Jenkins 持续集成与持续交付系列

Jenkins Pipeline全解析

主讲人：杨晓飞
yangxf@doudianyun.com

北京逗点云创科技有限公司

Learn Today
Lead Tomorrow


Jenkins 持续集成与持续交付系列
Jenkins Pipeline全解析
Jenkins Pipeline

Jenkins Pipeline 全解析系列

第一节： Pipeline 简介

Jenkins Pipeline概念

- 一套插件，将持续交付(CD)的实现和实施集成到 Jenkins 中
- 持续交付的一种自动化流程实现，将基于版本控制管理的软件持续的交付到用户和消费者手中
- 提供了一套可扩展的工具，用于将各种简单或复杂的交付流程实现为“持续交付即代码”(Continuous delivery as code)
- Jenkins Pipeline 的定义通常被写入到一个文本文件（称为 Jenkinsfile ）中，该文件可以被放入项目的源代码控制库中
- 通过Pipeline Domain Specific Language（DSL）syntax 可以达到Pipeline as Code（Jenkinsfile存储在项目的源代码库）的目的

Pipeline 优点

- 代码 （Code）:Pipeline以代码的形式实现，通常可以被源代码控制，使团队能够编辑、审查和迭代其CD流程
- 可持续性(Durable)：Jenkins重启或者中断后都不会影响Pipeline Job
- 暂停运行(Pausable)：Pipeline可以选择暂停并等待输入，然后再继续Pipeline运行
- 多功能(Versatile)：Pipeline支持复杂的CD要求，包括fork/join子进程，循环和并行执行任务的能力
- 可扩展(Extensible)：Pipeline插件支持自定义扩展以及与其他插件集成

Pipeline基础语法

声明式(Declarative)与脚本化(Scripted) Pipeline语法

- Jenkinsfile可以使用两种语法编写 - Declarative和Scripted，声明式和脚本化 Pipeline 语法构造不同。 Declarative Pipeline是Jenkins Pipeline最近的一个特性（Pipeline Plugin 2.5 引入），包括：
  - 提供比Scripted Pipeline语法更丰富的语法功能
  - 以及旨在使编写和理解Pipeline代码更容易
- Scripted Pipeline 是基于Groovy语言实现

Pipeline基础语法示例

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any (1)
    stages {
        stage('Stage 1') {
            steps {
                echo 'Hello world!' (2)
            }
        }
    }
}
```

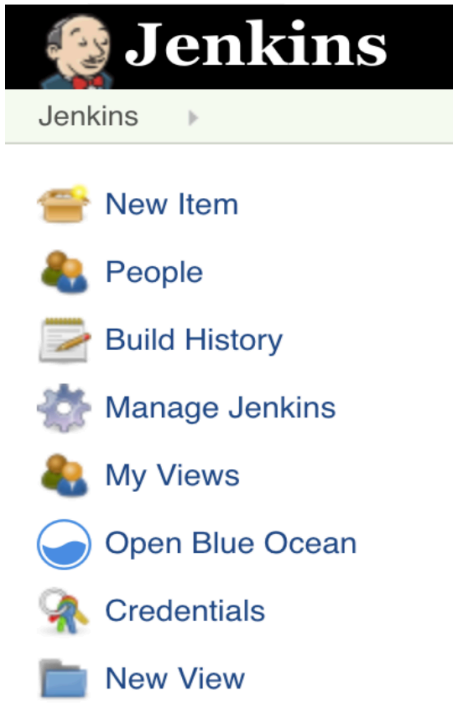
Jenkinsfile (Scripted Pipeline)

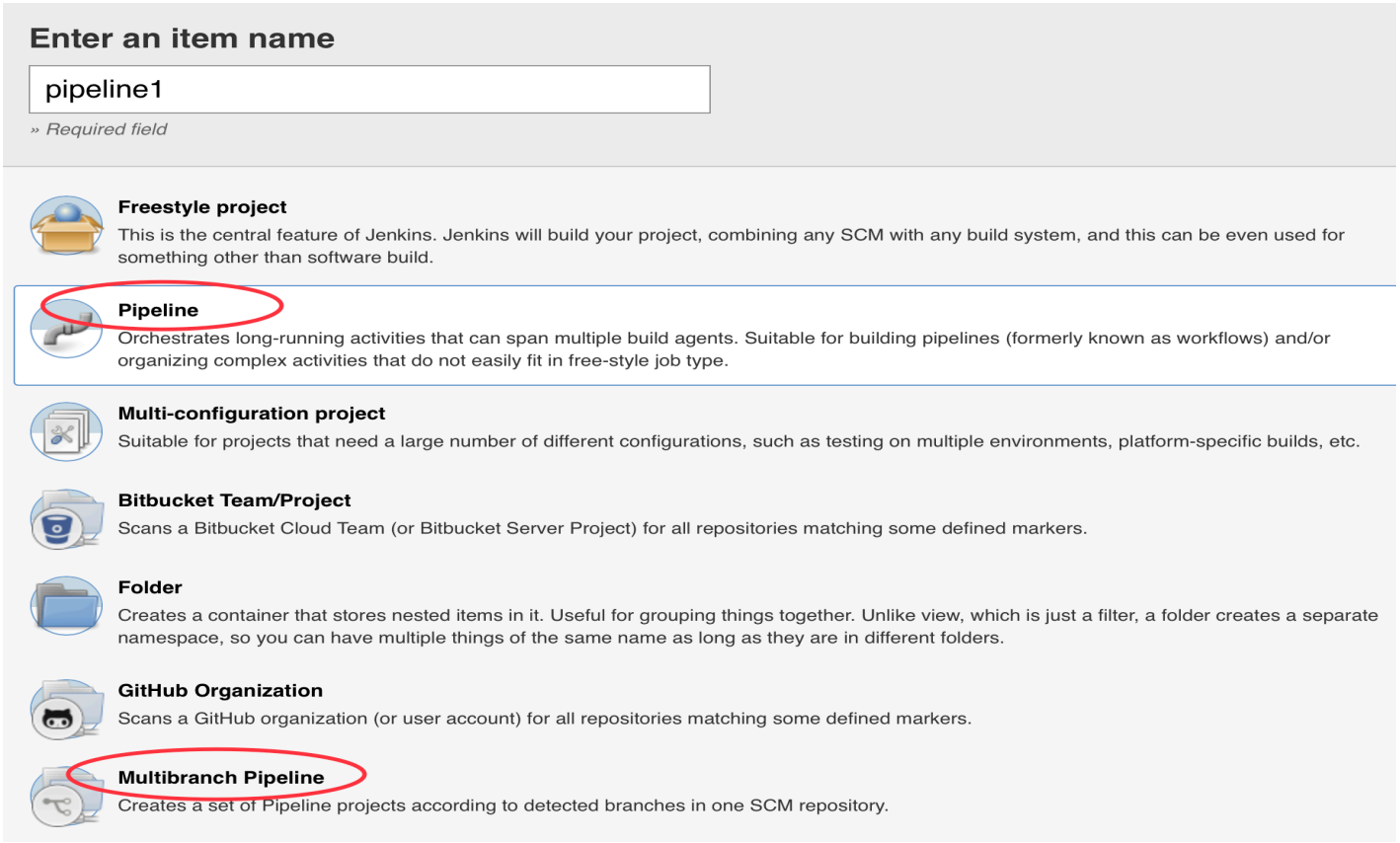
```
node { (3)
    stage('Stage 1') {
        echo 'Hello World' (2)
    }
}
```

- Agent:Jenkins为Pipeline分配的执行空间和工作空间
- Echo在控制台输出字符串
- Node作用等同于Agent

创建Pipeline流程实现

- 单击Jenkins中的New Item 菜单





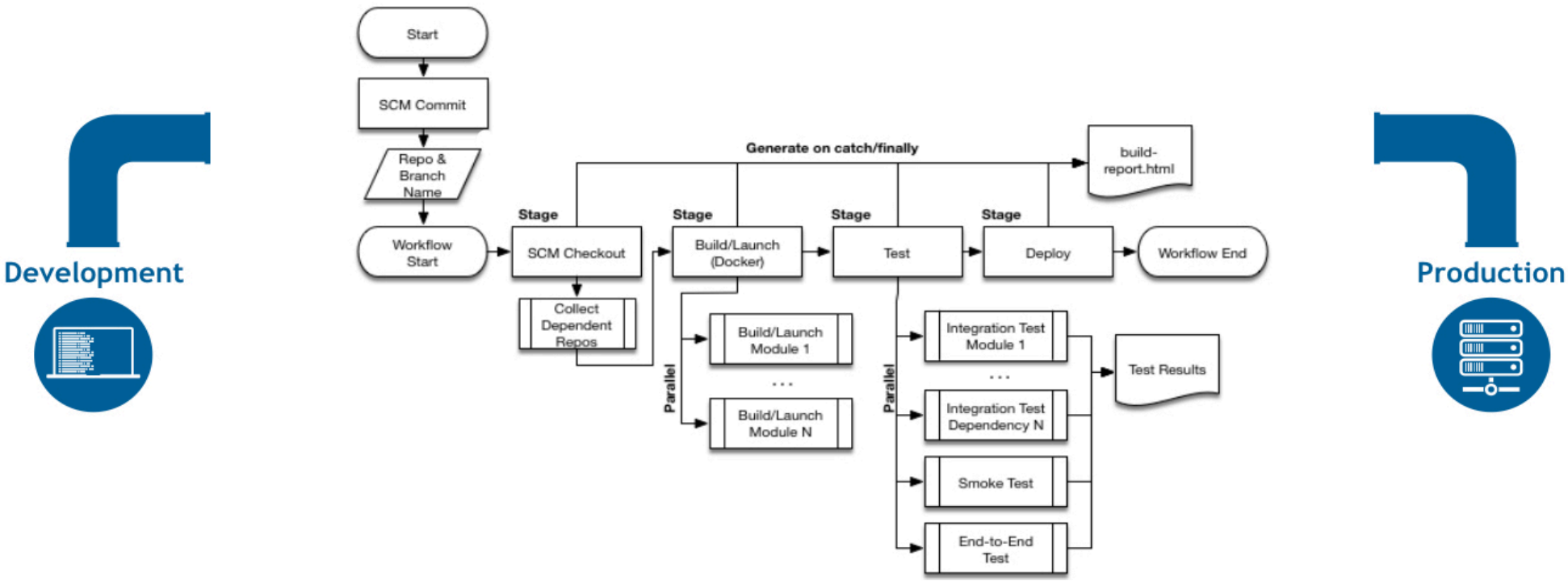
Pipeline语法基本概念

Pipeline通过Domain Specific Language（DSL）syntax定义Pipeline as Code并且实现持续交付的目的。Pipeline的代码定义了整个构建过程，通常包括构建应用程序，测试然后交付应用程序的阶段。下面是Pipeline语法中基本概念：

- Stage：一个Pipeline可以划分成若干个Stage，每个Stage代表一组操作，例如：“Build”，“Test”，“Deploy”。注意，Stage是一个逻辑分组的概念，可以跨多个Node（Agent）
- Node：一个Node就是一个Jenkins节点，或者是Master，或者是Agent，是执行Step的具体运行环境
- Step：Step是最基本的操作单元，小到创建一个目录，大到构建一个Docker镜像，由各类Jenkins Plugin提供，例如：sh ‘make’

Pipeline示意流程图

- 下面是在Jenkins Pipeline中模拟CD场景的示例流程图



Jenkins Pipeline 全解析系列

第二节： 声明式Pipeline基本语法简介

Declarative（声明式） Pipeline

- Declarative （声明式）Pipeline是Jenkins Pipeline最近的特性（Pipeline Plugin 2.5 引入），包括：
  - 提供比Scripted Pipeline语法更丰富的语法功能
  - 声明式Pipeline代码更容易编写和理解
- 所有Declarative Pipeline指令必须包含在Pipeline块中，如下：

```
pipeline {
    /* insert Declarative Pipeline here */
}
```

Declarative（声明式） Pipeline基本结构

Declarative Pipeline 需注意：

- 每个声明语句必须独立一行，行尾无需使用分号
- 声明块（blocks{}）只能包含区块（Sections），指令（Directives），步骤（Steps）或赋值语句

- **区块（Sections）**： 通常包含一个或多个指令或步骤，如 agent 、post、stages、steps
- **指令（Directives）**：通常指 environment、options、parameters、triggers（触发）、stage、tools、when
- **步骤（Steps）**：可以使用“Step Reference”参考中记录的所有可用步骤，包含完整的步骤列表，链接：https://jenkins.io/doc/pipeline/steps/

Declarative Pipeline语法细节

- 详细细节，参见 https://jenkins.io/doc/book/pipeline/syntax/ ，列举声明式Pipeline语法支持的指令步骤等
- 以Agent 区块指令为例

  必选项，agent必须在Pipeline块内的顶层定义，但stage内是否使用为可选的

  支持参数：any/none/label/node/docker/dockerfile

  常用选项： label/customWorkspace/reuseNode

```
pipeline {
    agent { docker 'maven:3-alpine' }
    stages {
        stage('Example Build') {
            steps {
                sh 'mvn -B clean verify'
            }
        }
    }
}
```

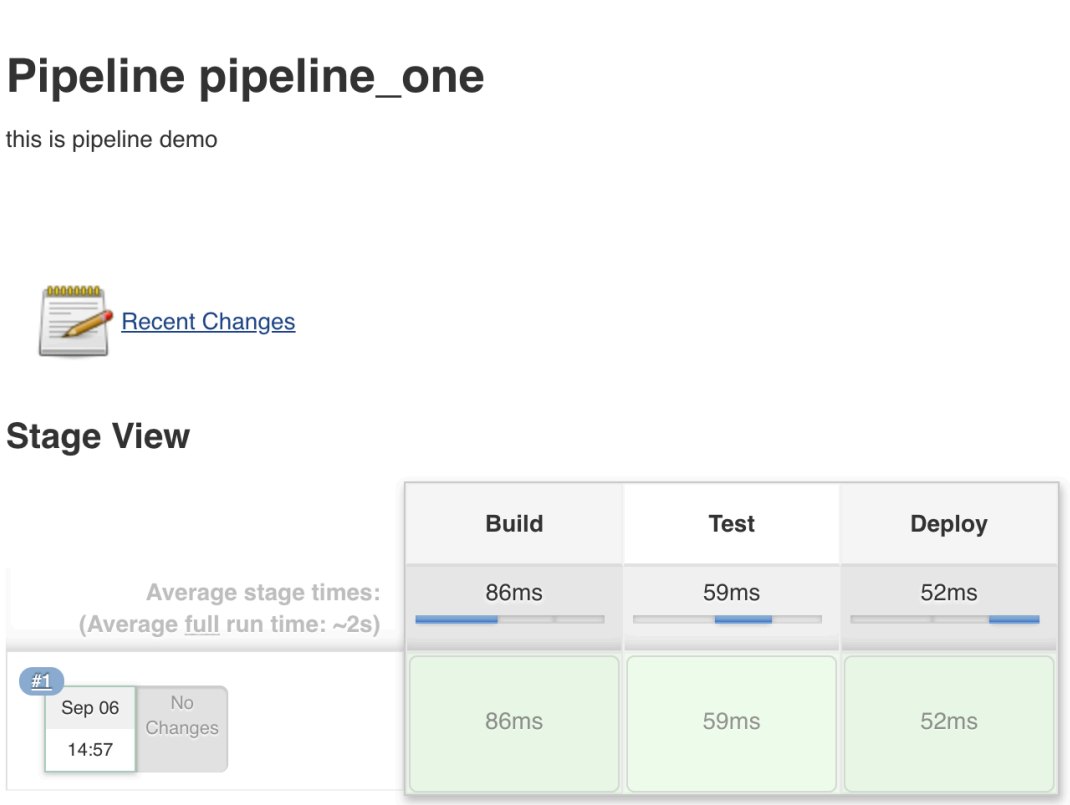
Declarative Pipeline基本结构

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

1. 在任何可用的Agent上执行此Pipeline或其任何阶段
2. 定义“构建”阶段
3. 执行与“构建”阶段相关的步骤
4. 定义“测试”阶段
5. 执行与“测试”阶段相关的步骤
6. 定义“部署”阶段
7. 执行与“部署”阶段相关的步骤

Declarative Pipeline示例

```
pipeline{
    agent any
    stages {
        stage('Build') {
            steps{
                echo 'This is a build step'
            }
        }
        stage('Test') {
            steps{
                echo 'This is a test step'
            }
        }
        stage('Deploy') {
            steps{
                echo 'This is a deploy step'
            }
        }
    }
}
```

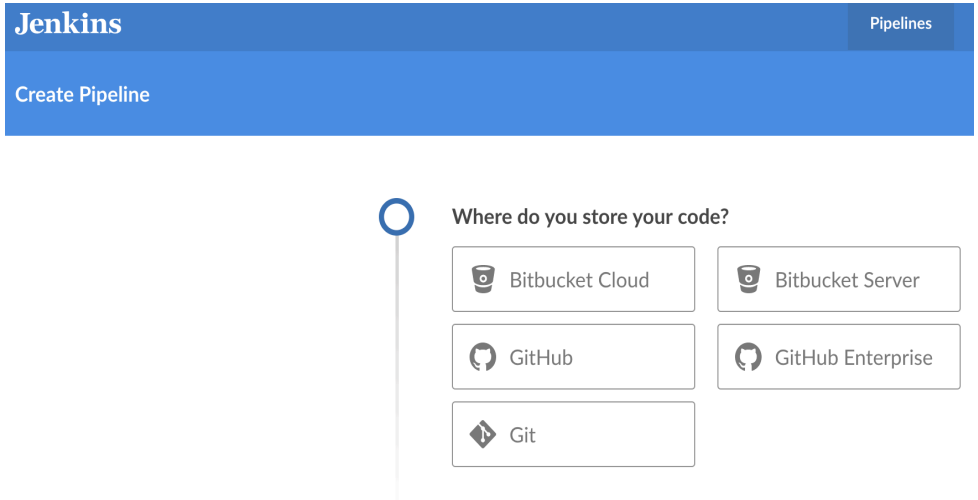


Jenkins Pipeline 全解析系列

第三节： BlueOcean简介

Blue Ocean是什么

- BlueOcean UI是一组Jenkins插件，需要Jenkins版本2.7以上
- Blue Ocean 旨在提升 Jenkins Pipeline 用户体验，与自由式作业(FreeStyle)兼容
- Blue Ocean目的是在执行任务时，降低CI和CD工作流程的复杂度和提升工作流程的清晰度



Blue Ocean特性

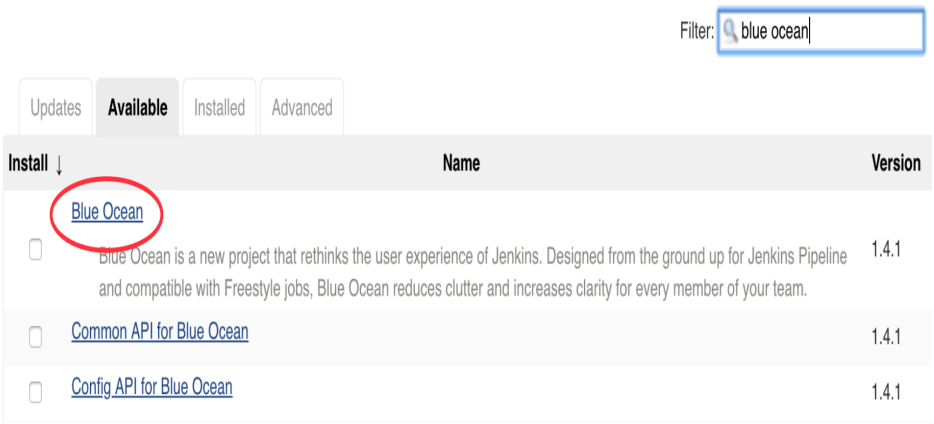
- 流水线编辑器：用于创建贯穿始终的持续交付流水线，是一种直观并可视化的流水线编辑器
- 流水线的可视化：对流水线的可视化表示，提高了持续交付过程的清晰度
- 流水线的诊断：即刻定位自动化问题，无需持续扫描日志或关注多个屏幕，在需要干预和/或出现问题时，精确定位
- 与SCM良好集成：在与GitHub 和 Bitbucket中的其他人协作编码时实现最大程度的开发人员生产力

安装 Blue Ocean

Blue Ocean 可以通过以下方式安装：

- 作为 已有Jenkins实例上的一组插件
- Jenkins Docker镜像的一部分

具体参见：https://jenkins.io/doc/book/pipeline/getting-started/

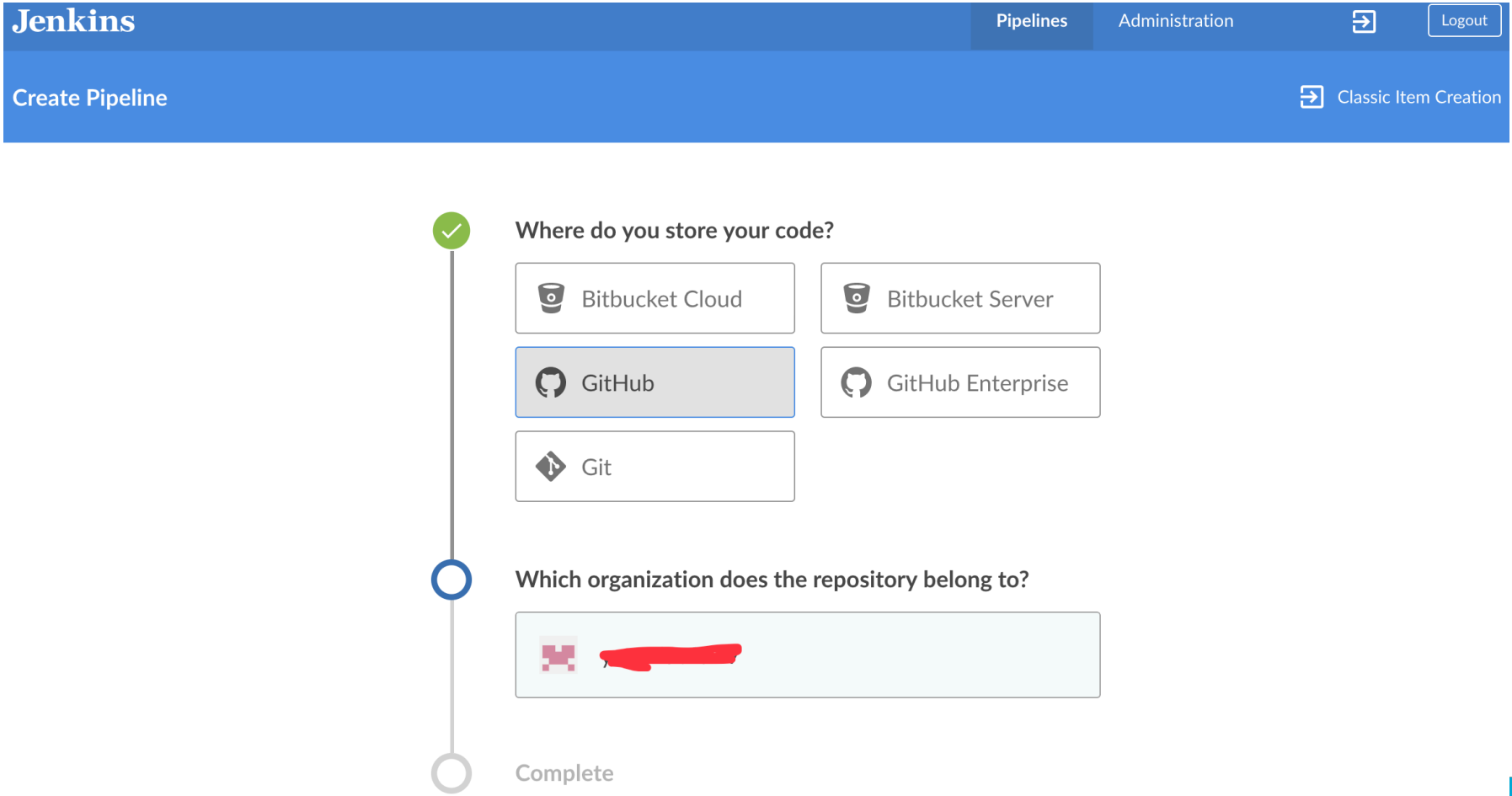



```
docker run \
  -u root \
  --rm \
  -d \
  -p 8080:8080 \
  -p 50000:50000 \
  -v jenkins-data:/var/jenkins_home \
  -v /var/run/docker.sock:/var/run/docker.sock \
  jenkinsci/blueocean
```

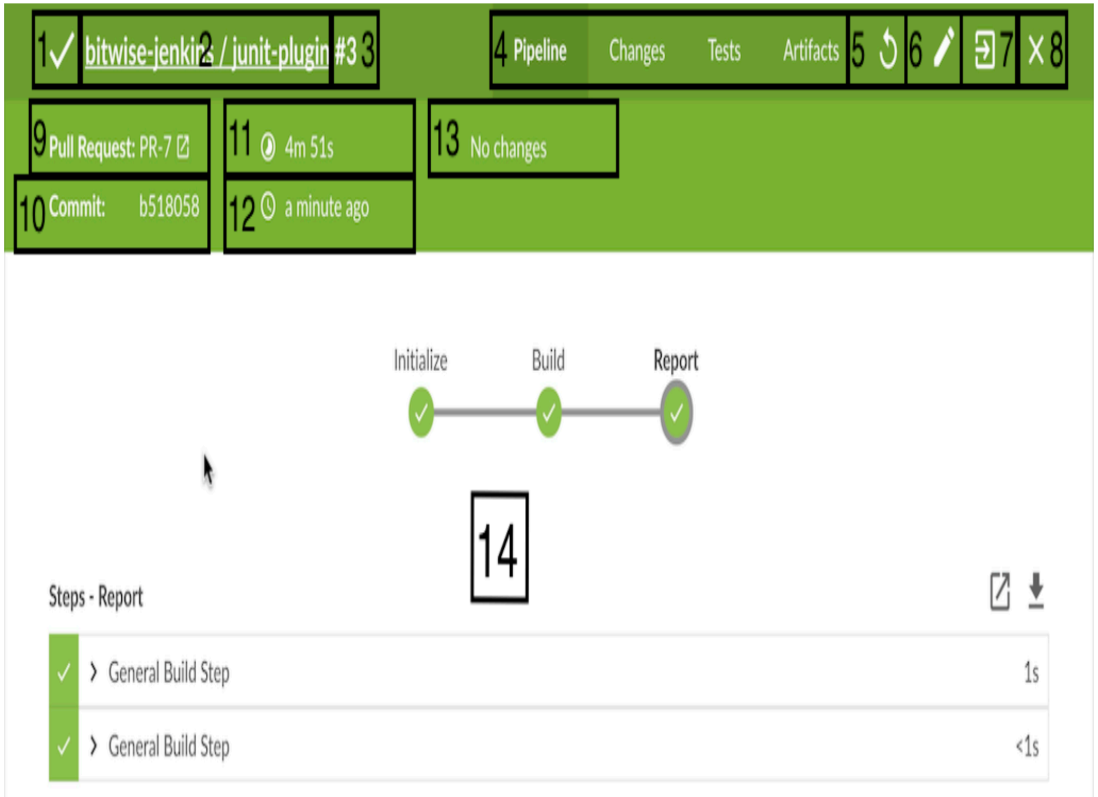
访问Blue Ocean

登录Jenkins经典UI 后，通过点击左侧的 打开Blue Ocean 来访问Blue Ocean页面

Blue Ocean 中创建Pipeline

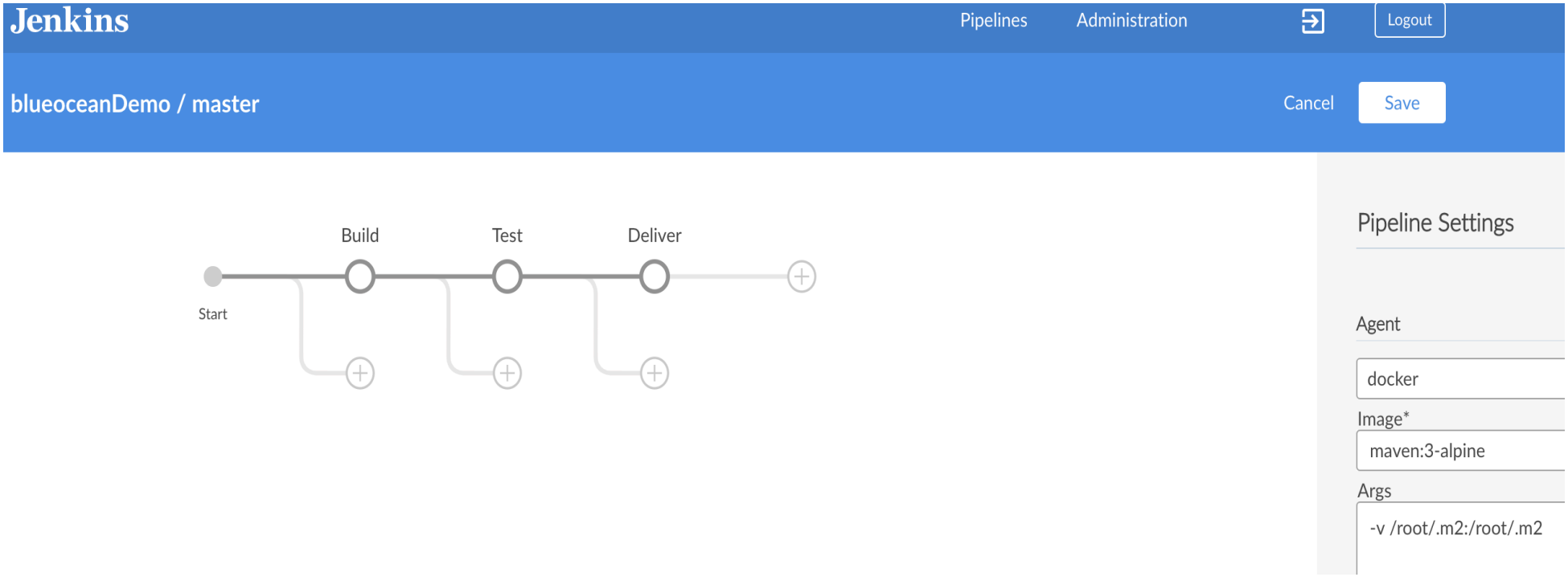


Pipeline运行视图

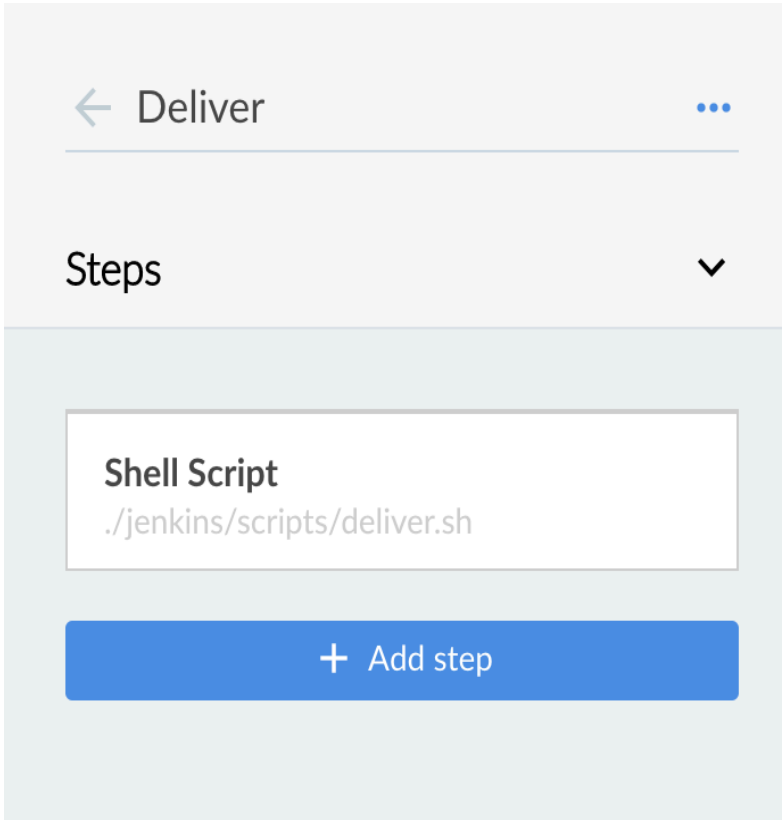


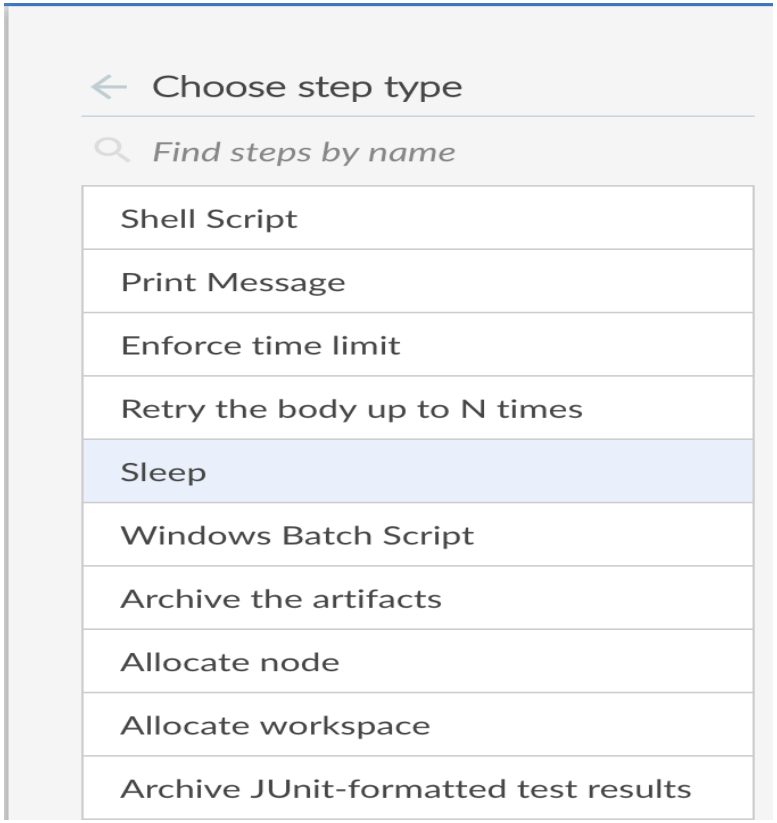
1. Run Status - 此图标, 以及顶部菜单栏的背景色,表明了流水线运行的状态
2. Pipeline Name - 运行的流水线的名称
3. Run Number - 流水线运行的ID号。 对于流水线的每个分支(和 Pull 请求) , ID号是唯一的
4. Tab Selector - 查看该运行细节选项卡。 默认值是“Pipeline”
5. Re-run Pipeline - 再次执行运行的流水线
6. Edit Pipeline - 在 Pipeline Editor中打开运行的流水线
7. Go to Classic - 转换到该运行细节的“经典” UI 视图
8. Close Details - 这将关闭细节视图并使用户回到该流水线的 活动视图
9. Branch or Pull Request - 该运行的分支或pull请求
10. Commit Id - 提交本次运行的ID
11. Duration - 本次运行的持续时间
12. Completed Time - 多久之前运行完成
13. Change Author - 更改的作者姓名
14. Tab View - 显示所选选项卡的信息

编辑Pipeline


Jenkins
Pipelines
Administration
Logout
blueoceanDemo / master
Cancel
Save
Build
Test
Deliver
Start
Pipeline Settings
Agent
docker
Image*
maven:3-alpine
Args
-v /root/.m2:/root/.m2

编辑Pipeline Step





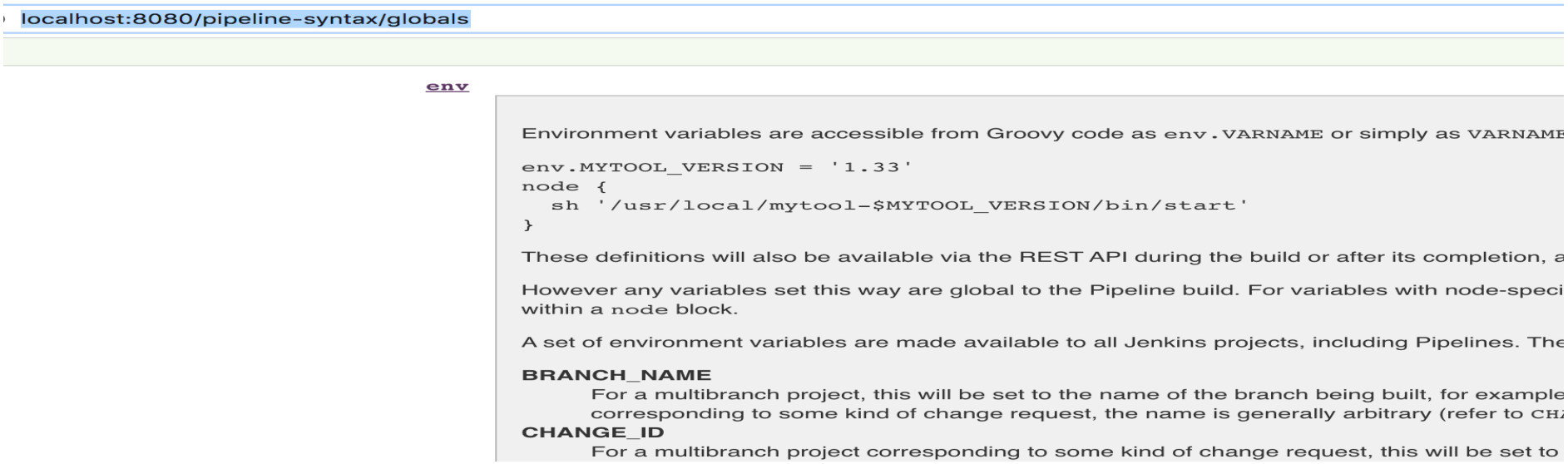
逗点云创

Jenkins Pipeline 全解析系列

第四节： Pipeline环境变量

Pipeline全局变量 Env

Jenkins Pipeline通过全局变量 env 提供环境变量，可在Jenkinsfile 文件的任何地方使用。可访问${YOUR_JENKINS_URL}/pipeline-syntax/globals#env得到完整环境变量列表



Pipeline全局变量 Env

BUILD_ID
当前构建的 ID，与 Jenkins 版本 1.597+ 中创建的构建号 BUILD_NUMBER 是完全相同的

BUILD_NUMBER
当前构建号，比如 “153”

BUILD_TAG
字符串 ``jenkins-${JOB_NAME}-${BUILD_NUMBER}``。可以放到源代码、jar 等文件中便于识别

BUILD_URL
可以定位此次构建结果的 URL（比如 http://buildserver/jenkins/job/MyJobName/17/ ）

EXECUTOR_NUMBER
用于识别执行当前构建的执行者的唯一编号（在同一台机器的所有执行者中）。这个就是你在“构建执行状态”中看到的编号，只不过编号从 0 开始，而不是 1

Pipeline全局变量 Env

JAVA_HOME
如果你的任务配置了使用特定的一个 JDK，那么这个变量就被设置为此 JDK 的 JAVA_HOME。当设置了此变量时，PATH 也将包括 JAVA_HOME 的 bin 子目录

JENKINS_URL
Jenkins 服务器的完整 URL，比如 https://example.com:port/jenkins/ （注意：只有在“系统设置”中设置了 Jenkins URL 才可用）

JOB_NAME
本次构建的项目名称，如 “foo” 或 “foo/bar”

NODE_NAME
运行本次构建的节点名称。对于 master 节点则为 “master”

WORKSPACE
workspace 的绝对路径

引用全局环境变量

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
            }
        }
    }
}
```

设置环境变量

在 Jenkins 流水线中，对于声明式和脚本式Pipeline，设置环境变量的方法不同

- 声明式Pipeline使用 environment 指令

  Environment 指令制定一个 键-值对序列，该序列定义环境变量，环境变量适用于所有步骤或者是特定某阶段的步骤， 取决于 environment 指令在流水线内的位置

- 脚本式Pipeline使用 withEnv 步骤

设置环境变量

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    environment { (1)
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment { (2)
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
```

1. 用在最高层的 pipeline 块的 environment 指令适用于流水线的所有步骤
2. 定义在 stage 中的 environment 指令只适用于 stage 中的步骤

Jenkinsfile (Scripted Pipeline)

```
node {
    /* .. snip .. */
    withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {
        sh 'mvn -B verify'
    }
}
```

动态设置环境变量

1. 环境变量可在运行时设置，各种脚本都可以返回 “returnStatus” 或 “returnStdout”
2. 使用 returnStdout 时，返回的字符串末尾会追加一个空格。可以使用 .trim() 将其移除

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    environment {
        // 使用 returnStdout
        CC = """${sh(
                returnStdout: true,
                script: 'echo "clang"'
            )}"""
        // 使用 returnStatus
        EXIT_STATUS = """${sh(
                returnStatus: true,
                script: 'exit 1'
            )}"""
    }
```

```
    stages {
        stage('Example') {
            environment {
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
```

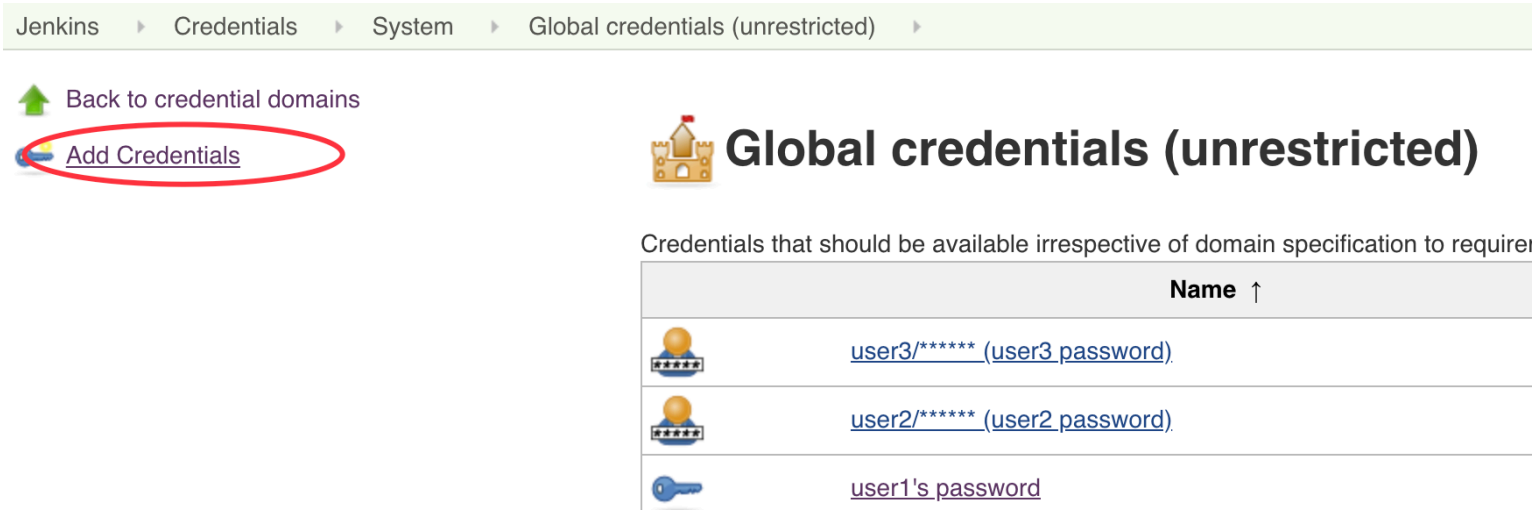
逗点云创

Jenkins Pipeline 全解析系列

第五节： Pipeline 安全凭据处理（一讲）：定义安全凭据

使用 Credentials

- Jenkins管理员在Jenkins中添加/配置credentials后，Jenkins项目就可使用 credentials
- Jenkins credentials功能由 Credentials Binding 插件提供
- 存储在Jenkins中的credentials，如是全局credentials，则可在Jenkins的任何地方使用

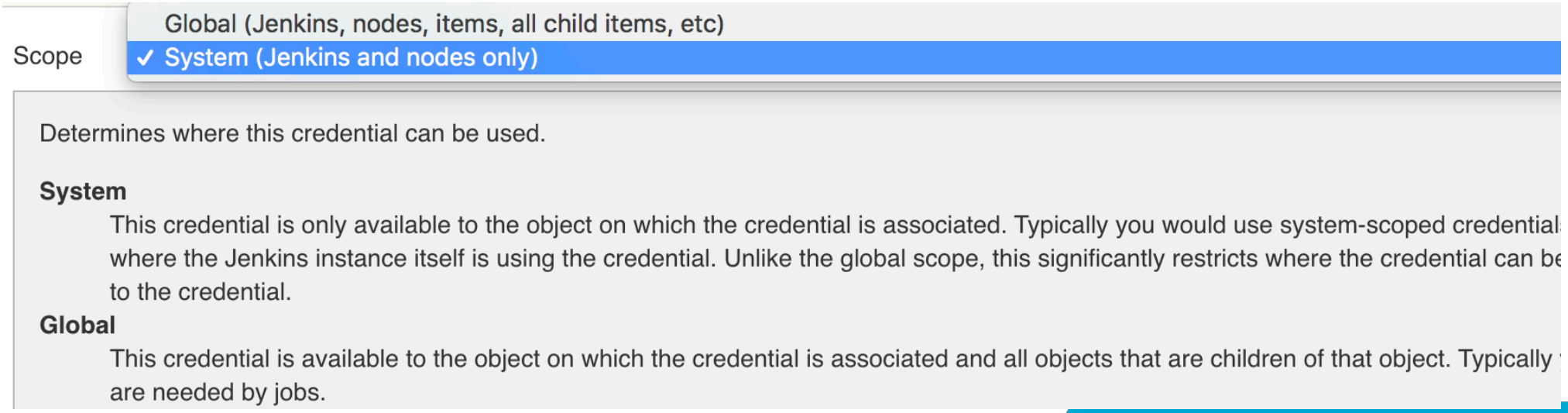


Jenkins Credential 类型

- Secret text - API token之类的token (如GitHub个人访问token)
- Username and password - 可以为独立的字段，也可以为冒号分隔的字符串：username:password
- Secret file - 保存在文件中的加密内容
- SSH Username with private key - SSH 公钥/私钥对
- Certificate - a PKCS#12 证书文件 和可选密码

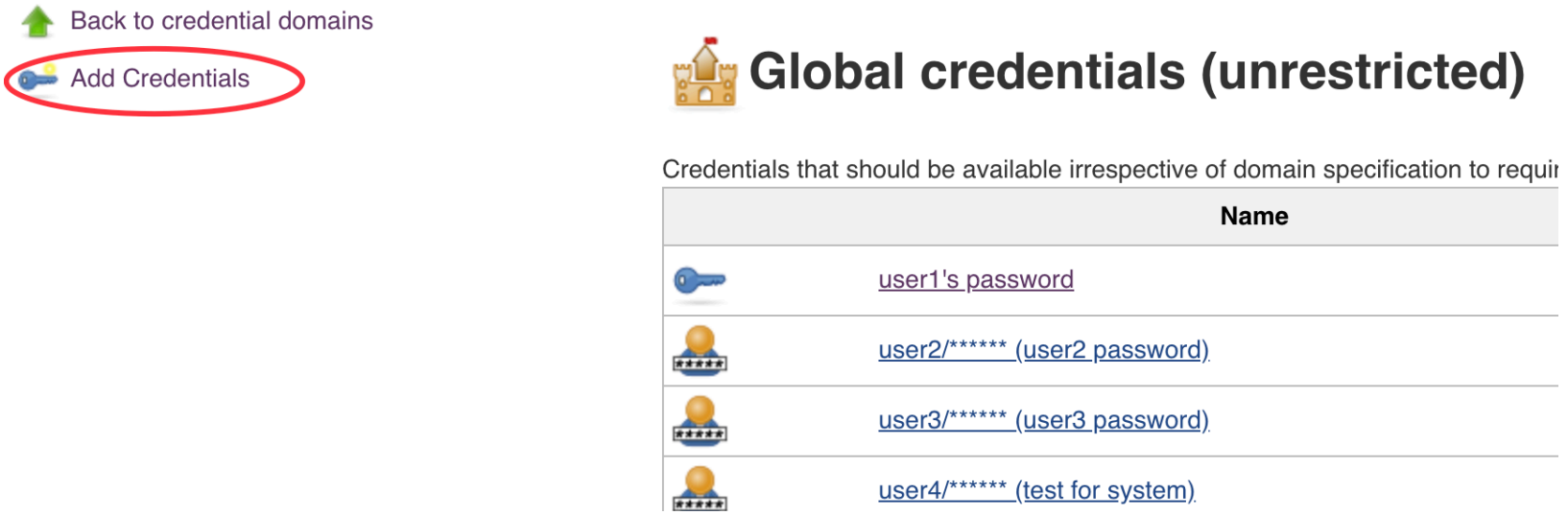
Credential Scope

- Global – credential用于Pipeline项目/其他项目，选择此项将crendential应用于Pipeline项目及其所有子对象
- System –credential用于Jenkins实例本身与系统管理功能（例如电子邮件认证，代理连接等）等交互。 选择此选项会将credential的应用于单个对象



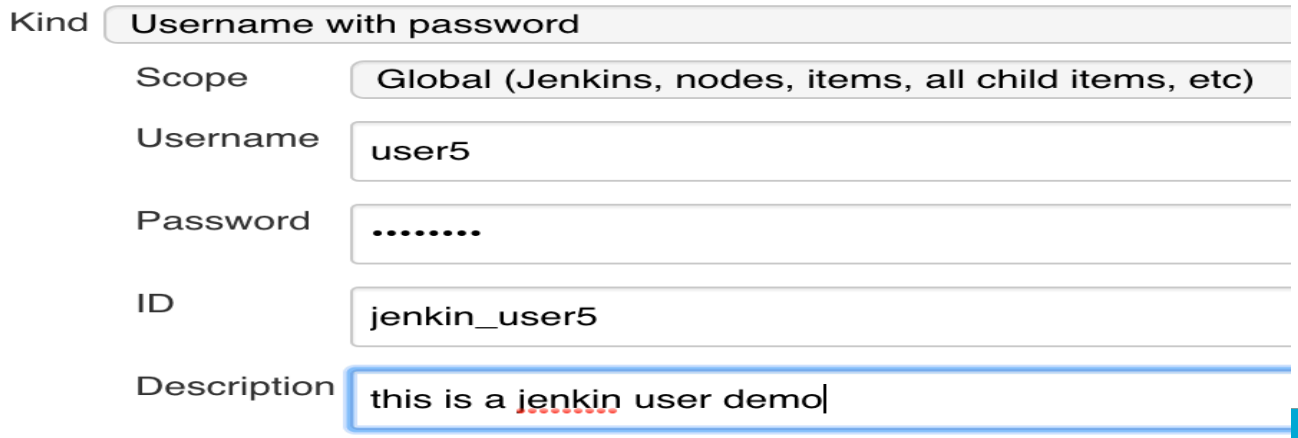
添加 Credential

- 确保登录到Jenkins (拥有 Credentials Create 权限的用户)
- 从Jenkins主页，点击左侧的 Credentials > System 。在 System 中， 点击 Global credentials (无限制) 链接，点击左侧的 添加 Credentials



添加 Credential

- 在 Scope 字段中，选择 Global 或者 System
- 将credentials 本身添加到所选择的credentials类型的相应字段中，如Username and password ，对应字段指定credential 的 Username 和 Password
- 在 ID 字段中，指定一个有意义的credential ID - 例如 jenkins-user-for-xyz-artifact-repository



逗点云创

Jenkins Pipeline 全解析系列

第五节： Pipeline 安全凭据处理（二讲）：安全凭据处理

声明式Pipeline Environment 指令

- 声明式Pipeline使用 environment 指令

  Environment 指令制定一个 键-值对序列，该序列定义环境变量，环境变量适用于所有步骤或者是特定某阶段的步骤， 取决于 environment 指令在流水线内的位置

- Environment 指令提供方法 credentials() ，该方法可以在 Jenkins 环境中用于通过标识符访问预定义的安全凭证

Secret text安全凭证

- 如下Pipeline代码演示了如何使用环境变量引用 secret 文本凭据示例
- 安全凭据已在 Jenkins 中配置各自的凭据 ID jenkins-aws-secret-key-id 和 jenkins-aws-secret-access-key

```
pipeline {
    agent {
        // 此处定义 agent 的细节
    }
    environment {
        AWS_ACCESS_KEY_ID     = credentials('jenkins-aws-secret-key-id')
        AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key')
    }
    stages {
        stage('Example stage 1') {
            steps {
                // ❶
            }
        }
        stage('Example stage 2') {
            steps {
                // ❷
            }
        }
    }
}
```

Secret text安全凭证注意事项

- 通过环境变量 $AWS_ACCESS_KEY_ID 和 $AWS_SECRET_ACCESS_KEY 来引用安全凭据
- 为了保持安全凭据的安全性和匿名性，如试图从Pipeline中显示凭据变量的值（如 echo $AWS_SECRET_ACCESS_KEY），Jenkins 只会返回“****”来降低机密信息被写到控制台输出和任何日志中的风险。凭据 ID 本身的任何敏感信息（如用户名）也会以“****”的形式返回到流水线运行的输出中
- 在该示例中，分配给两个 AWS_... 环境变量的凭据在整个流水线的全局范围内都可访问，然而，如果流水线中的 environment 指令被定义到一个特定的阶段，那么这些 AWS_... 环境变量就只能作用于该阶段的步骤中

Username and password（ 带密码的用户名 ）

- 带密码的用户名凭据被分配环给境变量，该安全凭据已在 Jenkins 中配置了凭据 ID，如该示例中的ID jenkins-bitbucket-common-creds
- 按照惯例，环境变量的变量名通常以大写字母指定，每个单词用下划线分割。注意 credentials() 方法所创建的带密码用户名的环境变量可用后缀 _USR 和 _PSW（即以下划线后跟三个大写字母的格式）用于引用相应的用户名和密码

带密码的用户名示例

- BITBUCKET_COMMON_CREDS - 包含一个以冒号分隔的用户名和密码，格式为 username:password
- BITBUCKET_COMMON_CREDS_USR - 附加的一个仅包含用户名部分的变量
- BITBUCKET_COMMON_CREDS_PSW - 附加的一个仅包含密码部分的变量

```
pipeline {
    agent {
        // 此处定义 agent 的细节
    }
    stages {
        stage('Example stage 1') {
            environment {
                BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-common-creds')
            }
            steps {
                // ❶
            }
        }
        stage('Example stage 2') {
            steps {
                // ❷
            }
        }
    }
}
```

带密码的用户名示例注意事项

- 在该流水线示例中，分配给三个 COMMON_BITBUCKET_CREDS... 环境变量的凭据仅作用于 Example stage 1，所以在 Example stage 2 阶段的步骤中这些凭据变量不可用。然而，如果马上把流水线中的 environment 指令移动到 pipeline 块中（正如上面的 Secret 文本流水线示例一样），这些 COMMON_BITBUCKET_CREDS... 环境变量将应用于全局并可以在任何阶段的任何步骤中使用
- 为了维护凭据的安全性和匿名性，如果任务试图从流水线中显示这些凭据变量的值，那么上面的 Secret 文本 描述的行为也同样适用于这些带密码的用户名凭据变量类型，即输出以 “****”形式表示

Jenkins Pipeline 全解析系列

第六节： Pipeline 参数输入

声明式Pipeline参数处理

- 在声明式Pipeline中，通过Parameters 指令提供为用户可在Pipeline中使用的参数列表
- 参数的值可通过 Params 对象提供给Pipeline中使用

如在 Jenkinsfile 中配置了名为 “Greeting” 的字符串参数， 可以通过 ${params.Greeting} 访问该参数

Required	No
Parameters	*None*
Allowed	Only once, inside the `pipeline` block.

声明式Pipeline参数类型

- **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: ‘’) }
- **Text文本类型参数**，可以包含多行：例如： parameters { text(name: 'DEPLOY_TEXT', defaultValue: 'One\nTwo\nThree\n', description: '') }

声明式Pipeline参数类型

- **choice 选项类型参数**：例如: parameters { choice(name: ‘CHOICES’, choices: [‘one’, ‘two’, ‘three’], description: ‘’) }
- **Password 密码参数**：例如: parameters { password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'A secret password') }

声明式Pipeline参数示例

```
pipeline {
    agent any
    parameters {
        string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

        text(name: 'BIOGRAPHY', defaultValue: '', description: 'Enter some information about the person')

        booleanParam(name: 'TOGGLE', defaultValue: true, description: 'Toggle this value')

        choice(name: 'CHOICE', choices: ['One', 'Two', 'Three'], description: 'Pick something')

        password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'Enter a password')
    }
```

```
    stages {
        stage('Example') {
            steps {
                echo "Hello ${params.PERSON}"

                echo "Biography: ${params.BIOGRAPHY}"

                echo "Toggle: ${params.TOGGLE}"

                echo "Choice: ${params.CHOICE}"

                echo "Password: ${params.PASSWORD}"
            }
        }
    }
```

声明式Pipeline参数示例

Pipeline pipeline_parameters

This build requires parameters:

PERSON: DouDianYun
Who should I say hello to?

BIOGRAPHY: this is a test
Enter some information about the person

TOGGLE: ☑
Toggle this value

CHOICE: Two
Pick something

PASSWORD: ••••••••••
Enter a password

Build

Console Output

```
Started by user jenkin
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/pipeline_parameters
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Example)
[Pipeline] echo
Hello DouDianYun
[Pipeline] echo
Biography: this is a test
[Pipeline] echo (hide)
Toggle: true
[Pipeline] echo
Choice: Two
[Pipeline] echo
Password: SECRET
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Jenkins Pipeline 全解析系列

第七节： Pipeline 故障处理

声明式Pipeline故障处理

- 声明式 Pipeline 默认通过 Post 指令 支持故障处理
- 支持不同的 “Post Condition”，比如：always、unstable、success、failure 和 changed

```
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                sh 'make check'
            }
        }
    }
    post {
        always {
            junit '**/target/*.xml'
        }
        failure {
            mail to: team@example.com, subject: 'The Pipeline failed :('
        }
    }
}
```

脚本式Pipeline故障处理

- 脚本化的Pipeline依赖于 Groovy 的内置的 try/catch/finally 语义来处理流水线运行期间的故障

```
node {
    /* .. snip .. */
    stage('Test') {
        try {
            sh 'make check'
        }
        finally {
            junit '**/target/*.xml'
        }
    }
    /* .. snip .. */
}
```

Post 指令

- Post 定义一个或多个steps ，Steps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
- 根据Pipeline或Stage的完成状态，这些条件块允许在 post 部分的步骤的执行

Required	No
Parameters	*None*
Allowed	In the top-level `pipeline` block and each `stage` block.

Post Conditions

- Always:无论流水线或阶段的完成状态如何，都允许在 post 部分运行该步骤
- Changed:只有当前流水线或阶段的完成状态与它之前的运行不同时，才允许在 post 部分运行该步骤
- Failure:只有当前流水线或阶段的完成状态为“failure”，才允许在 post 部分运行该步骤, 通常web UI是红色
- Success:只有当前流水线或阶段的完成状态为“success”，才允许在 post 部分运行该步骤, 通常web UI是蓝色或绿色
- Unstable:只有当前流水线或阶段的完成状态为“unstable”，才允许在 post 部分运行该步骤, 通常由于测试失败,代码违规等造成。通常web UI是黄色
- Aborted:只有当前流水线或阶段的完成状态为“aborted”，才允许在 post 部分运行该步骤, 通常由于流水线被手动的aborted。通常web UI是灰色

声明式Pipeline故障处理

```
pipeline {
    // no agent required to run here. All steps run in flyweight executor on Master
    agent none

    stages {
        stage("foo") {
            steps {
                echo "hello"
            }
        }
    }
    post {
    /*
     * These steps will run at the end of the pipeline based on the condition.
     * Post conditions run in order regardless of their place in pipeline
     * 1. always - always run
     * 2. changed - run if something changed from last run
     * 3. aborted, success, unstable or failure - depending on status
     */
```

```
always {
    echo "I AM ALWAYS first"
}
changed {
    echo "CHANGED is run second"
}
aborted {
  echo "SUCCESS, FAILURE, UNSTABLE, or ABORTED are exclusive of each other"
}
success {
    echo "SUCCESS, FAILURE, UNSTABLE, or ABORTED runs last"
}
unstable {
  echo "SUCCESS, FAILURE, UNSTABLE, or ABORTED runs last"
}
failure {
    echo "SUCCESS, FAILURE, UNSTABLE, or ABORTED runs last"
}
```

Jenkins Pipeline 全解析系列

第八节： Pipeline 片段生成器

Pipeline 片段生成器（ Snippet Generator ）

- 片段生成器：Jenkins内置的Pipeline“代码生成器”实用工具
- 可以为特定 Step 步骤生成该Step Pipeline 的有效代码，发现特定插件提供的步骤等，或为特定步骤尝试不同的参数
- Snippet Generator将动态填充Jenkins可用的步骤（Step）列表，该步骤列表取决于已安装的插件

调用Pipeline 片段生成器

- 访问${YOUR_JENKINS_URL}/pipeline-syntax 链接，导航到Pipeline Syntax链接
- 点击Snippet Generator链接，在“样品步骤”（Sample Step）下拉菜单中选择所需的步骤，填写必要的参数来配置所选步骤
- 单击“生成Pipeline脚本”以生成片段，可将其复制到Pipeline中

Pipeline 片段生成器示例

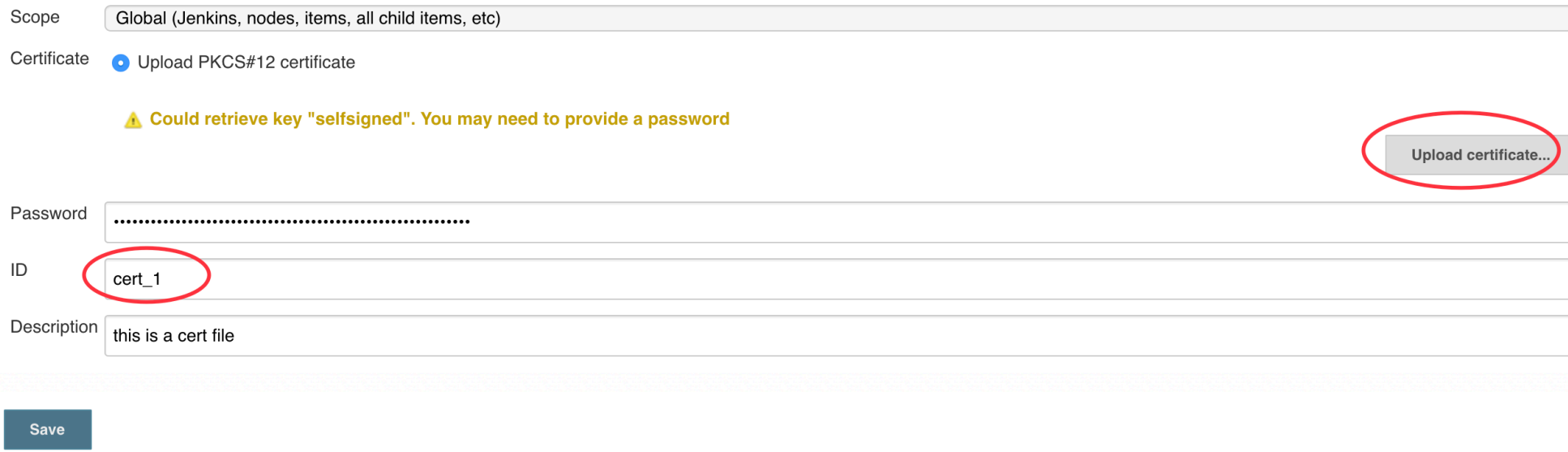
- 利用Snippet Generator生成引用安全凭据Certificate （证书）Pipeline 代码片段
- 首先生成PKCS#12 证书， 如下命令：

keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass password -validity 360 -keysize 2048

keytool -importkeystore -srckeystore keystore.jks -srcstorepass password -destkeystore my.p12 -srcstoretype JKS -deststoretype PKCS12 -deststorepass password

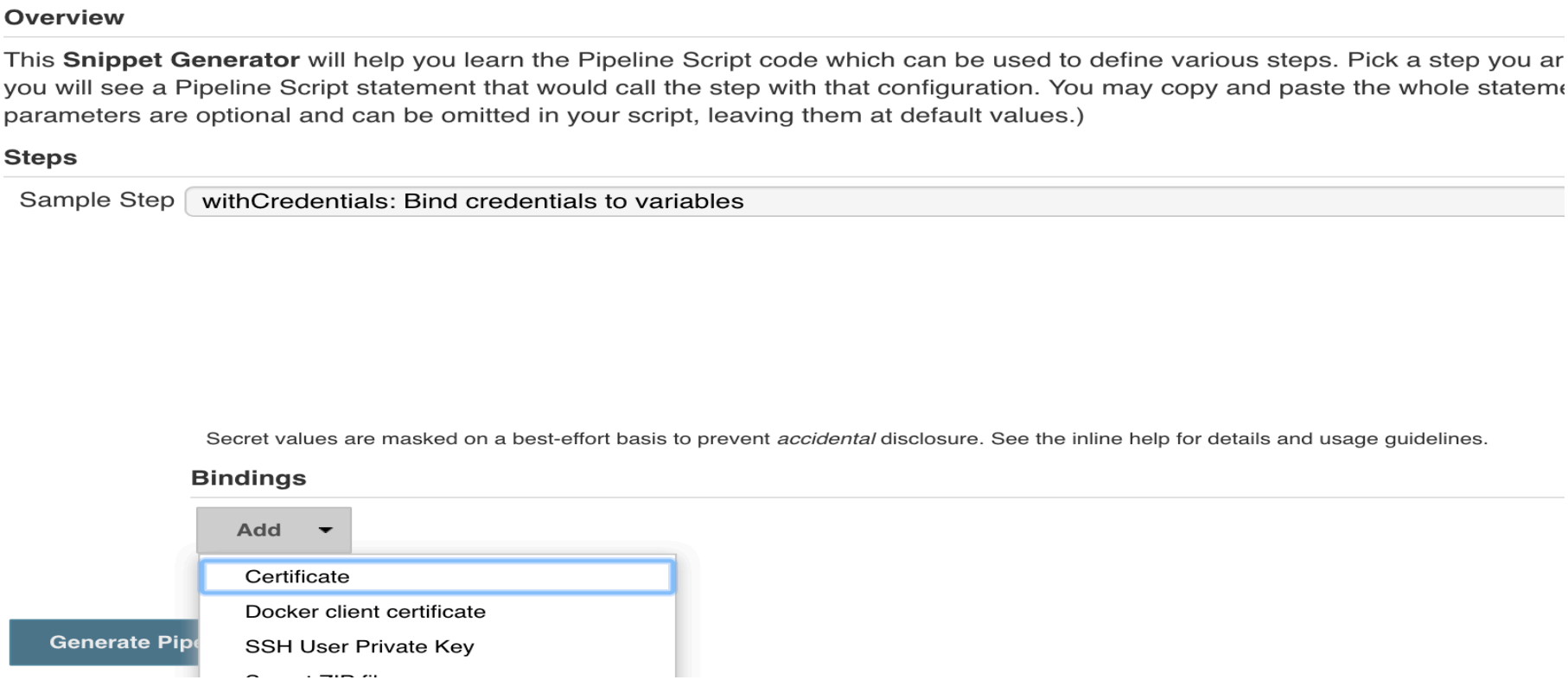
Jenkins定义证书安全凭据

- 上传生成的P12证书，提供正确的Keystore 密码



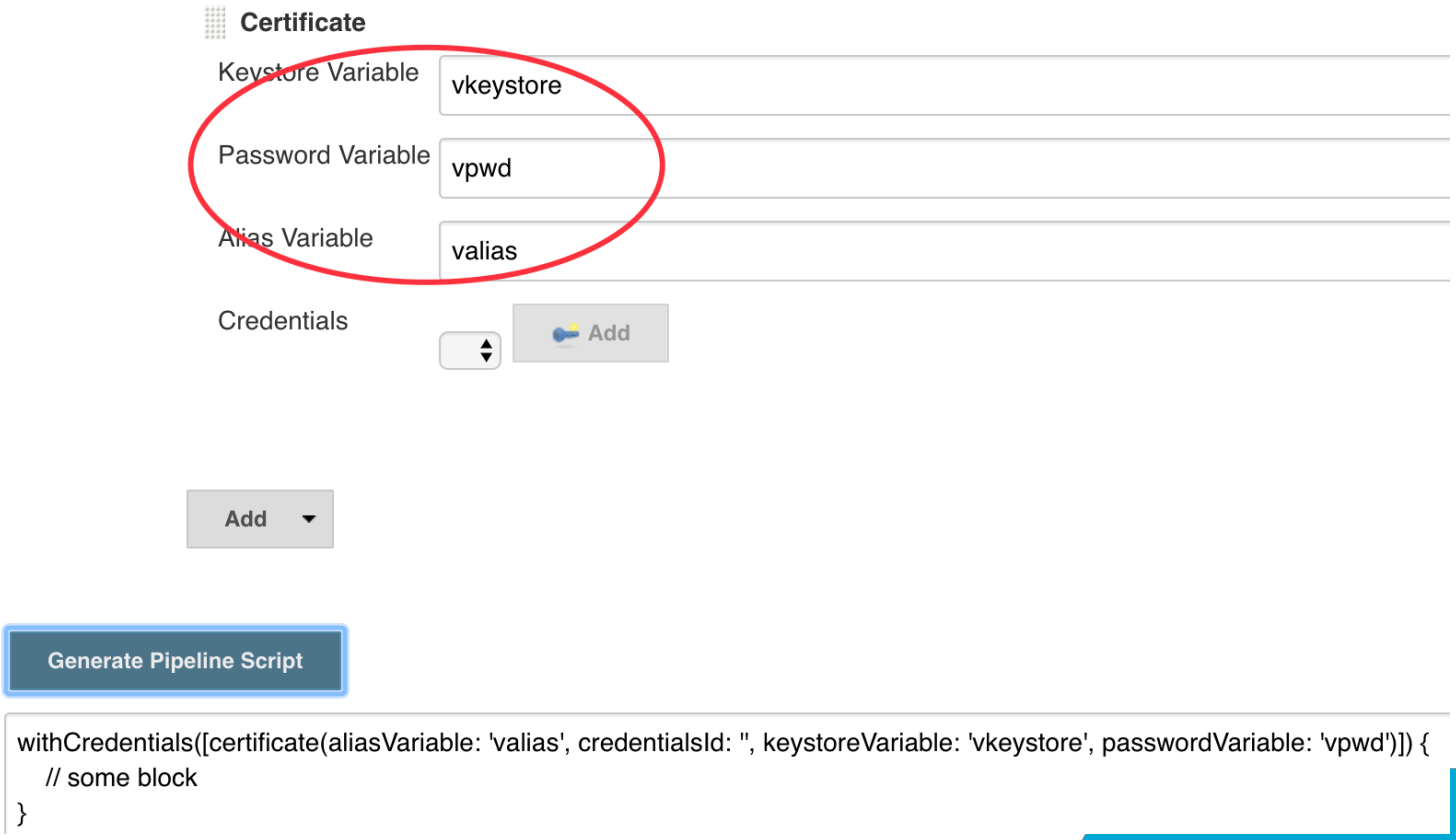
Pipeline 片段生成器示例

- Sample Step 选择 WithCredentials,选择“证书”安全凭据



Pipeline 片段生成器示例

- 输入必要的字段信息，点击生成Pipeline 脚本 按钮



复制并使用生成的Pipeline脚本

- 复制并使用生成的Pipeline脚本片段，注意替换相应的安全凭证ID，如下所示：

```
        }
    }

stage("cert") {
        steps {

withCredentials([certificate(aliasVariable: 'valias', credentialsId: 'cert_1', keys

    sh 'echo "keystore is $vkeystore"'

}

        echo "hello2"
    }
```

Use Groovy Sandbox

```
Started by user jenkin
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/cert_example
[Pipeline] {
[Pipeline] stage
[Pipeline] { (foo)
[Pipeline] echo
hello
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { ("cert")
[Pipeline] withCredentials
Masking supported pattern matches of $valias or $vpwd or $vkeystore
[Pipeline] {
[Pipeline] sh
+ echo 'keystore is ****'
keystore is ****
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] echo
hello2
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

逗点云创

Jenkins Pipeline 全解析系列

第九节： 声明式Pipeline语法之区块(一讲)：Agent 区块

Agent区块

- Agent:Jenkins为Pipeline或Stage分配的执行空间和工作空间
- Agent 必须在 pipeline 块的顶层被定义, 但 stage 级别的使用是可选的

Required	Yes
Parameters	Described below
Allowed	In the top-level `pipeline` block and each `stage` block.

Agent区块之参数

为了各种Pipeline适用场景, agent 支持不同类型的参数，这些参数应用在‘pipeline’块的顶层, 或 stage 指令内部

- Any：在任何可用的Agent上执行流水线或阶段。例如: agent any
- None:当在 pipeline 块的顶层设置该参数，不会为整个Pipeline运行分配全局Agent，每个Stage部分都需要包含自己的Agent部分
- Label：在提供了标签的 Jenkins 环境中可用的Agent上执行流水线或Stage。 例如: agent { label 'my-defined-label' }

Agent区块之参数

- Node: agent { node { label ‘labelName’ } } 和 agent { label ‘labelName’ } 作用一样, 但Node 可以加其他选项 (比如 customWorkspace)
- Docker：基于Docker的Pipeline，使用给定的容器执行Pipeline或Stage。该容器将在预置的 node上，或在匹配可选定义的`label` 参数上。Docker 可接受 Args 参数，参数可包含直接传递到 docker run 调用的参数, 以及 alwaysPull 选项, 该选项强制 docker pull ，即使镜像名称已经存在，例如：

```
agent {
    docker {
        image 'maven:3-alpine'
        label 'my-defined-label'
        args  '-v /tmp:/tmp'
    }
}
```

Agent区块之参数

- Dockerfile：执行Pipeline或Stage，使用从源代码库包含的 Dockerfile 构建的容器。如在别的目录下构建 Dockerfile，使用 dir 选项: agent { dockerfile {dir 'someSubDir' } } 。此外，还有filename 选项用于指定文件名；使用 additionalBuildArgs 选项传递Docker构建参数等，如：

```
agent {
    // Equivalent to "docker build -f Dockerfile.build --build-arg version=1.0.2 ./build/
    dockerfile {
        filename 'Dockerfile.build'
        dir 'build'
        label 'my-defined-label'
        additionalBuildArgs  '--build-arg version=1.0.2'
    }
}
```

Agent区块之选项

- Label：该标签用于表明在哪个节点上运行Pipeline或 stage, 该选项对 node, docker 和 dockerfile 可用, `node`参数 Agent要求必须选择该选项
- customWorkspace: 在自定义工作区运行 Pipeline或 stage，可以是一个相对路径, 或绝对路径，该选项对 node, docker 和 dockerfile 有用，如:

```
agent {
    node {
        label 'my-defined-label'
        customWorkspace '/some/other/path'
    }
}
```

Agent区块之示例

```
pipeline {
    agent none (1)
    stages {
        stage('Example Build') {
            agent { docker 'maven:3-alpine' } (2)
            steps {
                echo 'Hello, Maven'
                sh 'mvn --version'
            }
        }
        stage('Example Test') {
            agent { docker 'openjdk:8-jre' } (3)
            steps {
                echo 'Hello, JDK'
                sh 'java -version'
            }
        }
    }
}
```

1. 在Pipeline顶层定义 agent none, 确保 Executor 没有被分配， 使用 agent none 也会强制 stage 部分包含自身的 agent 定义
2. 使用镜像在一个新建的容器中执行该Stage的该步骤
3. 使用一个与之前Stage不同的镜像在一个新建的容器中执行该Stage的步骤

逗点云创

Jenkins Pipeline 全解析系列

第九节： 声明式Pipeline语法之区块（二讲）：Post/Stages/Steps 介绍

Post 区块

- Post 定义一个或多个steps ，这些Steps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
- 根据Pipeline或Stage的完成状态，这些条件块允许在 post 部分的步骤的执行

Required	No
Parameters	*None*
Allowed	In the top-level `pipeline` block and each `stage` block.

Stages区块

- 包含一个或多个 Stage 指令, Stages 区块是Pipeline定义大部分任务的地方
- 通常 Stages 至少包含一个 stage 指令用于定义持续交付过程的各个阶段,比如构建, 测试和部署等

Required	Yes
Parameters	*None*
Allowed	Only once, inside the `pipeline` block.

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

Steps区块

- Steps 区块用于在给定的 Stage 区块中执行定义的一个或多个steps

Required	Yes
Parameters	*None*
Allowed	Inside each `stage` block.

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

逗点云创

Jenkins Pipeline 全解析系列

第十节： 声明式Pipeline语法之指令（一讲）：Options介绍

Options指令常用选项

Options 指令在Pipeline中配置特定于该Pipeline的选项。常见支持的选项如下：

- buildDiscarder：为最近的Pipeline运行的保存指定数量的运行结果和控制台输出。例如：options { buildDiscarder(logRotator(numToKeepStr: '1')) }
- disableConcurrentBuilds：不允许同时执行Pipeline。 用来防止同时访问共享资源等。 例如：options { disableConcurrentBuilds() }

Required	No
Parameters	*None*
Allowed	Only once, inside the `pipeline` block.

Options指令常用选项

- skipDefaultCheckout: 在”agent” 区块，跳过从源代码控制管理系统中检出代码。例如: options { skipDefaultCheckout() }
- skipStagesAfterUnstable：如构建状态变得UNSTABLE，跳过该Stage。如: options { skipStagesAfterUnstable() }
- checkoutToSubdirectory：在工作空间的子目录中自动地执行源代码控制检出。例如: options { checkoutToSubdirectory('foo') }

Options指令常用选项

- Timeout

  设置Pipeline运行的超时时间, 超过该设置后，Jenkins将中止Pipeline。例如: options { timeout(time: 1, unit: 'HOURS') }

- Retry

  在失败时, 重新尝试指定次数的Pipeline

  For example: options { retry(3) }

- Timestamps:

  Pipeline生成的控制台输出附带时间戳， 例如: options { timestamps() }

Options指令示例

- 指定timeout时间为一小时，在此之后，Jenkins 将中止Pipeline运行

```
pipeline {
    agent any
    options {
        timeout(time: 1, unit: 'HOURS')
    }
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

Stage Options指令常用选项

Stage 的 options 指令类似于Pipeline根目录上的 options 指令，只支持有限的选项：

- skipDefaultCheckout：跳过从源代码控制中检出源代码。例如: options { skipDefaultCheckout() }
- Timeout：设置此阶段的超时时间, 在此之后， Jenkins 会终止该阶段。 例如: options { timeout(time: 1, unit: 'HOURS') }
- Retry：失败时, 重试此阶段指定次数。 例如: options { retry(3) }
- Timestamps：此阶段生成的控制台输出附带时间戳。例如: options { timestamps() }

Stage Options指令示例

- 指定 Example 阶段的执行超时时间，在此之后，Jenkins 将中止流水线运行

```
pipeline {
    agent any
    stages {
        stage('Example') {
            options {
                timeout(time: 1, unit: 'HOURS')
            }
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

逗点云创

Jenkins Pipeline 全解析系列

第十节： 声明式Pipeline语法之指令（二讲）：Triggers介绍

Triggers（触发器）简介

- Triggers 指令定义Pipeline被触发的自动化方法，可用的触发器有： cron, pollSCM 和 upstream

Required	No
Parameters	*None*
Allowed	Only once, inside the `pipeline` block.

触发器类别

- Cron：接收 cron 样式的字符串来定义要重新触发Pipeline的常规时间间隔 ,比如: triggers { cron(‘H */4 * * 1-5’) }
- pollSCM：接收 cron 样式的字符串来定义一个固定的时间间隔，根据间隔，Jenkins 会检查新的源代码更新。如果存在更改, Pipeline就会被重新触发。例如: triggers { pollSCM('H */4 * * 1-5') }
- Upstream: 接受逗号分隔的工作任务字符串和条件。 当字符串中的任何作业结束时并且触发条件满足时，Pipeline被重新触发。例如: triggers { upstream(upstreamProjects: 'job1,job2', threshold: hudson.model.Result.SUCCESS) }

Cron语法

- Jenkins cron语法遵循cron实用程序的语法，每行包含5个用TAB或空格分隔的字段，如下：

MINUTE	HOUR	DOM	MONTH	DOW
Minutes within the hour (0–59)	The hour of the day (0–23)	The day of the month (1–31)	The month (1–12)	The day of the week (0–7) where 0 and 7 are Sunday.

要为一个字段指定多个值，可以使用以下运算符：

- *指定所有有效值
- M-N指定值的范围
- M-N / X或* / X以X的间隔逐步达到指定范围或整个有效范围
- A，B，...，Z枚举多个值

如每15分钟，触发一次： triggers {cron ('H / 15 * * * *') }

Cron示例

- 每隔两分钟，触发Pipeline 执行

```
pipeline {
    agent any
    triggers {
        cron('H/2 * * * *')
    }
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

Console Output

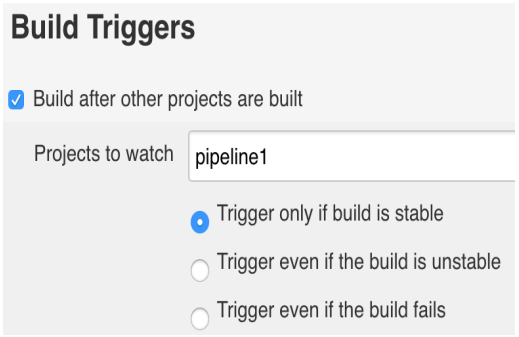
```
Started by user jenkin
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/trigger-demo
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Example)
[Pipeline] echo
Hello World
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Upstream 触发条件

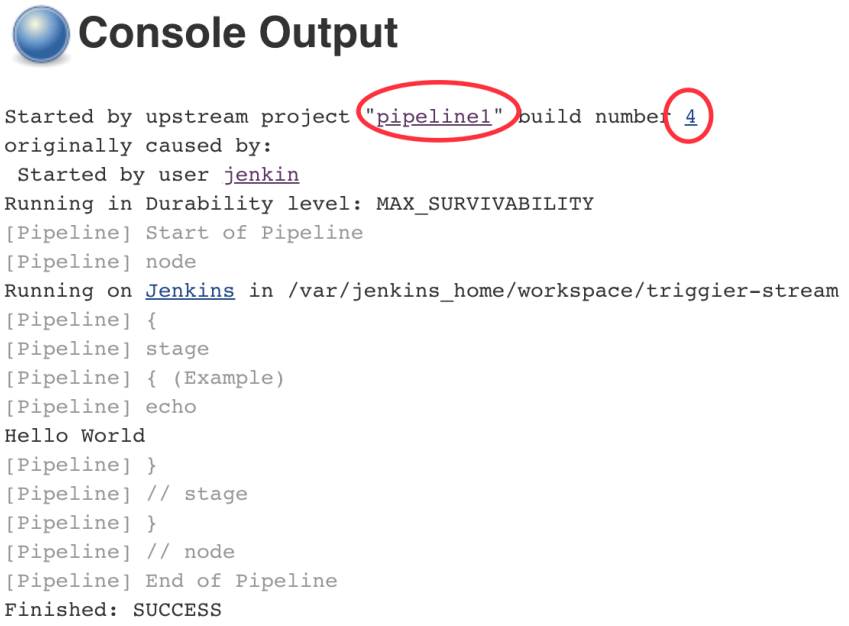
static Result	**Result.ABORTED** The build was manually aborted.
static Result	**Result.FAILURE** The build had a fatal error.
static Result	**Result.NOT_BUILT** The module was not built.
protected Result	**Run.result** The build result.
static Result	**Result.SUCCESS** The build had no errors.
static Result	**Result.UNSTABLE** The build had some errors but they were not fatal.

Upstream示例

- 当Pipeline1作业正常结束时，该Pipeline将会被触发




```
pipeline {
    agent any
  triggers{
upstream(upstreamProjects: 'pipeline1', threshold: hudson.model.Result.SUCCESS)
}
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
```



逗点云创

Jenkins Pipeline 全解析系列

第十节： 声明式Pipeline语法之指令（三讲）：tools/input 指令介绍

Tools指令介绍

- 定义在Pipeline中可以用的Tools，支持的Tools： JDK, Ant，Maven,docker 等
- Tools须在Jenkins中的Manage Jenkins→全局工具配置事先配置

Required	No
Parameters	*None*
Allowed	Inside the `pipeline` block or a `stage` block.

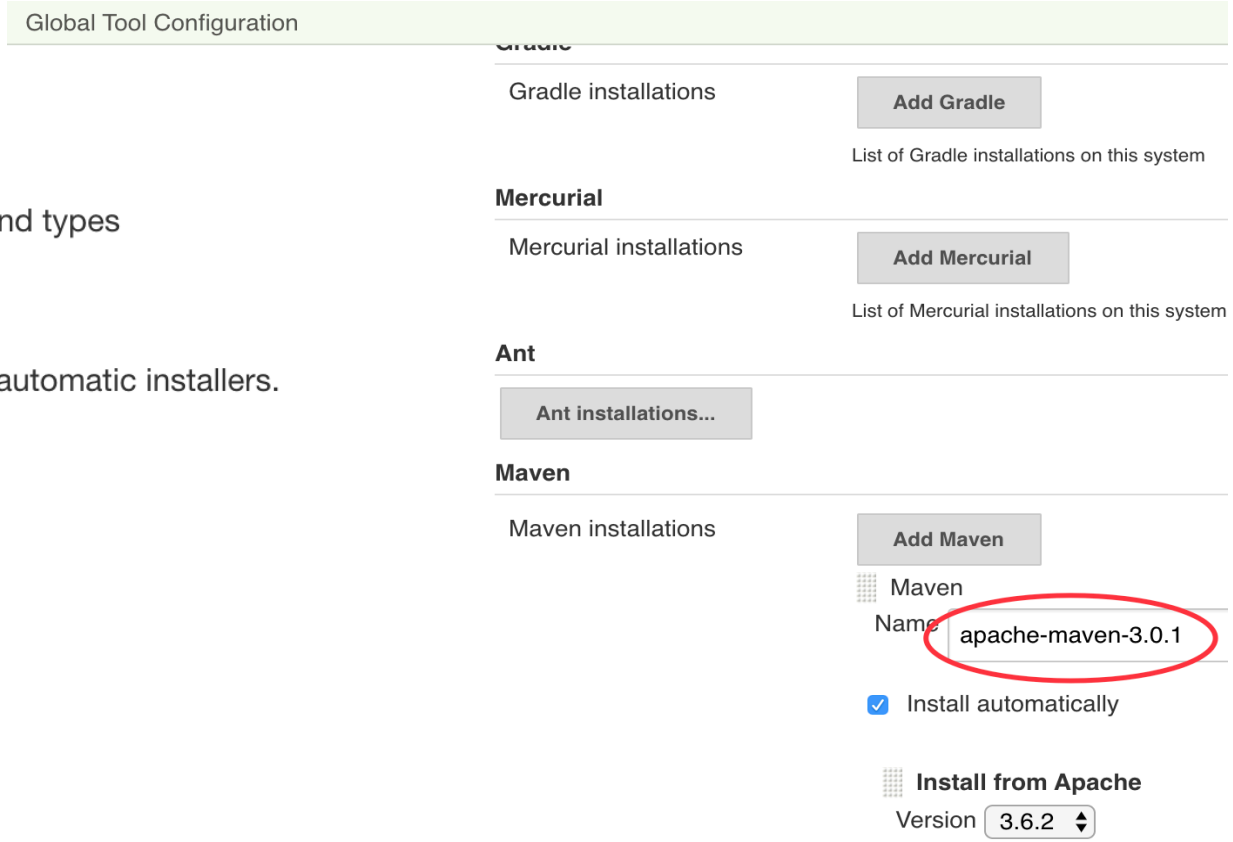
全局工具配置

Configure Credentials

Configure the credential providers and types

Global Tool Configuration

Configure tools, their locations and automatic installers.



Tools示例

```
pipeline {
    agent any
    tools {
        maven 'apache-maven-3.0.1'
    }
    stages {
        stage('Example') {
            steps {
                sh 'mvn --version'
                echo 'tool example'
            }
        }
```

```
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] sh
+ mvn --version
Apache Maven 3.6.2 (40f52333136460af0dc0d7232c0dc0bcf0d9e117; 2019-08-27T15:06:16Z)
Maven home: /var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/apache-maven-3.0.1
Java version: 1.8.0_212, vendor: IcedTea, runtime: /usr/lib/jvm/java-1.8-openjdk/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.9.184-linuxkit", arch: "amd64", family: "unix"
[Pipeline] echo
tool example
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Input指令

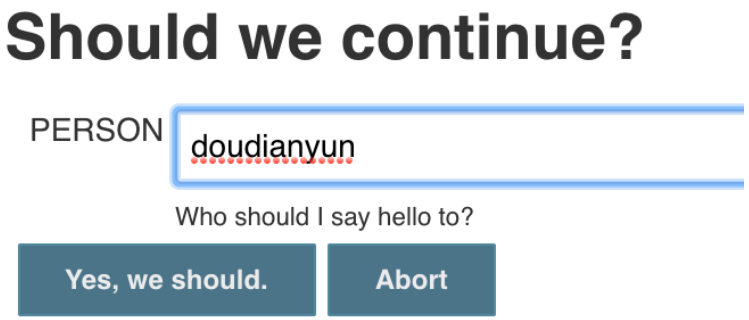
Input 指令允许提示输入给Pipeline，如Input 被提供参数，stage 将会继续。Input 提交的任何参数可在相应的stage中引用

- Message：必需的，用户提交 Input 时展现给用户
- Id：input 的可选标识符， 默认为 stage 名称
- Ok：input表单上的“ok” 按钮的可选文本
- Submitter：可选以逗号分隔的用户列表，默认允许任何用户
- Parameters：提示参数列表

Input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."
            }
        }
    }
}
```

Input示例




```
[Pipeline] stage
[Pipeline] { (Example)
[Pipeline] input (hide)
Input requested
Approved by jenkin
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
Hello, doudianyun, nice to meet you.
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

逗点云创

Jenkins Pipeline 全解析系列

第十节： 声明式Pipeline语法之指令（四讲）：When 指令

When 指令

- When 指令允许Pipeline根据给定的条件决定是否应该执行Stage
- When 指令必须至少包含一个条件，如果 when 指令包含多个条件，所有的子条件必须返回True，Stage才能执行
- 使用Not, allOf, 或 anyOf 嵌套条件可构建更复杂的条件结构

Required	No
Parameters	*None*
Allowed	Inside a `stage` directive

When 内置条件

- Branch：当正在构建的分支与模式给定的分支匹配时，执行阶段, 例如: when { branch ‘master’ }，注意，只适用于多分支 Pipeline

```
pipeline {
    agent any
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                branch 'production'
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
```

When 内置条件

- Environment:当指定的环境变量是给定的值时，执行步骤，例如: when { environment name: 'DEPLOY_TO', value: 'production' }

```
pipeline {
  agent any

  environment {
      // This returns 0 or 1 depending on whether build number is even or odd
      FOO = "${currentBuild.getNumber() % 2}"
  }

  stages {
    stage("Hello") {
      steps {
        echo "Hello"
      }
    }
    stage("Evaluate FOO") {
      when {
        // stage won't be skipped as long as FOO == 0, build number is even
        environment name: "FOO", value: "0"
      }
      steps {
        echo "World"
      }
    }
  }
}
```

When 内置条件

- Expression:当指定的Groovy表达式为true时，执行阶段,例如: when { expression { return params.DEBUG_BUILD } }

```
pipeline {
  agent any
  stages {
    stage("Hello") {
      steps {
        echo "Hello"
      }
    }
    stage("Always Skip") {
      when {
        // skip this stage unless the expression evaluates to 'true'
        expression {
          echo "Should I run?"
          return false
        }
      }
      steps {
        echo "World"
      }
    }
  }
}
```

When 内置条件

```
1  pipeline {
2    agent any
3
4    stages {
5      stage("Hello") {
6        steps {
7          echo "Hello"
8        }
9      }
10     stage("Branch Test") {
11       when {
12         // skip this stage unless branch is NOT master
13         not {
14           branch "master"
15         }
16       }
17       steps {
18         echo "World"
19         echo "Heal it"
20       }
21     }
22   }
23 }
```

- Not:当嵌套条件是false时，执行阶段，必须包含一个条件，如: when { not { branch 'master' } }

When 内置条件

- allOf:当所有的嵌套条件都为TRUE时，执行该阶段,必须包含至少一个条件，例如: when { allOf { branch 'master'; environment name: 'DEPLOY_TO', value: 'production' } }

```
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'
            }
        }
    }
}
```

When 内置条件

- anyOf：当至少有一个嵌套条件为真时，执行阶段,必须包含至少一个条件，如: when { anyOf { branch 'master'; branch 'staging' } }

```
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'
            }
        }
    }
}
```

When beforeAgent内置条件

```
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'
            }
        }
    }
}
```

- 默认情况下，如果定义了某个阶段的Agent，在进入该Stage的agent 后，该 stage 的 when 条件将会被评估。可以通过在 when 块中指定 beforeAgent 选项来更改此选项。 如果 beforeAgent 被设置为 true，那么就会首先对 when 条件进行评估，并且只有在 when 条件验证为真时才会进入 agent

When beforeInput内置条件

```
pipeline {
    agent none
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                beforeInput true
                branch 'production'
            }
            input {
                message "Deploy to production?"
                id "simple-input"
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
```

- 可通过在when条件块中指定beforeInput选项来改变是否在Input指令执行之前评估When条件块还是之后。如将beforeInput设置为true，则将首先评估when条件，并且仅当when条件评估为true时才执行Input指令

When示例

```
pipeline {
  agent any

  environment {
      // This returns 0 or 1 depending on whether build number is even or odd
      FOO = "${currentBuild.getNumber() % 2}"
  }

  stages {
    stage("Hello") {
      steps {
        echo "Hello"
      }
    }
    stage("Evaluate FOO") {
      when {
        // stage won't be skipped as long as FOO == 0, build number is even
        environment name: "FOO", value: "0"
      }
      steps {
        echo "World"
      }
    }
  }
}
```

```
Started by user jenkin
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/when
[Pipeline] { (hide)
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Hello)
[Pipeline] echo
Hello
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Evaluate FOO)
Stage "Evaluate FOO" skipped due to when conditional
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

```
Started by user jenkin
Running in Durability level: M
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jen
[Pipeline] {
[Pipeline] withEnv (hide)
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Hello)
[Pipeline] echo
Hello
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Evaluate FOO)
[Pipeline] echo
World
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

感谢大家

逗点云创

逗点云创专注于计算机领域和相关技术领域（大数据集成，云计算和人工智能等）的知识分享和线上线下的知识推广

联系我们：www.doudianyun.com

contact@doudianyun.com