**持续集成(CI) / 持续交付 (CD)**

# 指南

## 1、入门指南

### 1.1 安装

下载并运行Jenkins：

下载WAR文件（Jenkins的Web应用程序ARchive（WAR）文件版本可以安装在任何支持Java的操作系统或平台上。）

打开终端，进入到下载目录

启动命令 java -jar jenkins.war --httpPort=8080

后台启动 nohup java -jar jenkins.war --httpPort=8080 &

启动后会生成一个nohup.out输出，可以tail -f nohup.out实时查看日志。

打开浏览器进入链接 http://localhost:8080

重启Jenkins：

http://127.0.0.1:8080/restart

### 1.2 安装后设置向导

此安装向导会引导您完成几个快速“一次性”步骤来解锁Jenkins，使用插件对其进行自定义，并创建第一个可以继续访问Jenkins的管理员用户。

1）解锁Jenkins

第一次访问新的Jenkins实例时，系统会要求您使用自动生成的密码对其进行解锁。浏览到 http://localhost:8080（或安装时为Jenkins配置的任何端口），并等待解锁Jenkins页面出现。

2）在解锁Jenkins页面上，将此密码粘贴到管理员密码字段中。

3）自定义jenkins插件

解锁Jenkins之后，在Customize Jenkins页面内，您可以安装任何数量的有用插件作为初始步骤的一部分。

两个选项可设置：

* 安装建议的插件
* 选择要安装的插件

4）创建第一个管理员用户

### 1.3 创建第一个Pipeline

#### 什么是Jenkins Pipeline？

Jenkins Pipeline是一套插件，将持续交付的实现和实施集成到Jenkins中。

持续交付Pipeline自动化的表达了这样一种流程：将基于版本控制管理的软件持续的交付到您的用户和消费者手中。

Jenkins Pipeline提供了一套可扩展的工具，用于将“简单到复杂”的交付流程实现为“可持续交付即代码”。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。

Pipeline和Jenkins的更多相关信息，参考用户手册中的相关链接[Pipeline](https://jenkins.io/doc/book/pipeline)和[Using a Jenkinsfile](https://jenkins.io/doc/book/pipeline/jenkinsfile)。

#### 什么是流水线？

Jenkins的流水线简单认知：它可以把若干可通用的Shell脚本像胶水一样连接起来，这只是流水线的冰山一角。

Jenkins流水线的两种形式：在升级2.x版本之前，流水线只有一种脚本式流水线（Script Pipeline），升级2.x版本后，新增了声明式流水线（Declarative Pipeline）。

什么是Jenkins：无论哪种流水线，都需要为流水线定义流水线脚本，称之为Jenkinsfile，使用Groovy语法写成。

#### 脚本式流水线

1. 特征

在脚本化流水线语法中，一个或多个node块在整个流水线中执行核心工作。

1. 限制

虽然这不是脚本化流水线语法的强制性要求，但它限制了你的流水线在node块内只能做两件事：

* 需要通过在Jenkins队列中添加一个项来调度块中包含的步骤。节点上的执行器一空闲，该步骤就会运行。
* 创建一个工作区（特定为特定流水间建立的目录），其中工作可以在从源代码控制检出的文件上完成。

1. 基本语法

Jenkinsfile（脚本式流水线）

node { //需要有一个或多个node节点表示一系列操作

stage('Build') { //每个stage表示一个步骤

// Build步骤内容

}

stage('Test') {

// Test步骤内容

}

stage('Deploy') {

// Deploy步骤内容

}

}

#### 声明式流水线

1. 特征

声明式流水线最外层Jenkinsfile脚本节点时pipeline

1. 与脚本式流水线不同点

* 相比脚本化的流水线语法，它提供更丰富的语法特性
* 为了使编写和读取流水线代码更容易而设计的。

1. 基础语法

Jenkinsfile（声明式流水线）

pipeline { // pipeline是声明式流水线的一种特定语法，在块内定义了整个流水线的所有内容

agent any // agent是声明式流水线的一种特定语法，它指示 Jenkins 为整个流水线分配一个执行器 (在节点上)和工作区，等效于脚本式流水线node块

stages { // 所有流程（状态）的外层块，仅有一个

stage('Build') { //每个stage为一流程，与脚本式基本一致，每个stage可以定义名称

steps { //步骤块，内部包含具体操作

sh 'make' // sh操作，其引号间的文字会当成shell直接执行

}

}

stage('Test'){

steps {

sh 'make check'

junit 'reports/\*\*/\*.xml' //junit使用匹配的定义测试xml进行单元测试

}

}

stage('Deploy') {

steps {

sh 'make publish'

}

}

}

}

#### 快速开始使用Pipeline

1）将快速开始示例复制到自己的仓库中并命名为Jenkinsfile

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker 'maven:3.3.3' }

stages {

stage('build') {

steps {

sh 'mvn --version'

}

}

}

}

2）单击Jenkins中的New Item菜单。

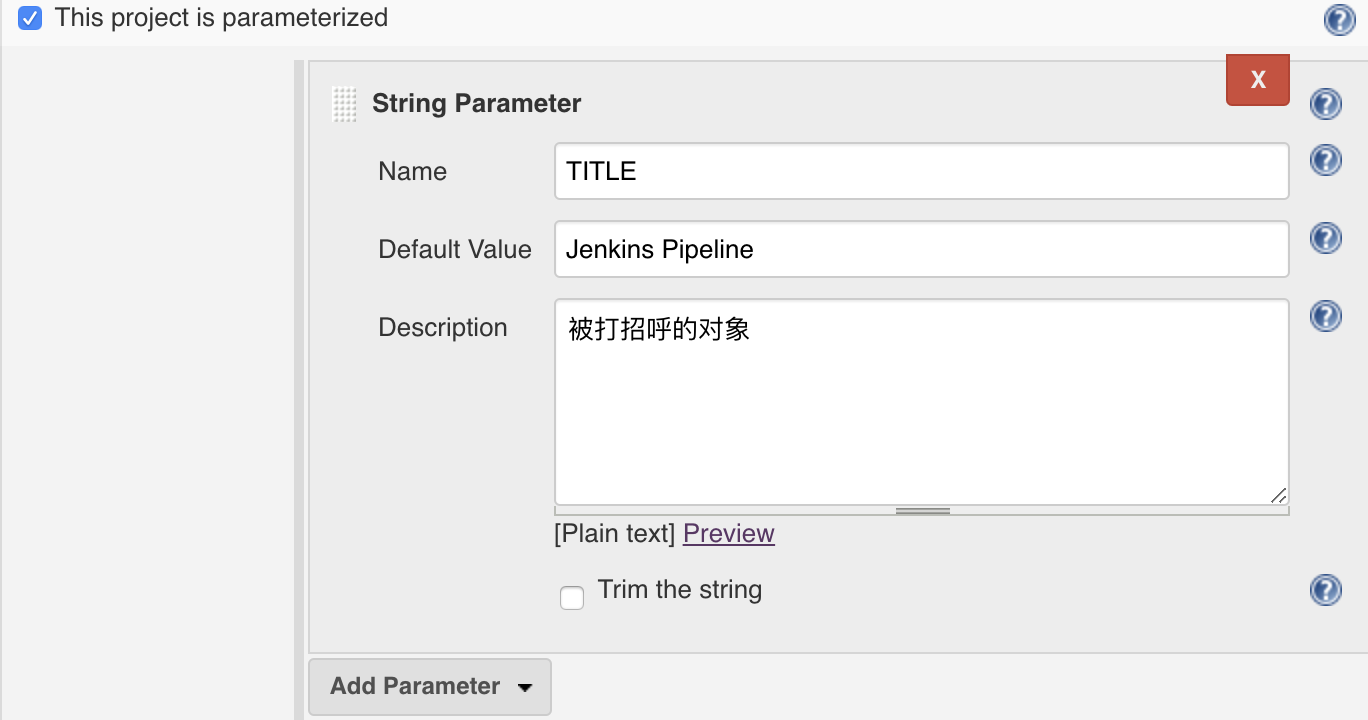
3）为新工程起一个名字，选择Multibranch Pipeline。

流水线与多分支流水线的区别：

* 流水线只能指定一个分支的Jenkinsfile，还可以把脚本直接写到工程中，而多分支流水线可以指定多个分支的Jenkinsfile，但是不能把脚本写在工程中；
* SCM插件也支持多分支流水线，而单分支流水线没有

1. 勾选This project is parameterized，添加参数。

这里加一个字符串参数，这样就可以在Jenkinsfile中就可以去到这个值。



注：添加这个参数是为了让大家先有个参数化构建的准备。

1. 找到Pipeline，定义处可选直接写脚本还是走版本控制。保存

pipeline {

agent any;

environment { //环境变量

GREETING="Hello";

}

stages{

stage('打招呼') {

steps{

sh 'echo "$GREETING $TITLE"'

}

}

}

post { //构建完成后置操作

aborted { //如果构建中断，则执行

echo '构建被中止!'

}

success { //构建成功执行

echo '构建成功!'

}

failure { //构建失败执行

echo '构建失败!'

}

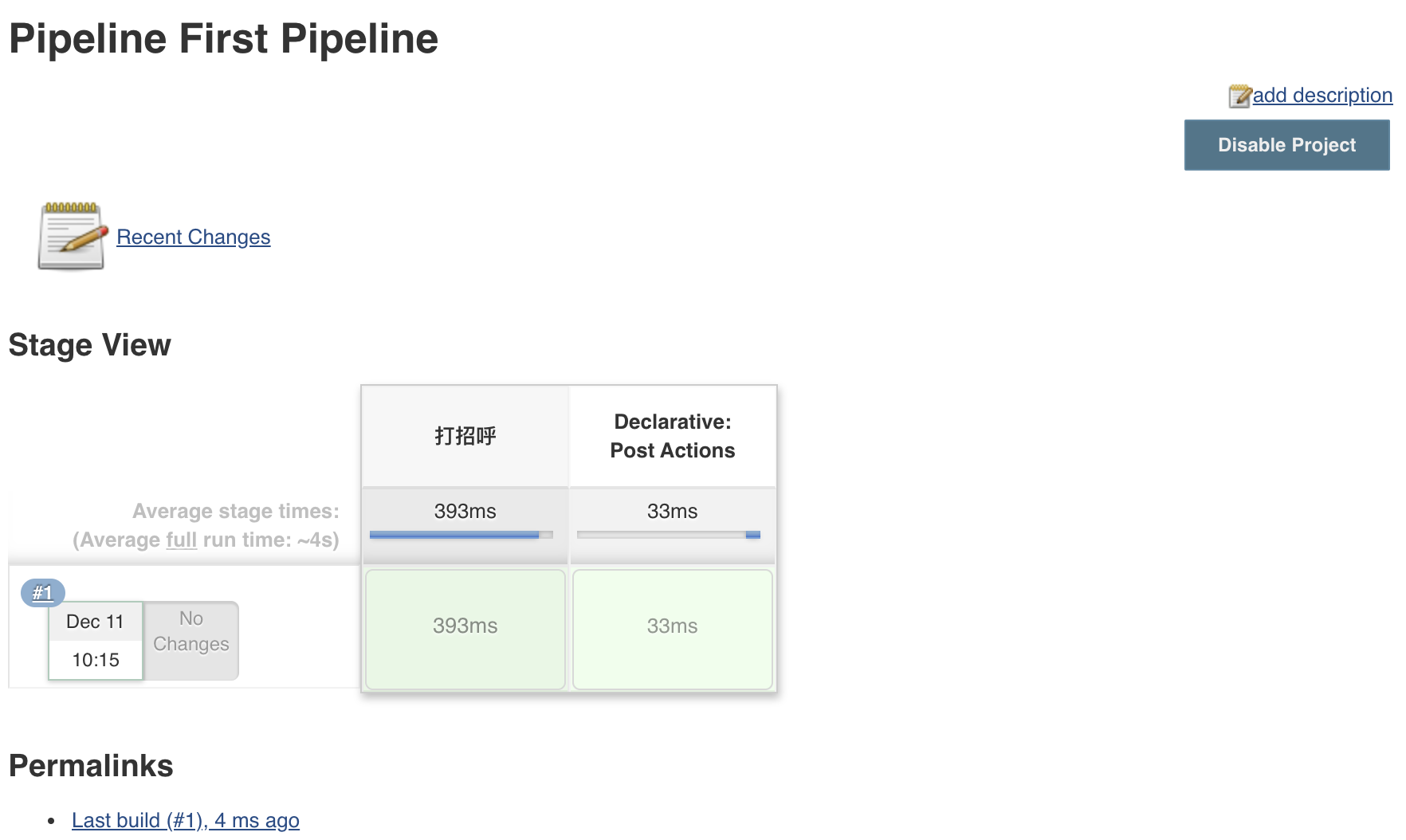
}

}

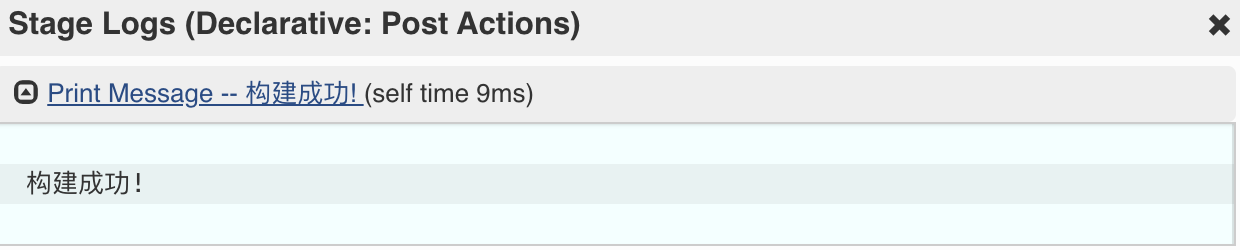
6）构建这个工程，点击Build with Parameters。



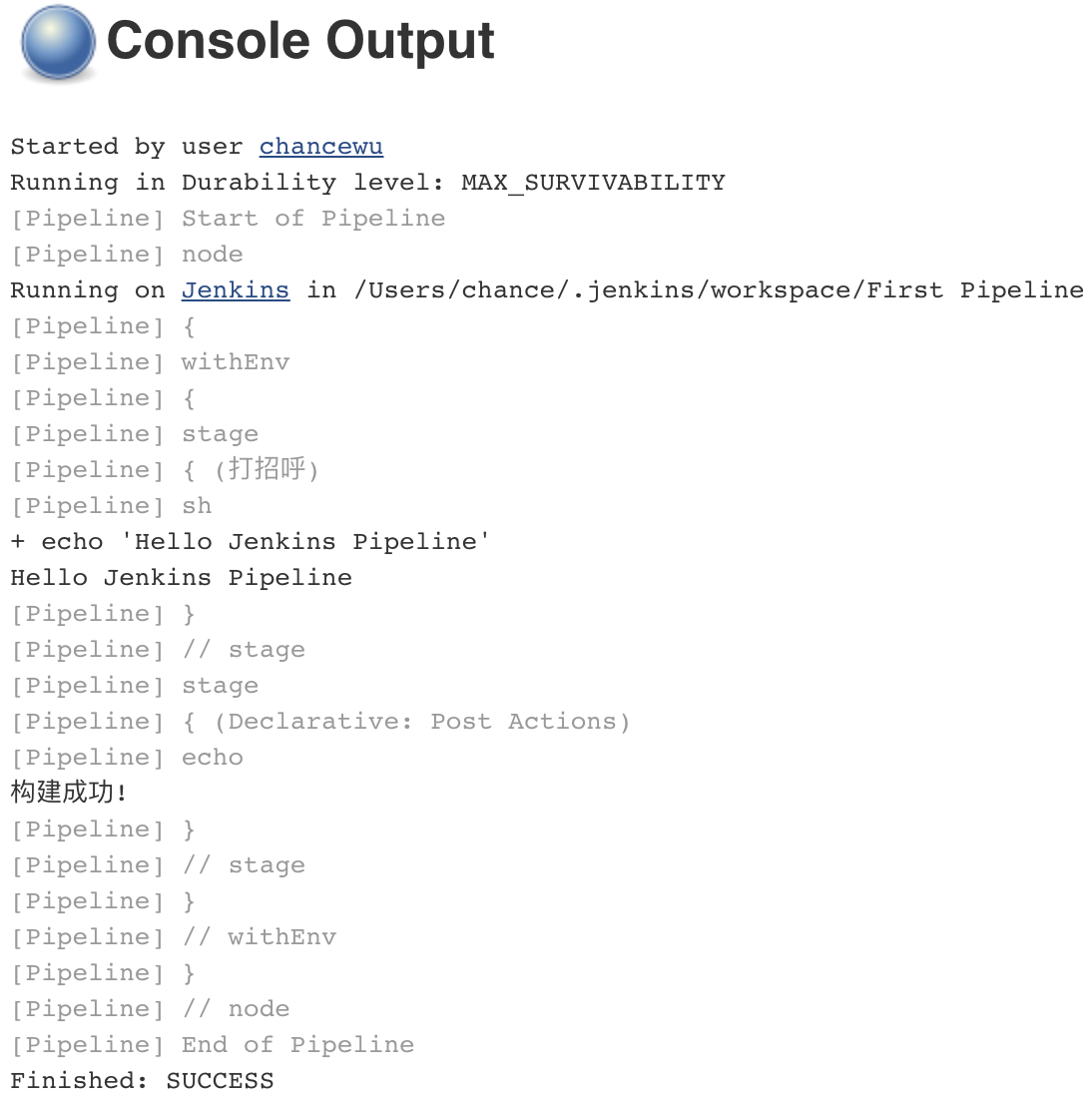
这里看到刚才定义的参数值是可以修改的，然后开始Build。输出如下：



把鼠标放在打招呼下边的绿色框上，点出现的logs。可以看到输出了预期的值，查看下Post Actions的日志：



可以通过左下角的构建版本号，如当前是#1，点进去，选择Console Output查看详细的执行日志：



构建的工作空间在我们指定的/Users/chance/.jenkins/workspace/First Pipeline内。

## 2、创建Hello World流水线

## 3、执行多个step

Pipelines由多个步骤（step）组成，允许你构建、测试和部署应用。Jenkins Pipeline允许您使用一种简单的方式组合多个步骤，以帮助您实现多种类型的自动化构建过程。

可以把“step”看作是一个执行单一动作的单一命令。当一个步骤运行成功时继续运行下一个步骤。当任何一个步骤执行失败时，Pipeline的执行结果也为失败。当所有的步骤都执行完成并且为成功时，Pipeline 的执行结果为成功。

### 3.1 Linux、BSD和Mac OS

在Linux、BSD和Mac OS（类unix）系统中的shell命令，对应于Pipeline中的一个sh step。

Jenkinsfile（Declarative Pipeline）

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'echo "Hello World"'

sh '''

echo "Multiline shell steps works too"

ls -lah

'''

}

}

}

}

Jenkinsfile (Scripted Pipeline)

node {

stage('Build') {

sh 'echo "Hello World"'

sh '''

echo "Multiline shell steps works too"

ls -lah

'''

}

}

### 3.2 Windows

基于Windows的系统使用bat step表示执行批处理命令。

Jenkinsfile（Declarative Pipeline）

pipeline {

agent any

stages {

stage('Build') {

steps {

bat 'set'

}

}

}

}

Jenkinsfile (Scripted Pipeline)

node {

stage('Build') {

bat 'set'

}

}

### 3.3 超时、重试和更多

Jenkins Pipeline提供了很多的step，这些step可以互相组合嵌套，方便地解决像重复执行步骤直到成功（重试）和如果一个步骤执行花费的事件太长则退出（超时）等问题。

Jenkinsfile(Declarative Pipeline)

pipeline {

agent any

stages {

stage('Deploy') {

steps {

retry(3) {

sh './flakey-deploy.sh'

}

timeout(time: 3, unit: 'MINUTES') {

sh './health-check.sh'

}

}

}

}

}

Jenkinsfile(Script Pipeline)

node {

stage('Deploy') {

retry(3) {

sh './flakey-deploy.sh'

}

timeout(time: 3, unit: 'MINUTES') {

sh './health-check.sh'

}

}

}

“Deploy”阶段（stage）重复执行flakey-deploy.sh脚本3次，然后等待health-check.sh脚本最长执行3分钟。如果health-check.sh脚本在3分钟内没有完成，Pipeline将会标记在“Deploy”阶段失败。

内嵌类型的步骤，例如timeout和retry可以包含其他的步骤，包括timeout和retry。比如我们想要重试部署5次，但是总共话费的事件不能超过3分钟。

Jenkinsfile(Declarative Pipeline)

node {

stage('Deploy') {

timeout(time: 3, unit: 'MINUTES') {

retry(5) {

sh './flakey-deploy.sh'

}

}

}

}

Jenkinsfile(Script Pipeline)

node {

stage('Deploy') {

timeout(time: 3, unit: 'MINUTES') {

retry(5) {

sh './flakey-deploy.sh'

}

}

}

}

### 3.4 完成时动作

当Pipeline运行完成时，你可能需要做一些清理工作或者基于Pipeline的运行结果执行不同的操作，这些操作可以放在post部分。

Jenkinsfile(Declarative Pipeline)

pipeline {

agent any

stages {

stage('Test') {

steps {

sh 'echo "Fail!"; exit 1'

}

}

}

post {

always {

echo 'This will always run'

}

success {

echo 'This will run only if successful'

}

failure {

echo 'This will run only if failed'

}

unstable {

echo 'This will run only if the run was marked as unstable'

}

changed {

echo 'This will run only if the state of the Pipeline has changed'

echo 'For example, if the Pipeline was previously failing but is now successful'

}

}

}

Jenkinsfile(Script Pipeline)

node {

try {

stage('Test') {

sh 'echo "Fail!"; exit 1'

}

echo 'This will run only if successful'

} catch (e) {

echo 'This will run only if failed'

// Since we're catching the exception in order to report on it,

// we need to re-throw it, to ensure that the build is marked as failed

throw e

} finally {

def currentResult = currentBuild.result ?: 'SUCCESS'

if (currentResult == 'UNSTABLE') {

echo 'This will run only if the run was marked as unstable'

}

def previousResult = currentBuild.previousBuild?.result

if (previousResult != null && previousResult != currentResult) {

echo 'This will run only if the state of the Pipeline has changed'

echo 'For example, if the Pipeline was previously failing but is now successful'

}

echo 'This will always run'

}

}

## 4、定义执行环境

agent指令告诉Jenkins在哪里以及如何执行Pipeline或者Pipeline子集。所有的Pipeline都需要agent指令。

在执行引擎中，agent指令会引起以下操作的执行：

* 所有在块block中的steps会被Jenkins保存在一个执行队列中。一旦一个执行器executor是可以利用的，这些步骤将会开始执行。
* 一个工作空间workspace将会被分配，工作空间中会包含来自远程仓库的文件和一些用于Pipeline的工作文件。

在Pipeline中可以很容易的运行Docker镜像和容器。Pipeline可以定义命令或者应用运行需要的环境和工具，不需要在执行代理中手动去配置各种各样的系统工具和依赖。这种方式可以让你使用Docker容器工具包中的任何工具。

Jenkinsfile(Declarative Pipeline)

pipeline {

agent {

docker { image 'node:7-alpine' }

}

stages {

stage('Test') {

steps {

sh 'node --version'

}

}

}

}

Jenkinsfile(Script Pipeline)

node {

/\* Requires the Docker Pipeline plugin to be installed \*/

docker.image('node:7-alpine').inside {

stage('Test') {

sh 'node --version'

}

}

}

当执行Pipeline时，Jenkins将会自动运行指定的容器,并执行Pipeline中已经定义好的步骤steps：

[Pipeline] stage

[Pipeline] { (Test)

[Pipeline] sh

[guided-tour] Running shell script

+ node --version

v7.4.0

[Pipeline] }

[Pipeline] // stage

[Pipeline] }

在Pipeline中，混合和搭配不同的容器或者其他代理可以获得更大的灵活性。更多配置选项和信息，可以参考继续“使用环境变量”。

## 5、使用环境变量

环境变量可以像下面的示例设置为全局的，也可以是阶段（stage）级别的。阶段（stage）级别的环境变量只能在定义变量的阶段（stage）使用。

Jenkinsfile(Declarative Pipeline)

pipeline {

agent any

environment {

DISABLE\_AUTH = 'true'

DB\_ENGINE = 'sqlite'

}

stages {

stage('Build') {

steps {

sh 'printenv'

}

}

}

}

Jenkinsfile(Script Pipeline)

node {

withEnv(['DISABLE\_AUTH=true',

'DB\_ENGINE=sqlite']) {

stage('Build') {

sh 'printenv'

}

}

}

这种在Jenkinsfile中定义环境变量的方法对于指令性的脚本定义非常有用方便，比如Makefile文件，可以在Pipeline中配置构建或者测试的环境，然后在Jenkins中运行。

环境变量的另一个常见用途是设置或者覆盖构建或测试脚本中的凭证。因为把凭证信息直接写入Jenkinsfile很显然是一个坏主意，Jenkins Pipeline允许用户快速安全地访问在Jenkinsfile中预定义的凭证信息，并且无需知道它们的值。

环境变量中的凭证信息：

更多信息参考[用户手册](https://jenkins.io/doc/book)中的[凭证信息处理](https://jenkins.io/doc/book/pipeline/jenkinsfile#handling-credentials)。

## 6、记录测试结果和构建包

为了简化操作，只要你的测试运行可以输出测试结果文件，Jenkins就可以记录和汇总这些测试结果。Jenkins通常与junit步骤捆绑在一起，但如果您的测试运行结果无法输出JUnit样式的XML报告，那么还有其他插件可以处理任何广泛使用的测试报告格式。

为了收集我们的测试结果，我们将使用post部分。

Jenkinsfile(Declarative Pipeline)

pipeline {

agent any

stages {

stage('Test') {

steps {

sh './gradlew check'

}

}

}

post {

always {

junit 'build/reports/\*\*/\*.xml'

}

}

}

Jenkinsfile(Script Pipeline)

node {

try{

stage('Test') {

sh './gradlew check'

}

finally {

junit 'build/reports/\*\*/\*.xml'

}

}

这将会获得测试结果，Jenkins会持续跟踪并计算测试的趋势和结果。如果存在失败的测试用例，Pipeline会被标记为“UNSTABLE”，在网页上用黄色表示，这不同于使用红色表示的 “FAILED” 失败状态。

当出现测试失败时，通常可以从 Jenkins 中获取构建结果报告进行本地分析和测试。Jenkins 内置支持存储构建结果报告，在 Pipeline 执行期间生成记录文件。

通过 archiveArtifacts 步骤和文件匹配表达式可以很容易的完成构建结果记录和存储，如下例所示：

Jenkinsfile(Declarative Pipeline)

pipeline {

agent any

stages {

stage('Build') {

steps {

sh './gradlew build'

}

}

stage('Test') {

steps {

sh './gradlew check'

}

}

}

post {

always {

archiveArtifacts artifacts: 'build/libs/\*\*/\*.jar', fingerprint: true

junit 'build/reports/\*\*/\*.xml'

}

}

}

Jenkinsfile(Scripted Pipeline)

node {

try{

stage('Test') {

sh './gradlew check'

}

finally {

archiveArtifacts artifacts: 'build/libs/\*\*/\*.jar', fingerprint: true

junit 'build/reports/\*\*/\*.xml'

}

}

如果在archiveArtifacts步骤中指定了多个参数，那么每个参数的名称必须在步骤代码中明确指定，即文件的路径、文件名和fingerprint三个参数。如果您只需指定文件的路径和文件名，那么你可以省略参数名称artifacts，例如：archiveArtifacts 'build/libs/\*/.jar'

## 7、清理和通知

因为post部分保证在Pipeline结束的时候运行，所以我们可以添加通知或者其他的步骤去完成清理、通知或者其他的Pipeline结束任务。

Jenkinsfile(Declarative Pipeline)

pipeline {

agent any

stages {

stage('No-op') {

steps {

sh 'ls'

}

}

}

post {

always {

echo 'One way or another, I have finished'

deleteDir() /\* clean up our workspace \*/

}

success {

echo 'I succeeeded!'

}

unstable {

echo 'I am unstable :/'

}

failure {

echo 'I failed :('

}

changed {

echo 'Things were different before...'

}

}

}

Jenkinsfile(Scripted Pipeline)

node {

try {

stage('No-op') {

sh 'ls'

}

}

catch (exc) {

echo 'I failed'

}

finally {

if (currentBuild.result == 'UNSTABLE') {

echo 'I am unstable :/'

} else {

echo 'One way or another, I have finished'

}

}

}

有很多方法可以发送通知，下面是一些示例展示了如何通过电子邮件、Hipchat room或者Slack channel发送Pipeline的相关信息。

### 7.1 电子邮件

post {

failure {

mail to: 'team@example.com',

subject: "Failed Pipeline: ${currentBuild.fullDisplayName}",

body: "Something is wrong with ${env.BUILD\_URL}"

}

}

### 7.2 Hipchat

post {

failure {

hipchatSend message: "Attention @here ${env.JOB\_NAME} #${env.BUILD\_NUMBER} has failed.",

color: 'RED'

}

}

### 7.3 Slack

post {

success {

slackSend channel: '#ops-room',

color: 'good',

message: "The pipeline ${currentBuild.fullDisplayName} completed successfully."

}

}

当Pipeline失败、不稳定甚至是成功时，团队都会受到通知。

## 8、部署

大多数最基本的持续交付Pipeline至少会三个阶段：Build构建、Test测试和Deploy部署，这些阶段被定义在Jenkinsfile中。这小结主要关注部署阶段，但应该指出稳定的构建和测试阶段是任何部署活动的重要前提。

Jenkinsfile(Declarative Pipeline)

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building'

}

}

stage('Test') {

steps {

echo 'Testing'

}

}

stage('Deploy') {

steps {

echo 'Deploying'

}

}

}

}

Jenkinsfile(Scripted Pipeline)

node {

stage('Build') {

echo 'Building'

}

stage('Test') {

echo 'Testing'

}

stage('Deploy') {

echo 'Deploying'

}

}

### 8.1 阶段即为部署环境

一个常见的模式是扩展阶段的数量以获取额外的部署环境信息，如“staging”或者“production”，如下例所示。

stage('Deploy - Staging') {

steps {

sh './deploy staging'

sh './run-smoke-tests'

}

}

stage('Deploy - Production') {

steps {

sh './deploy production'

}

}

在这个示例中，假定./run-smoke-tests脚本所运行的冒烟测试足以保证或者验证可以发布到生产环境。这种可以自动部署代码一直到生产环境的Pipeline可以认为是“持续部署”的一种实现。虽然这是一个伟大的想法，但是有很多理由表明“持续部署”不是一种很好的实践，即便如此，这种方式仍然可以享有“持续交付”带来的好处。Jenkins Pipeline可以很容易支持两者。

### 8.2 人工确认

通常在stage之间，特别是不同环境阶段之间，可能需要人工确认是否可以继续运行。例如，判断应用程序是否在一个足够好的状态下进入到生产环境阶段。这可以使用input步骤完成。在下面的例子中，“Sanity check”阶段会等待人工确认，并且在没有人工确认的情况下不会继续执行。

Jenkinsfile(Declarative Pipeline)

pipeline {

agent any

stages {

/\* "Build" and "Test" stages omitted \*/

stage('Deploy - Staging') {

steps {

sh './deploy staging'

sh './run-smoke-tests'

}

}

stage('Sanity check') {

steps {

input "Does the staging environment look ok?"

}

}

stage('Deploy - Production') {

steps {

sh './deploy production'

}

}

}

}

Jenkinsfile(Scripted Pipeline)

node {

/\* "Build" and "Test" stages omitted \*/

stage('Deploy - Staging') {

sh './deploy staging'

sh './run-smoke-tests'

}

stage('Sanity check') {

input "Does the staging environment look ok?"

}

stage('Deploy - Production') {

sh './deploy production'

}

}

入门部分介绍了Jenkins和Jenkins Pipeline的基本使用。由于Jenkins是非常容易扩展的，它可以被修改和配置去处理任何类型的自动化。

# 教程

## 1、用Maven构建Java应用

展示如何使用Jenkins编排并构建一个使用Maven管理的简单Java应用程序。

如果你是使用Maven的Java开发人员，并且对CI/CD概念不熟悉，或者你可能熟悉这些概念，但不知道如何使用Jenkins实现构建应用程序，那么本教程适合你。

将在Github示例仓库中获取一个简单Java应用，该应用还包含若干测试，用于检查main方法。测试结果保存在JUnit XML报告中。

### 1.1 在Docker中运行Jenkins

docker run \

-u root \

--rm \ **(1可选)**

-d \ **(2可选)**

-p 8080:8080 \ **(3)**

-p 50000:50000 \ **(4可选)**

-v jenkins-data:/var/jenkins\_home \ **(5可选，强烈建议)**

-v /var/run/docker.sock:/var/run/docker.sock \ **(6可选)**

jenkinsci/blueocean **(7可选)**

1. jenkinsci/blueocean关闭时自动删除Docker容器。
2. jenkinsci/blueocean在后台运行容器（即“分离模式”）并输出容器ID。如果不指定此选项，则在终端窗口中输出正在运行此容器的Docker日志。
3. 映射jenkinsci/blueocean容器的端口8080到主机上的端口8080。第一个数字代表主机上的端口，而后一个代表容器的端口。因此，如果你为此选项指定-p 49000:8080，你将通过端口49000访问主机上的Jenkins。
4. 将jenkinsci/blueocean容器的端口50000映射到主机上的端口50000。如果您在其他机器上设置了一个或多个基于JNLP的Jenkins代理程序，而这些代理程序又与 jenkinsci/blueocean 容器交互（充当“主”Jenkins服务器，或者简称为“Jenkins主”），则这是必需的。默认情况下，基于JNLP的Jenkins代理通过TCP端口50000与Jenkins主站进行通信。您可以通过“ 配置全局安全性” 页面更改Jenkins主服务器上的端口号。如果您要将您的Jenkins主机的JNLP代理端口的TCP端口 值更改为51000（例如），那么您需要重新运行Jenkins（通过此 docker run …​命令）并指定此“发布”选项 -p 52000:51000，其中最后一个值与Jenkins master上的这个更改值相匹配，第一个值是Jenkins主机的主机上的端口号，通过它，基于JNLP的Jenkins代理与Jenkins主机进行通信 - 例如52000。
5. 映射在容器中的/var/jenkins\_home 目录到具有名字 jenkins-data 的volume。 如果这个卷不存在，那么这个 docker run 命令会自动为你创建卷。 如果您希望每次重新启动Jenkins（通过此 docker run … 命令）时保持Jenkins状态，则此选项是必需的 。 如果你没有指定这个选项，那么在每次重新启动后，Jenkins将有效地重置为新的实例。注意:所述的 jenkins-data 卷也可以 docker volume create命令创建： docker volume create jenkins-data 代替映射 /var/jenkins\_home 目录转换为Docker卷，还 可以将此目录映射到计算机本地文件系统上的目录。 例如，指定该选项 -v $HOME/jenkins:/var/jenkins\_home 会将容器的 /var/jenkins\_home 目录映射 到 本地计算机上目录中的 jenkins 子目录， 该$HOME目录通常是 /Users/<your-username>/jenkins 或/home/<your-username>/jenkins 。
6. /var/run/docker.sock 表示Docker守护程序通过其监听的基于Unix的套接字。该映射允许 jenkinsci/blueocean 容器与Docker守护进程通信，如果 jenkinsci/blueocean 容器需要实例化其他Docker容器，则该守护进程是必需的。如果运行声明式管道，其语法包含agent部分用 docker。（例如， agent { docker { … } } 此选项是必需的。在Pipeline Syntax 页面上阅读更多关于这个的信息 。）
7. jenkinsci/blueocean Docker镜像本身。如果此镜像尚未下载，则此 docker run 命令 将自动为您下载镜像。此外，如果自上次运行此命令后发布了此镜像的任何更新， 则再次运行此命令将自动为您下载这些已发布的镜像更新。 注意：这个Docker镜像也可以使用以下 docker pull命令独立下载（或更新） ： docker pull jenkinsci/blueocean

访问<http://localhost:8080>，之后的步骤与普通安装Jenkins一致。

### 1.2 访问Jenkins/BlueOcean Docker容器

希望或需要使用docker exec命令通过一个终端/命令提示符来访问Jenkins/Blue Ocean Docker 容器, 你可以添加如 —name jenkins-tutorials 选项(与上面的docker run), 这将会给Jenkins/Blue Ocean Docker容器一个名字"jenkins-tutorials"。

这意味着你可以通过 docker exec 命令访问Jenkins/Blue Ocean 容器(通过一个单独的终端/命令提示窗口) ，例如:

docker exec -it jenkins-tutorials bash

## 2、用npm构建Node.js和React应用

## 3、用PyInstaller构建Python应用

## 4、用Blue Ocean创建流水线

## 5、构建一个多分支的Pipeline项目

# 用户手册

* [用户手册概述](https://jenkins.io/zh/doc/book/getting-started)
* [安装Jenkins](https://jenkins.io/zh/doc/book/installing)
* [使用 Jenkins](https://jenkins.io/zh/doc/book/using)

本章节包含一般Jenkins用户（任何技能水平）关于Jenkins使用的主题，这些用例不包含Jenkins核心特征：Pipeline和Blue Ocean。

如果想通过JenkinsFile或者Blue Ocean创建、配置Pipeline项目，或者希望了解更多Jenkins核心特性，参阅Pipeline和Blue Ocean章节。

* [流水线](https://jenkins.io/zh/doc/book/pipeline)
* [Blue Ocean](https://jenkins.io/zh/doc/book/blueocean)
* [管理Jenkins](https://jenkins.io/zh/doc/book/managing)
* [系统管理](https://jenkins.io/zh/doc/book/system-administration)
* [规模 Jenkins](https://jenkins.io/zh/doc/book/scaling)
* [附录](https://jenkins.io/zh/doc/book/appendix)
* [术语表](https://jenkins.io/zh/doc/book/glossary)

# 参考资料

* [流水线语法参考](https://jenkins.io/zh/doc/book/pipeline/syntax)
* [流水线步骤参考](https://jenkins.io/zh/doc/pipeline/steps)
* [LTS 升级指导](https://jenkins.io/zh/doc/upgrade-guide)

