code**

|  |  |  |
| --- | --- | --- |
| **时间** | **标题** | **内容** |
| **上午** | **Pipeline 基础语法** | 1. 什么是Pipeline ？ Pipeline的定义 2. 如何使用Jenkinsfile描述Pipeline？ 3. 声明式管道与脚本式管道 4. Pipeline语法-Pipeline/Agent/Stages/Stage/Post 5. environment/options/parameters/triggers 6. tools/input/when/script |
| 1. Jenkins Pipeline开发技巧 2. 编写一条测试的Pipeline 3. Pipeline 问题排错思路 4. Jenkinsfile的常见管理方式 |
|
|
|

=

|  |  |  |
| --- | --- | --- |
| **下午** | **Groovy编程** | 1. 本地配置Groovy开发环境 2. Groovy数据类型 3. Groovy中的函数定义 4. Groovy中的常用方法 |
| 1. Jenkins ShareLibrary 共享库简介 2. 配置Jenkins ShareLibrary使用 3. 熟悉Pipeline 中常用的DSL方法 |

课程素材： [📎day2.zip](https://www.yuque.com/attachments/yuque/0/2021/zip/2584012/1623675485000-f32f933c-8a5d-4019-9fc7-055c750355ff.zip)

**1. 什么是Pipeline ？**

**Pipeline**

Jenkins的**核心**是Pipeline（流水线项目），实现了**Pipeline As Code**。即我们将构建部署测试等步骤全部以代码的形式写到**Jenkinsfile**中。Jenkins在运行Pipeline任务的时候会按照Jenkinsfile中定义的代码顺序执行。写Jenkinsfile是一项很重的工作，如果稍不注意很容易造成Jenkins的流水线任务失败。Jenkinsfile类似于Dockerfile，具有一套特定的语法。

**Stage**

在Jenkins pipeline中，**一条流水线是由多个阶段组成的**，每个阶段一个stage。例如：构建、测试、部署等等。

**Agent**

Jenkins采用**分布式架构**，分为**server节点和agent节点**。server节点也是可以运行构建任务的，但我们一般使其主要来做任务的调度。（毕竟server节点挂了就都...）**agent节点专门用于任务的执行**。随着现在容器的盛行，我们可以将server节点和agent节点在容器或者基于Kubernetes中部署。关于agent节点借助容器可以实现动态的资源分配等等好处。agent节点可以分为静态节点和动态节点。静态节点是固定的一台vm虚机或者容器。动态节点是随着任务的构建来自动创建agent节点。

我们可以根据地铁线路图来理解pipeline ：

* pipeline 等同于 13号线
* agent 等同于 13号线🚇
* stages 等同于 13号线完整的站点路线
* stage 等同于 13号线中的一个站点





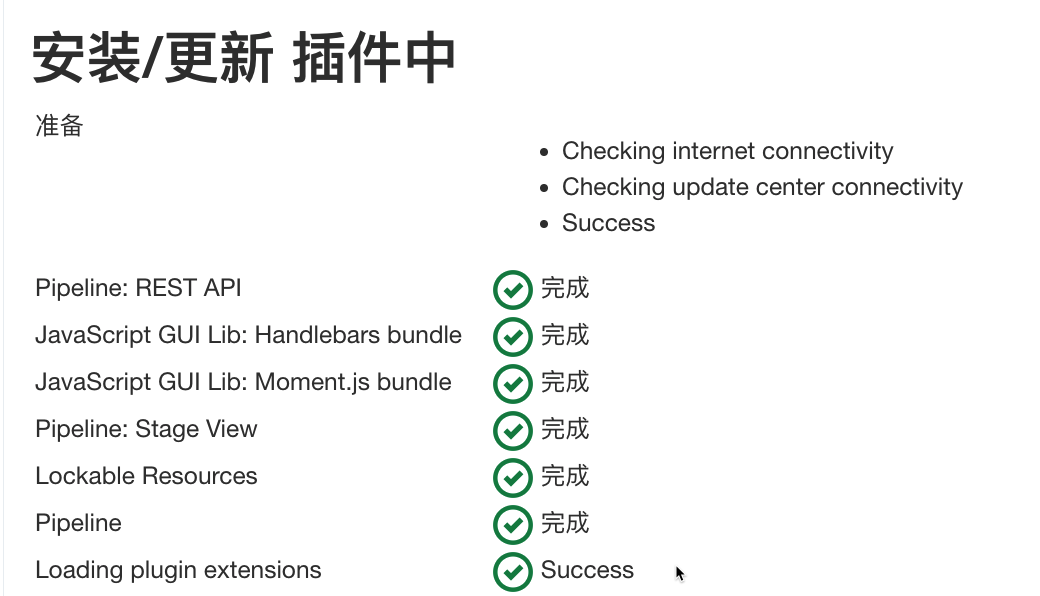
**Pipeline作业**

**安装Pipeline插件**

在创建Pipeline类型的作业的时候，需要提前安装好pipeline插件，不然可能会出现**找不到pipeline类型的**作业。

进入插件管理， 搜索关键字"pipeline" 。安装后重启一下。





当Jenkins重启成功后，我们创建一个流水线类型的作业。



点击创建，会进入作业的配置页面。此时我们可以对比一下pipeline类型的项目和自由风格的项目的区别。

**与自由风格项目对比**

**流水线类型作业 1**



**自由风格类型作业 0**



总结： General和构建触发器都是相同的， 自由风格的作业需要在**UI页面配置详细的构建过程和构建后操作**，而流水线类型的作业是直接将构建过程和操作放到了"流水线"中配置，**以代码的方式描述流水线**。

**流水线回放功能**

借助回放可以修改上次构建所使用的Jenkinsfile代码， 进行更改后可以立即运行进行调试。



**Jenkinsfile**

Jenkinsfile的是实现Pipeline as Code的核心功能。 该文件用于描述流水线的过程。以下是一个简单的实例：

pipeline{

//指定运行此流水线的节点

agent { node { label "build"}}

//管道运行选项

options {

skipStagesAfterUnstable()

}

//流水线的阶段

stages{

//阶段1 获取代码

stage("CheckOut"){

steps{

script{

println("获取代码")

}

}

}

stage("Build"){

steps{

script{

println("运行构建")

}

}

}

}

post {

always{

script{

println("流水线结束后，经常做的事情")

}

}

success{

script{

println("流水线成功后，要做的事情")

}

}

failure{

script{

println("流水线失败后，要做的事情")

}

}

aborted{

script{

println("流水线取消后，要做的事情")

}

}

}

}

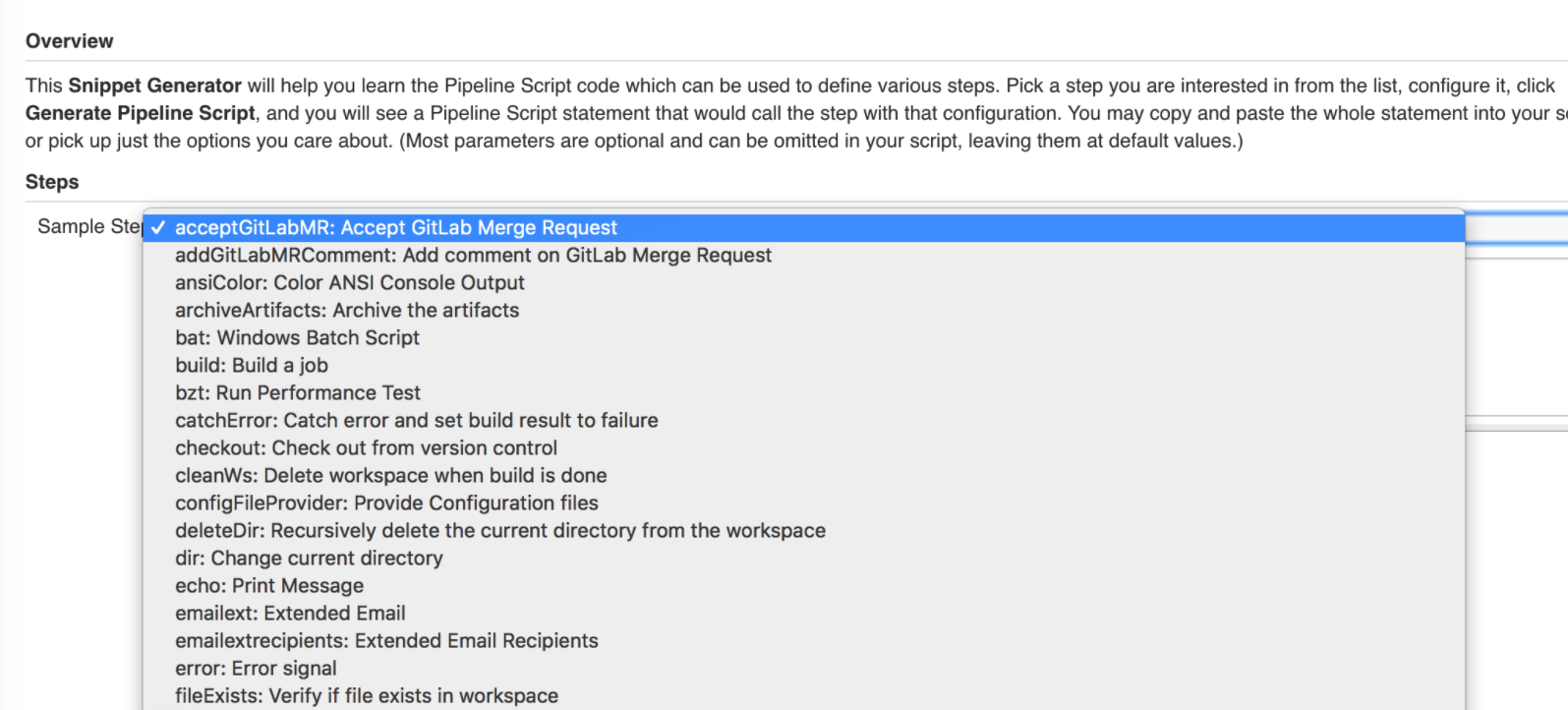
* 使用agent{}，指定流水线要运行的节点（可以使用名称或者标签）
* 指定options{} 定义流水线运行时的一些选项
* 指定stages{}（stages包含多个stage，stage包含steps。是流水线的每个步骤)
* 指定post{}（定义好此流水线运行成功或者失败后，根据状态做一些任务）

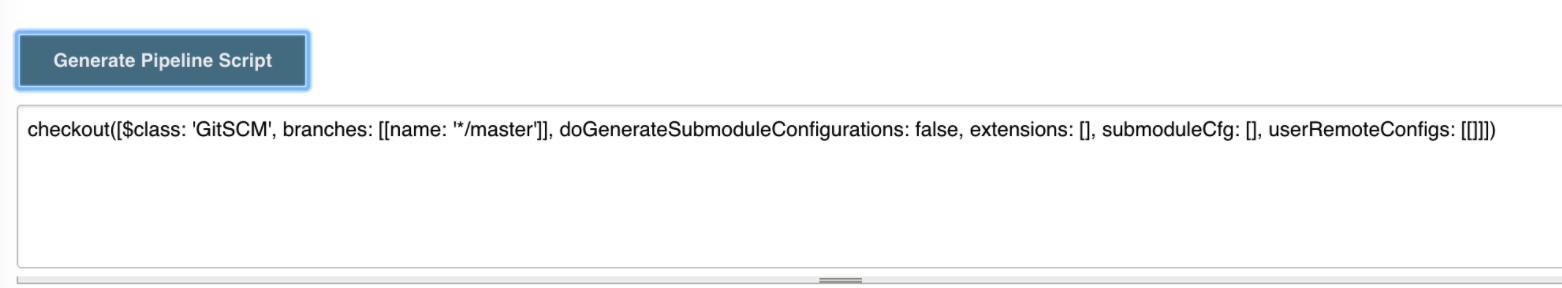
**2. Pipeline 开发工具**

**选择任意pipeline类型的作业，点击“流水线语法”即可进入pipeline开发工具页面。**

**片段生成器**

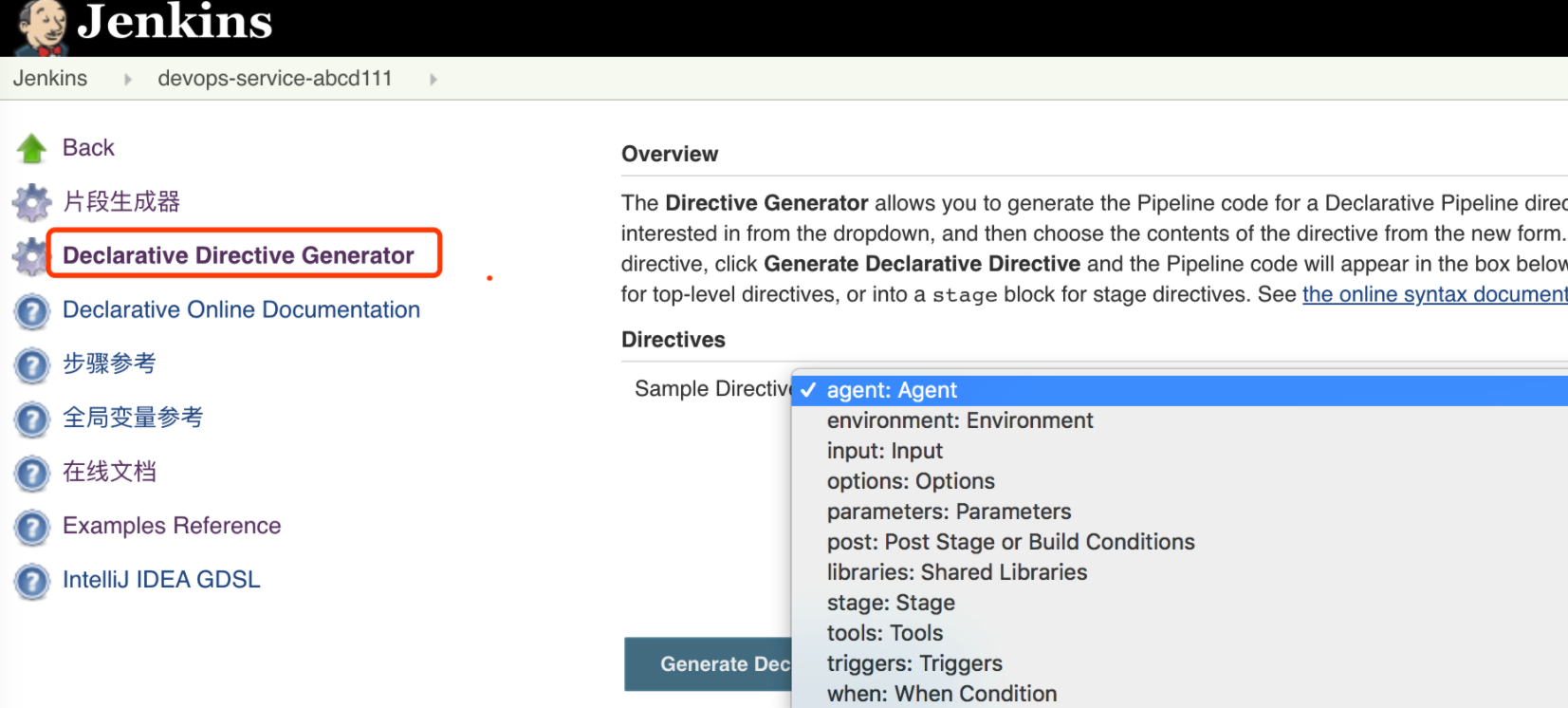
流水线代码片段生成器， 非常好用。在这里可以找到**每个插件以及Jenkins内置的方法**的使用方法。使用片段生成器可以根据个人需要生成方法，有些方法来源于插件，则需要先**安装相关的插件才能使用**哦。





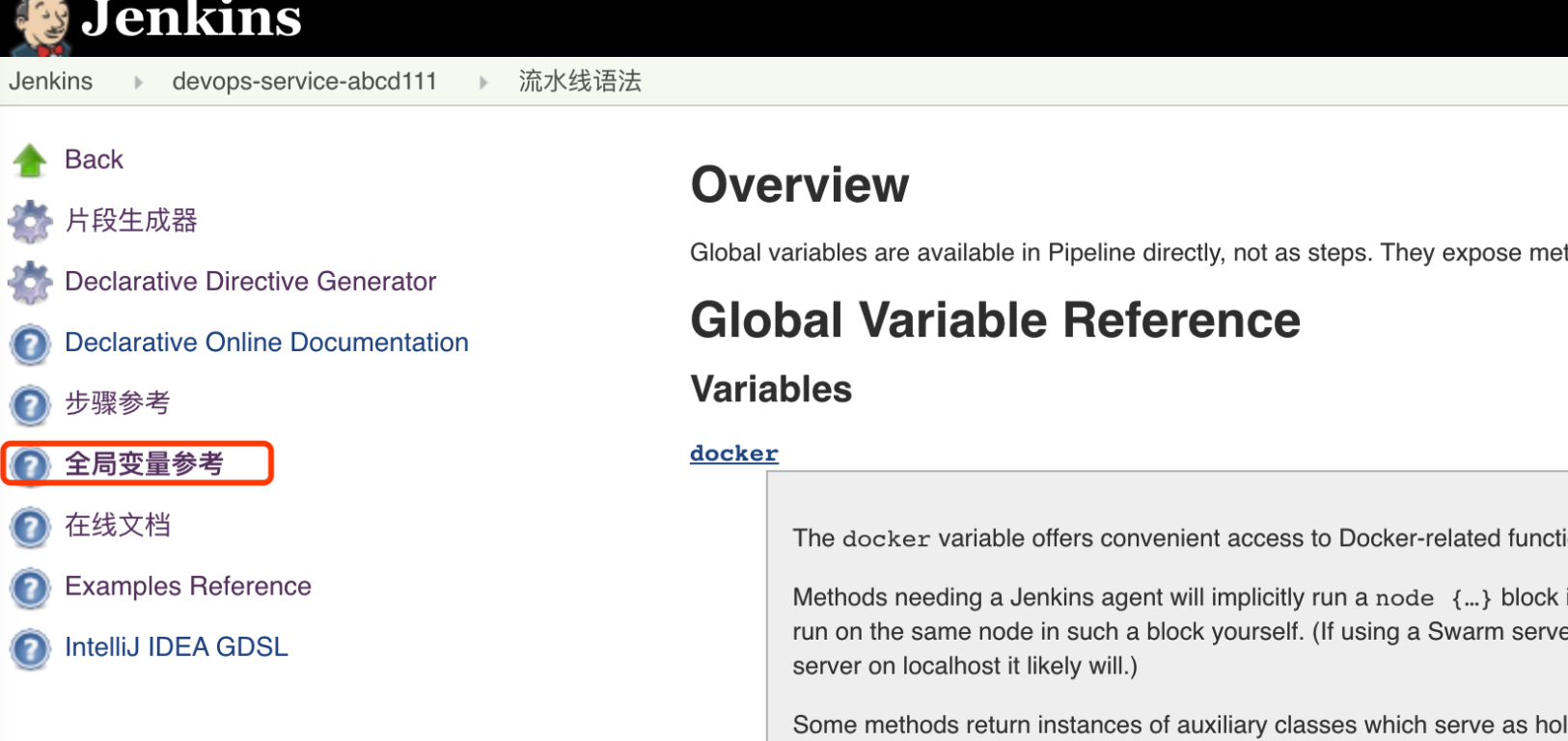
**声明式语法生成器**

可以生成声明式流水线语法的语句块。



**全局变量参考**

这些是已经安装的Jenkins插件和Jenkins内置的全局变量清单。



**3. Pipeline的核心语法**

声明式流水线的定义， 一个pipeline{}。

pipeline {

//pipeline

}

**agent 构建节点**

参数：

* any: 运行在任一可用节点。
* none：当pipeline全局指定agent为none，则根据每个stage中定义的agent运行（stage必须指定）。
* label：在指定的标签的节点运行。（标签=分组）
* node：支持自定义流水线的工作目录。

## 一

pipeline {

agent any

}

## 二

pipeline {

agent { label "label Name" }

}

## 三 自定义节点

pipeline {

agent {

node {

label "labelName",

customWorkspace "/opt/agent/workspace"

}

}

}

**stages构建阶段**

* 关系： stages > stage > steps > script
* 定义：
  + stages：包含多个stage阶段
  + stage：包含多个steps步骤
  + steps: 包含一组特定的脚本（加上**script后就可以实现在声明式脚本中嵌入脚本式语法**了）

pipeline {

agent { label "build" }

stages {

stage("build") {

steps {

echo "hello"

}

}

}

}

扩展： 在阶段中定义agent

## 在阶段中定义agent

pipeline {

agent none

stages{

stage('Build'){

agent { label "build" }

steps {

echo "building......"

}

}

}

}

**post 构建后操作**

* 定义： 根据流水线的最终状态匹配后做一些操作。
* 状态：
  + always： 不管什么状态总是执行
  + success： 仅流水线成功后执行
  + failure： 仅流水线失败后执行
  + aborted： 仅流水线被取消后执行
  + unstable：不稳定状态，单侧失败等等

pipeline {

.....

.....

post {

always{

script{

println("流水线结束后，经常做的事情")

}

}

success{

script{

println("流水线成功后，要做的事情")

}

}

failure{

script{

println("流水线失败后，要做的事情")

}

}

aborted{

script{

println("流水线取消后，要做的事情")

}

}

}

}

**env 构建时变量**

* 定义： 通过键值对（k-v）格式定义流水线在运行时的环境变量， 分为流水线级别和阶段级别。

流水线级别环境变量参考

pipeline {

environment {

NAME = "zeyang"

VERSION = "1.1.10"

ENVTYPE = "DEV"

}

}

阶段级别环境变量参考

pipeline {

...

...

stages {

stage("build"){

environment {

VERSION = "1.1.20"

}

steps {

script {

echo "${VERSION}"

}

}

}

}

}

**options 运行时选项**

## 设置保存最近的记录

options { buildDiscarder(logRotator(numToKeepStr: '1')) }

## 禁止并行构建

options { disableConcurrentBuilds() }

## 跳过默认的代码检出

options { skipDefaultCheckout() }

## 设定流水线的超时时间(可用于阶段级别)

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

## 设定流水线的重试次数(可用于阶段级别)

options { retry(3) }

## 设置日志时间输出(可用于阶段级别)

options { timestamps() }

参考：

pipeline {

options {

disableConcurrentBuilds()

skipDefaultCheckout()

timeout(time: 1, unit: 'HOURS')

}

stages {

stage("build"){

options {

timeout(time: 5, unit: 'MINUTES')

retry(3)

timestamps()

}

}

}

}

}

**FAQ: timestamps 报错， 需要安装插件 Timestamper 。**

WorkflowScript: 21: Invalid option type "timestamps". Valid option types: [authorizationMatrix, buildDiscarder, catchError, checkoutToSubdirectory, disableConcurrentBuilds, disableResume, durabilityHint, lock, overrideIndexTriggers, parallelsAlwaysFailFast, preserveStashes, quietPeriod, rateLimitBuilds, retry, script, skipDefaultCheckout, skipStagesAfterUnstable, timeout, waitUntil, warnError, withChecks, withContext, withCredentials, withEnv, wrap, ws] @ line 21, column 3.

timestamps()

^



**parameters 流水线参数**

* 定义： 流水线在运行时设置的参数，UI页面的参数。所有的参数都存储在params对象中。
* 将web ui页面中定义的参数，以代码的方式定义。

pipeline {

agent any

parameters {

string(name: 'VERSION', defaultValue: '1.1.1', description: '')

}

stages {

stage("Build"){

steps {

echo "${params.VERSION}"

}

}

}

}

**FAQ： 没有找到相关的环境变量， 这个是我们在parameters中引用了流水线中的变量导致的，可能因为加载顺序不同导致的，解决方法是可以在pipeline{} 外部定义变量进行引用。**

roovy.lang.MissingPropertyException: No such property: DEPLOY\_DESC for class: groovy.lang.Binding

at groovy.lang.Binding.getVariable(Binding.java:63)

**triggers 触发器**

* 流水线的触发方式
  + cron 定时触发: triggers { cron('H \*/7 \* \* 1-5') }
  + pollSCM: triggers { pollSCM('H \*/7 \* \* 1-5') }

## upstream

triggers {

upstream(upstreamProjects: 'job1,job2',

threshold: hudson.model.Result.SUCCESS)

}

demo：

pipeline {

agent any

triggers {

cron('H \*/7 \* \* 1-5')

}

stages {

stage('build') {

steps {

echo 'Hello World'

}

}

}

}

**input 流水线交互**

参数解析

* message: 提示信息
* ok: 表单中确认按钮的文本
* submitter: 提交人，默认所有人可以
* parameters： 交互时用户选择的参数

pipeline {

agent any

stages {

stage('Deploy') {

input {

message "是否继续发布"

ok "Yes"

submitter "zeyang,aa"

parameters {

string(name: 'ENVTYPE', defaultValue: 'DEV', description: 'env type..[DEV/STAG/PROD]')

}

}

steps {

echo "Deploy to ${ENVTYPE}, doing......."

}

}

}

}

**when 阶段运行控制**

判断条件

* 根据环境变量判断
* 根据表达式判断
* 根据条件判断（not/allOf/anyOf）

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'build......'

}

}

stage('Deploy') {

when {

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

}

steps {

echo 'Deploying.......'

}

}

}

}

### allOf 条件全部成立

when {

allOf {

environment name: 'CAN\_DEPLOY', value: 'true'

environment name: 'DEPLOY\_ENV', value: 'dev'

}

}

### anyOf 条件其中一个成立

when {

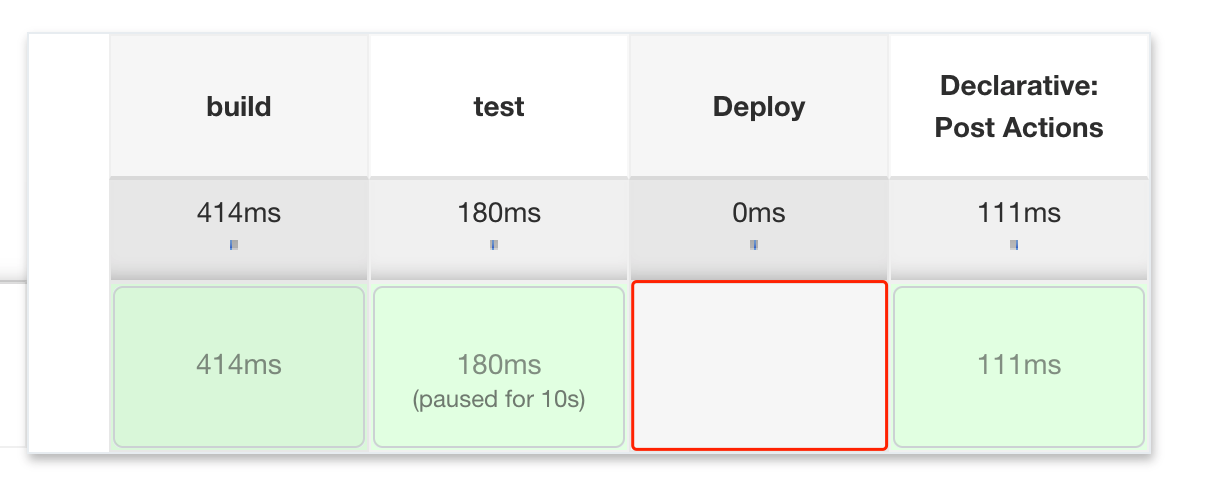
anyOf {

environment name: 'CAN\_DEPLOY', value: 'true'

environment name: 'DEPLOY\_ENV', value: 'dev'

}

}



**parallel 阶段并行**

场景： 自动化测试，多主机并行发布。

pipeline {

agent any

stages {

stage('Parallel Stage') {

failFast true

parallel {

stage('windows') {

agent {

label "master"

}

steps {

echo "windows"

}

}

stage('linux') {

agent {

label "build"

}

steps {

echo "linux"

}

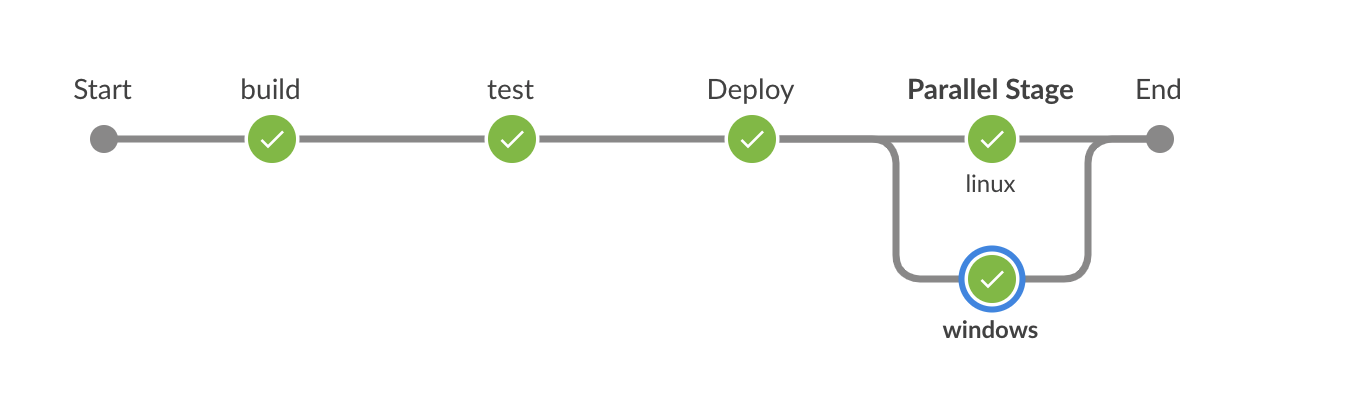
}

}

}

}

}



**FAQ:如何解决并发构建中的workspace问题？**

env.nworkspace = "/opt/agent/test/${JOB\_NAME}-${UUID.randomUUID().toString()}"

pipeline{

agent{

node{

label "build"

customWorkspace "${env.nworkspace}"

}

}

stages{

stage("build"){

steps{

echo "${env.nworkspace}"

}

}

}

}

### 输出

demo-fec54ca7-81a5-452e-91b5-2a187ab3562b



**4.流水线全局变量参考**

**内置变量**

BUILD\_NUMBER //构建号

BUILD\_ID //构建号

BUILD\_DISPLAY\_NAME //构建显示名称

JOB\_NAME //项目名称

EXECUTOR\_NUMBER //执行器数量

NODE\_NAME //构建节点名称

WORKSPACE //工作目录

JENKINS\_HOME //Jenkins home

JENKINS\_URL //Jenkins地址

BUILD\_URL //构建地址

JOB\_URL //项目地址

println(env)

env.branchName = "develop"

env.commitID = "${UUID.randomUUID().toString()}"

env.commitID = "${env.commitID.split("-")[0]}"

currentBuild.displayName = "#${env.branchName}-${env.commitID}"

currentBuild.description = "Trigger by user jenkins \n branch: master"

pipeline {

agent { label "build"}

stages{

stage("test"){

steps{

script{

echo "${BUILD\_NUMBER}"

echo "${BUILD\_ID}"

//currentBuild.displayName = "#${env.branchName}-${env.commitID}"

//currentBuild.description = "Trigger by user jenkins \n branch: master"

echo "当前下载代码分支为： ${env.branchName}"

}

}

}

}

}

currentbuild变量

result currentResult //构建结果

displayName //构建名称 #111

description //构建描述

duration //持续时间



**流水线中变量定义引用**

变量的类型：两种类型的变量。  
1.Jenkins系统内置变量 （全局变量）  
2.Pipeline中定义变量（全局/局部变量）  
  
  
Jenkins系统内置变量：是Jenkins系统在安装部署后预先定义好的变量。这些变量可以通过Jenkins流水线语法页面看到具体有哪些。这些变量都是全局的可以使用"**${env.变量名}"**引用。  
  
  
Pipeline中的变量：  
  
首先你要先理解pipeline可以用groovy语法来编写，而groovy是一门编程语言。所有的编程语言也都有各自的变量定义方式。 这就容易让大家产生疑惑的地方，pipeline中可以有很多种写法。  
  
def name = "devops"  
String name = "devops"  
  
以上这两种写法是一样，def可以自动推导出变量类型，而String这种写法是精确这个变量是一个字符串类型的。  
  
如果你在Jenkins图形界面设置了参数化构建，那么这些参数也都变成了Jenkins全局变量，可以使用与Jenkins内置变量相同的引用方式。  
  
  
如果在某个stage定义的变量默认是局部变量，在后续的stage中可能语法引用，所以如果需要引用最好定义为全局变量。  
  
全局变量的定义方式：  
env.name = "devops"  
  
引用方式： "${env.name}"  
  
  
总之变量是我们在编写Jenkins流水线是经常用到的。无处不在。如果你要定义全局的变量就用env.变量名的方式定义。 变量也不仅仅只有这几种写法，我建议如果想了解更多，可以多看下groovy这门需要的更多语法。

**5. DSL方法**

**此处不做本次课强制要求，暂时先了解即可，后面做实验会用到的。**

**JSON数据格式解析**

插件： Pipeline Utils Steps

// 插件

def response = readJSON text: "${scanResult}"

println(scanResult)

//原生方法

import groovy.json.\*

@NonCPS

def GetJson(text){

def prettyJson = JsonOutput.prettyPrint(text)

new JsonSlurperClassic().parseText(prettyJson)

}

**流水线中使用凭据**

withCredentials([string(credentialsId: "xxxxx", variable: "sonarToken")]) {

println(sonarToken)

}

**代码管理下载代码**

//Git

checkout([$class: 'GitSCM', branches: [[name: "brnachName"]],

doGenerateSubmoduleConfigurations: false,

extensions: [], submoduleCfg: [],

userRemoteConfigs: [[credentialsId: "${credentialsId}",

url: "${srcUrl}"]]])

//Svn

checkout([$class: 'SubversionSCM', additionalCredentials: [],

filterChangelog: false, ignoreDirPropChanges: false,

locations: [[credentialsId: "${credentialsId}",

depthOption: 'infinity', ignoreExternalsOption: true,

remote: "${svnUrl}"]], workspaceUpdater: [$class: 'CheckoutUpdater']]

)

**展示HTML报告**

插件：

publishHTML([allowMissing: false,

alwaysLinkToLastBuild: false,

keepAll: true,

reportDir: './report/',

reportFiles: "a.html, b.html",

reportName: 'InterfaceTestReport',

reportTitles: 'HTML'])

**获取当前的管道运行用户**

插件：[build user vars](https://plugins.jenkins.io/build-user-vars-plugin)

wrap([$class: 'BuildUser']){

echo "full name is $BUILD\_USER"

echo "user id is $BUILD\_USER\_ID"

echo "user email is $BUILD\_USER\_EMAIL"

}

**使用HttpRequest发起请求(CURL)**

插件： [HTTP Request](https://plugins.jenkins.io/http_request)

ApiUrl = "http://xxxxxx/api/project\_branches/list?project=${projectName}"

Result = httpRequest authentication: 'xxxxxxxxx',

quiet: true,

contentType: 'APPLICATION\_JSON' ,

url: "${ApiUrl}"

**6. Groovy编程**

参考文档：<http://docs.groovy-lang.org/docs/latest/html/documentation/#_map_coercion>

<http://groovy-lang.org/groovy-dev-kit.html>

Groovy是一种**功能强大，可选类型和动态 语言**，支持**Java平台**。旨在提高开发人员的生产力**得益于简洁**，熟悉且简单易学的语法。可以与任何Java程序顺利集成，并立即为您的应用程序提供强大的功能，包括脚本编写功能，**特定领域语言编写**，运行时和编译时元编程以及函数式编程。



**安装**

* 下载安装包(先安装JDK)
* 解压安装包 获取安装包bin目录
* 设置环境变量写入/etc/profile文件

<https://groovy.apache.org/download.html>

vi /etc/profile

export GROOVY\_HOME=/usr/local/groovy-3.0.7/

export PATH=$PATH:$GROOVY\_HOME/bin

source /etc/profile

groovysh

**注释：**

* 单行注释 //
* 多行注释 /\*\*/

**数据类型**

**字符串string**

字符串表示方式： 单引号、双引号、三单双引号。

//定义一个字符串类型变量name

String name = 'zhangsan'

String name = "zhangsan"

//定义一个变量包含多行内容

String zeyang = """

devops

"""

println(zeyang)

//字符串分割操作

String branchName = "release-1.1.1"

println(branchName.split("-"))

println(branchName.split("-")[-1])

println("${env.JOB\_NAME}".split("-")[0])

//是否包含release字符串

println(branchName.contains("release"))

//字符串的长度

println(branchName.size())

println(branchName.length())

//使用变量作为值

def message = "hello ${name}"

println(message)

println(message.toString())

//获取元素索引值

println(branchName.indexOf("-"))

//判断字符串以DEV结尾

String jobName = "test-service\_DEV"

println(jobName.endsWith("DEV"))

//字符串增添操作

String log = "error: xxxxxx aa"

println(log.minus("a"))

println(log - "a")

println(log.plus("aa"))

println(log + "aa")

//字符串反转

String nums = "1234567"

println(nums.reverse())

**列表list**

列表的表示： [] [1,2,3,4]

// list

// 定义一个list

def mylist = [1,2,3,4,4,"devops"]

println(mylist)

// list的元素增删

println(mylist + "jenkins")

println(mylist - "devops")

println(mylist << "java")

def newlist = mylist.add("gitlab")

println(newlist)

// 判断元素是否为空

println(mylist.isEmpty())

// 列表去重

println(mylist.unique())

// 列表反转

println(mylist.reverse())

// 列表排序

println(mylist.sort())

// 判断列表是否包含元素

println(mylist.contains("devops"))

// 列表的长度

println(mylist.size())

//扩展列表定义方式

String[] stus = ["zhangsan", "lisi","wangwu"]

def numList = [1,2,3,4,4,4] as int[]

// 通过索引获取列表元素

println(numList[2])

// 计算列表中元素出现的次数

println(numList.count(4))

Scripts not permitted to use staticMethod org.codehaus.groovy.runtime.DefaultGroovyMethods count int[] java.lang.Object. Administrators can decide whether to approve or reject this signature.

[Pipeline] End of Pipeline

**映射map**

types = [“maven”:“mvn”] [:]

// 定义map

def mytools = [ "mvn": "/usr/local/maven",

"gradle": "/usr/local/gradle" ]

// 根据key获取value

println(mytools["mvn"])

println(mytools["gradle"])

// 根据key重新赋值

mytools["mvn"] = "/opt/local/maven"

println(mytools)

mytools.gradle = "/opt/local/gradle"

println(mytools)

// 获取key的value

println(mytools.key("mvn"))

println(mytools.get("mvn"))

// 判断map是否包含某个key或者value

println(mytools.containsKey("gradle"))

println(mytools.containsValue("/usr/local/gradle"))

// 返回map的key 列表

println(mytools.keySet())

// 根据key删除元素

println(mytools.remove("mvn"))

println(mytools)

**条件语句**

**if语句**

在Jenkinsfile中可用于条件判断。

/\*

定义变量参数branchName

如果branchName 等于dev则打印dev，

如果branchName 等于test则打印test，

上面都不匹配则打印skipdeploy

\*/

String branchName = "dev"

if ( branchName == "dev" ){

println("dev....")

} else if (branchName == "test"){

println("test....")

} else {

println("skipdeploy......")

}

**switch语句**

/\*

定义参数branchName

匹配 develop 则打印develop ，跳出。

匹配 release 则打印release ，跳出。

默认匹配， 打印 error ，退出。

\*/

String branchName = "release"

switch(branchName) {

case "develop":

println("develop .....")

break

case "release":

println("release.....")

break

default:

println("error。。。。。。")

}

**for循环语句**

// for

// 遍历0-9，打印

for (i=1; i<10; i++ ){

println(i)

}

// 循环5次

5.times {

println("hello")

}

// 遍历 0-4

5.times { i ->

println(i)

}

// 遍历List

def serverList = ["server-1", "server-2", "server-3"]

for ( i in serverList){

println(i)

}

// 使用each遍历map

def stus = ["zeyang":"177", "jenkins":"199"]

stus.each { k, v ->

println(k+"="+v)

}

// 使用for遍历map

for (k in stus.keySet()){

println(k+"="+stus[k])

}

**while循环语句**

// while 循环

String name = "jenkins"

while (name == "jenkins"){

println("true....")

name = "lisi"

}

**异常处理**

/\*

如果println(a,b)失败（肯定失败，因为有语法错误）

catch捕获错误，并打印错误。

finally 总是执行。

\*/

try {

println(a,b)

}

catch(Exception e) {

println(e)

}

finally {

println("done")

}

**函数定义与使用**

在共享库中每个类中的方法。

/\*

def关键字 定义函数名为PrintMes， 带有一个参数msg，语句块内容是打印msg参数的值，返回msg值。

将PrintMsg函数的执行结果返回给response变量。

打印response

\*/

def PrintMsg(msg){

println(msg)

return msg

}

response = PrintMsg("jenkins ok okok!")

println(response)

**7. Jenkins共享库实践**

共享库这并不是一个全新的概念，其实在编程语言Python中，我们可以将Python代码写到一个文件中，当代码数量增加，我们可以将代码打包成模块然后再以import的方式使用此模块中的方法。

在Jenkins中使用Groovy语法，**共享库中存储的每个文件都是一个groovy的类**，**每个文件（类）中包含一个或多个方法。每个方法包含groovy语句块**。

可以在Git等版本控制系统中创建一个项目用于存储共享库。共享流水线有助于减少冗余并保持代码整洁。

属性：

* 共享库名称
* 共享库版本
* 共享库地址

├── src

│ └── org

│ └── devops

│ └── tools.groovy

├── vars

│ └── GetHosts.groovy

│ └── GetCommitId.groovy

│

└── resources

│ └── org

│ └── devops

│ └── config.json

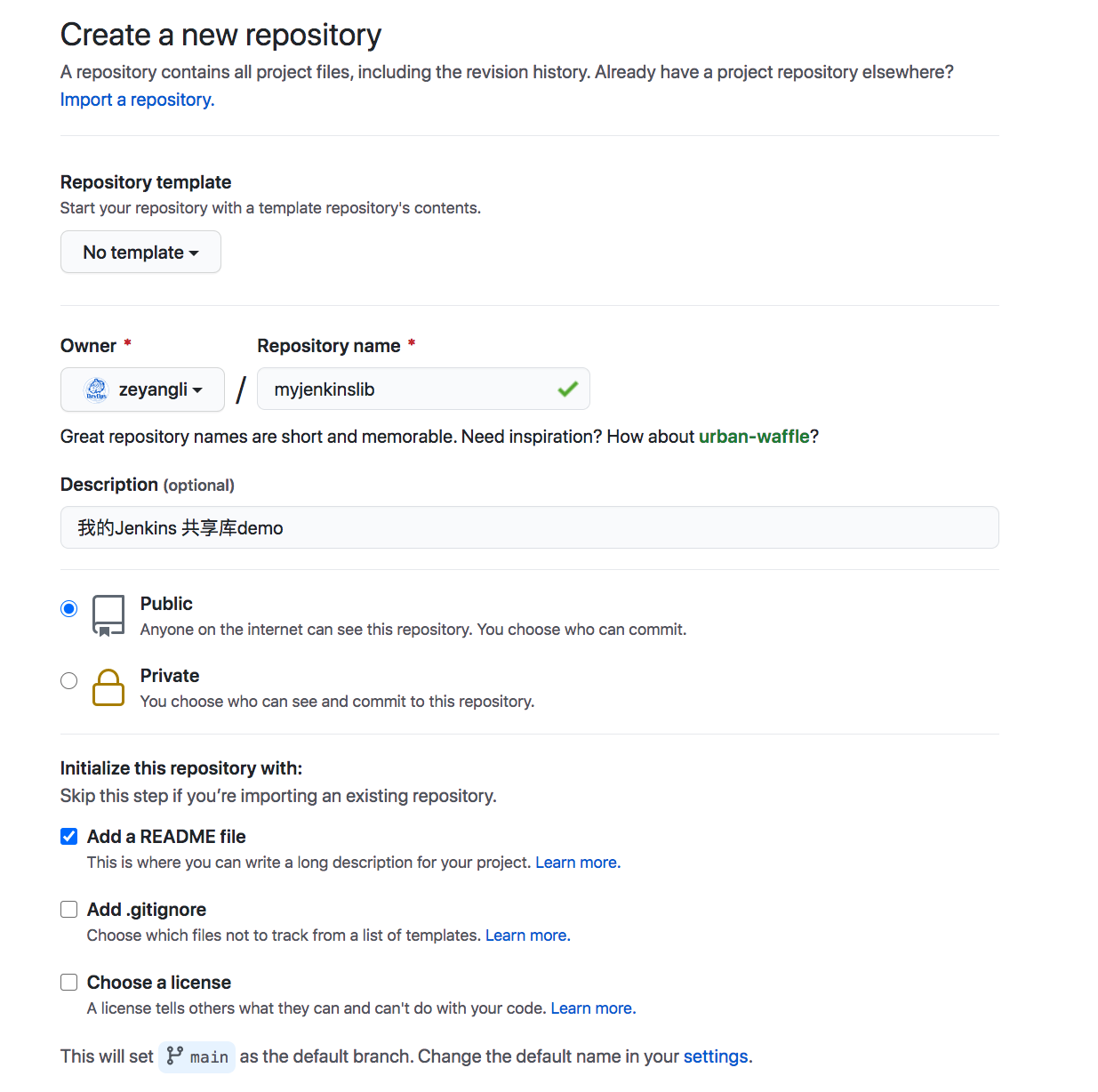
库结构:

* src： 类似于java的源码目录，执行流水线时会加载到class路径中。
* vars: 存放全局变量脚本，小的功能函数。
* resources： 存放资源文件，类似于配置信息文件。

**共享库配置**

**1. 创建一个共享库**

可以直接在github中创建一个公开类型的仓库，**仓库名称自定义**。公开类型的仓库是为了便于验证，也可以创建私有类型的，但是需要提前配置好仓库的认证凭据。



**2. 创建groovy类文件**

参考文件： <https://github.com/zeyangli/myjenkinslib/blob/main/src/org/devops/jenkinstest.groovy>

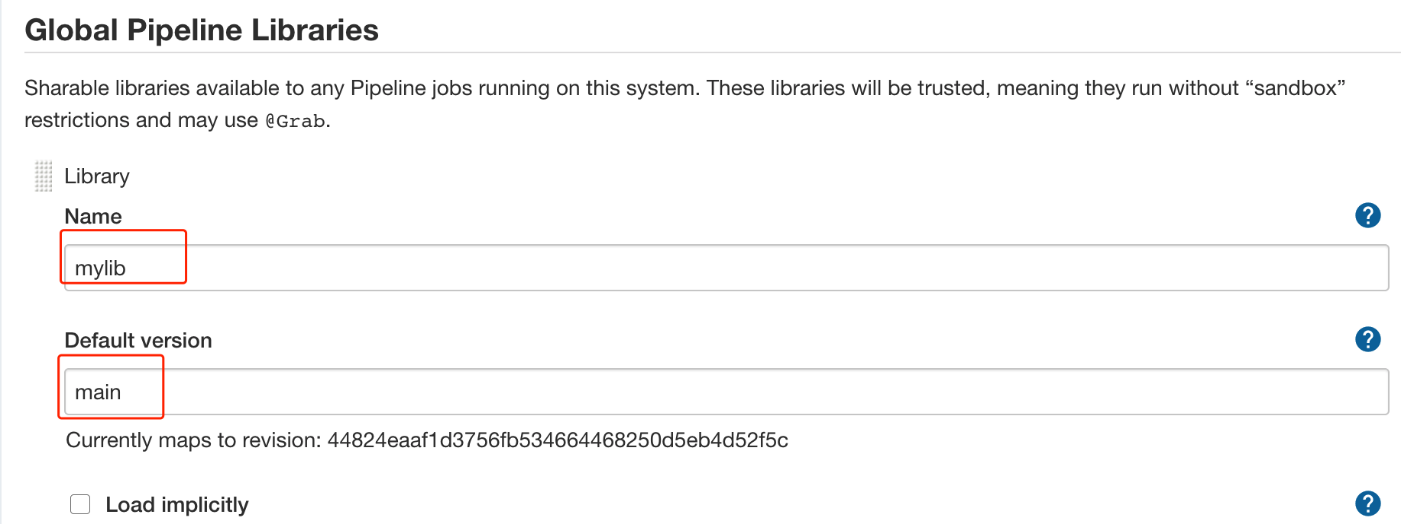
直接在github仓库的页面操作即可：



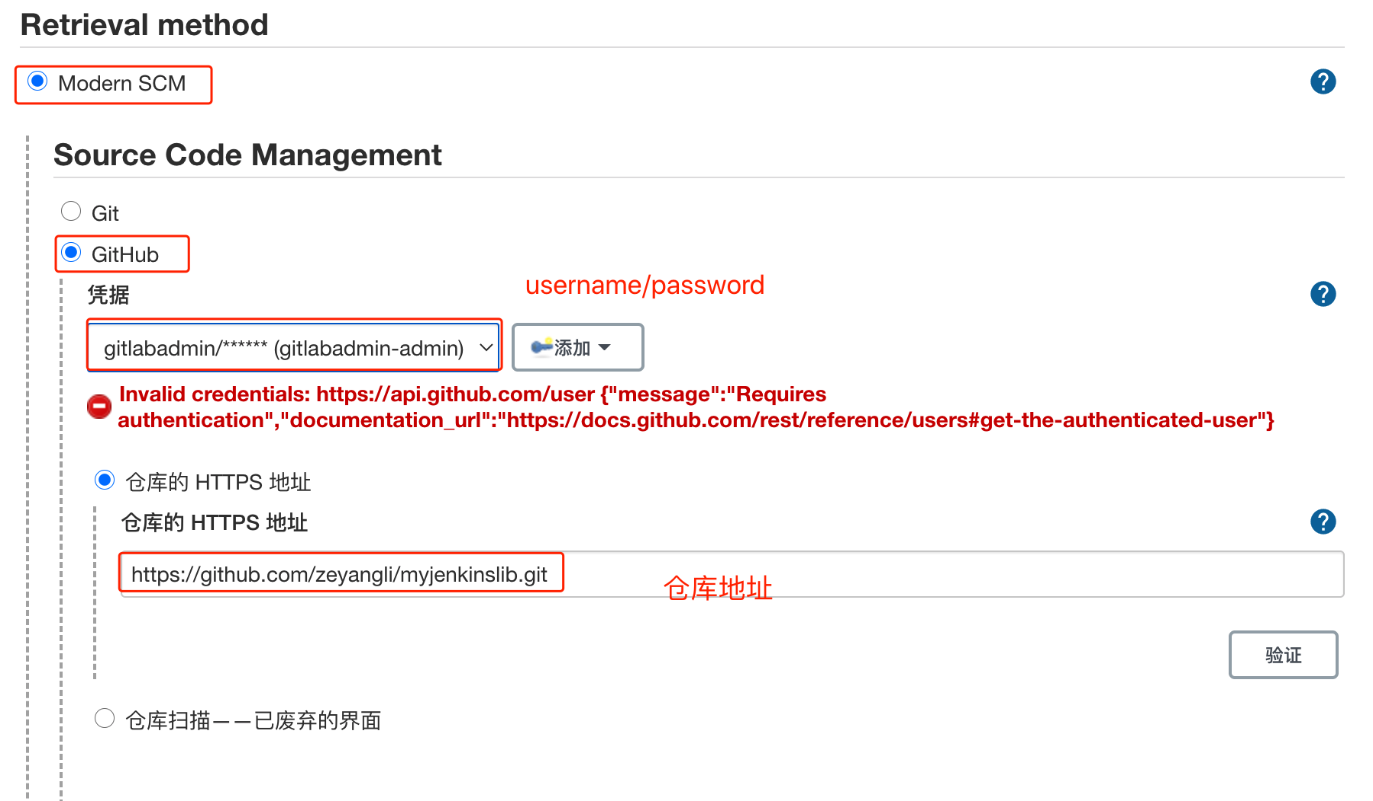
**使用共享库**

**Jenkins系统配置 -> Global Pipeline Libraries**

首先，我们为共享库设置一个名称 **mylib** （**自定义，无需与github仓库一致**），**注意这个名称后续在Jenkinsfile中引用**。 再设置一个默认的版本，这里的版本是**分支的名称**。我默认配置的是main版本。（github默认版本必须是main）



接下来我们配置**共享库的仓库地址，**我的仓库在github中，所以这里我填写的是github的方式。（如果你用的是gitlab可以使用gitlab方式或者git方式）。如果仓库是私有的方式，需要在jenkins的凭据中添加一个账号用于下载共享库。



在Jenkinsfile中使用@Library('mylib') \_ 来**加载共享库**，注意后面符号\_**用于加载**。 类的实例化def mytools = new org.devops.jenkinstest(),使用类中的方法 `mytools.PrintMsg(msg)` 。

@Library('mylib') \_

def mytools = new org.devops.jenkinstest()

pipeline {

agent { label "master" }

stages {

stage("build"){

steps{

script{

msg = "hello jenkins"

mytools.PrintMsg(msg)

}

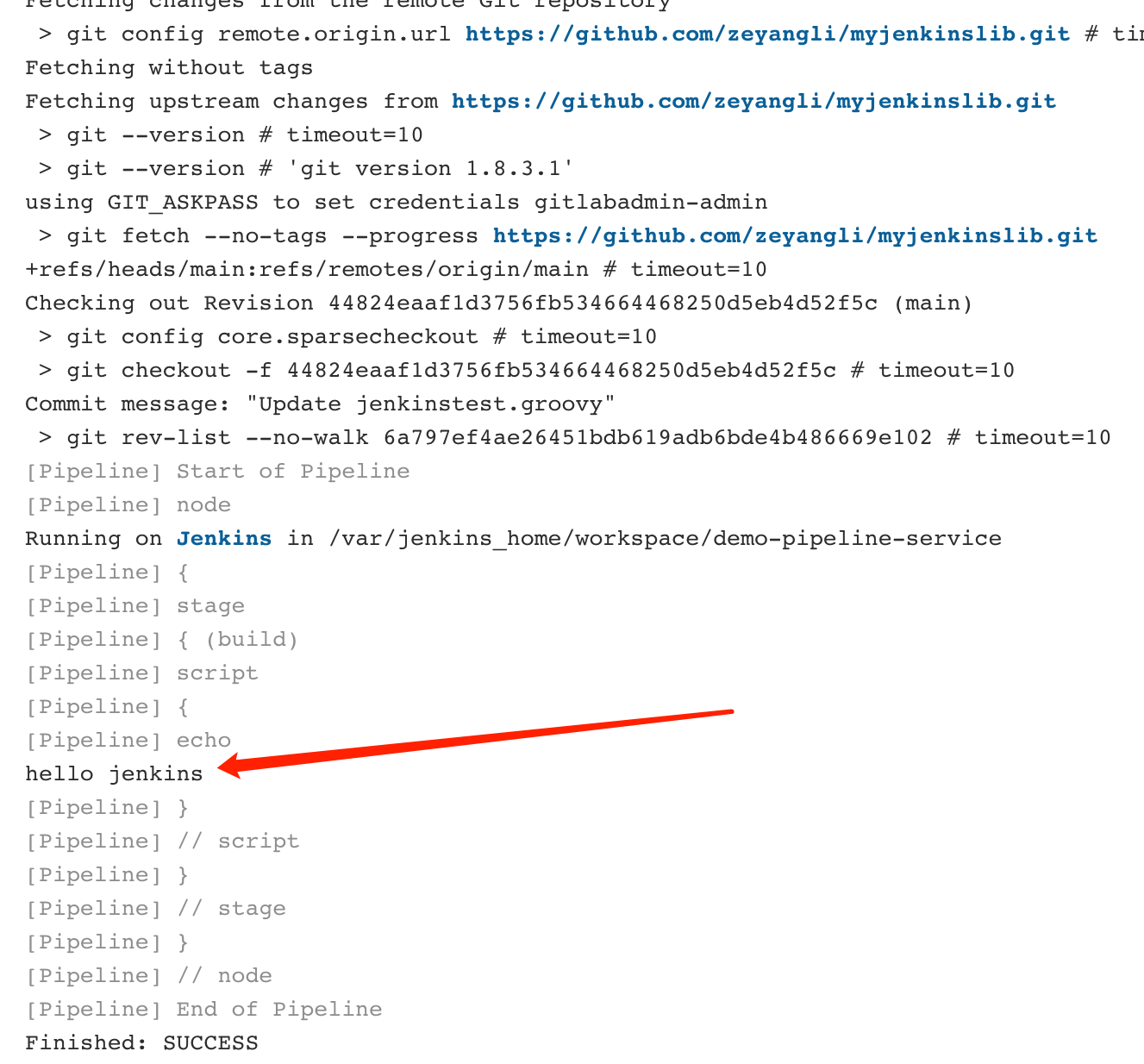
}

}

}

}

运行流水线查看测试结果

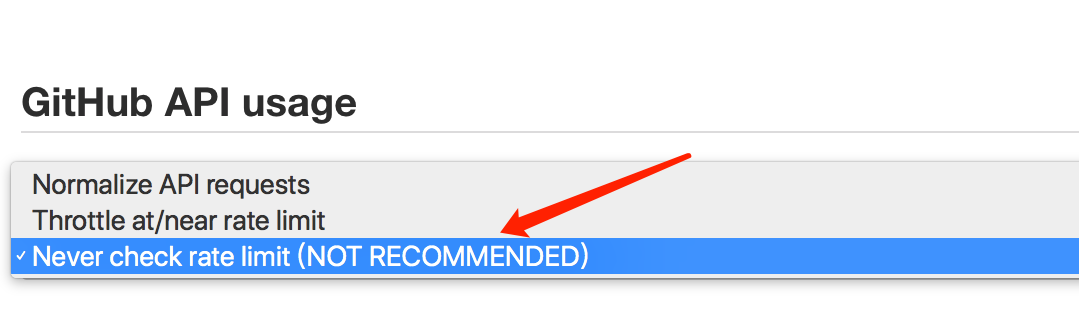


**可能遇到的错误： github api问题**

08:42:54 Jenkins-Imposed API Limiter: Current quota for Github API usage has 48 remaining (2 over budget). Next quota of 60 in 53 min. Sleeping for 5 min 27 sec.

08:42:54 Jenkins is attempting to evenly distribute GitHub API requests. To configure a different rate limiting strategy, such as having Jenkins restrict GitHub API requests only when near or above the GitHub rate limit, go to "GitHub API usage" under "Configure System" in the Jenkins settings.

解决方法： Jenkins系统设置 ，改下参数。



**FAQ: 找不到Github master 分支**

Attempting to resolve master as a branch

Attempting to resolve master as a tag

ERROR: Could not resolve master

ERROR: No version master found for library mylib

org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:

WorkflowScript: Loading libraries failed

**共享库扩展**

共享库加载

// 加载mylib共享库

@Library('mylib') \_

// 加载mylib共享库的1.0版本

@Library('mylib@1.0') \_

// 加载多个共享库， mylib共享库的默认版本， yourlib共享库的2.0版本（分支）

@Library(['mylib', 'yourlib@2.0']) \_